Test Environment Toolkit

TETware Programmers Guide
Revision 1.1
TET3-PG-1.1

Released: 31st July 1997

X/Open Company Limited

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

Copyright © 1992, 1993, 1996, 1997 X/Open Company Limited
Copyright © 1992 Open Software Foundation
Copyright © 1992 Unix International
Copyright © 1993 Information-Technology Promotion Agency, Japan
Copyright © 1994, 1995 UniSoft Ltd.

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

X/Open and the 'X' symbol are trademarks of X/Open Company Limited in the UK and other countries.

UNIX® is a registered trademark in the U.S. and other countries, licenced exclusively through X/Open Company Ltd.

Win32™, Windows NT™ and Windows 95™ are registered trademarks of Microsoft Corporation.

This document is produced by UniSoft Ltd. at:

150 Minories
LONDON
EC3N 1LS
United Kingdom

CONTENTS

1. Introduction 1
   1.1 Preface 1
   1.2 Product definition 1
   1.3 Audience 1
   1.4 Conventions used in this guide 2
   1.5 Related documents 2
   1.6 Problem reporting 2

2. Testing structure 3
   2.1 Introduction 3
   2.2 Test suite processing 3
   2.3 Directory structure 3
   2.4 Test case structure 4
      2.4.1 Introduction 4
      2.4.2 Test Case Manager 4
      2.4.3 API-conforming test cases 4
      2.4.4 Non API-conforming test cases 5
   2.5 Test suite structure 6
      2.5.1 Introduction 6
      2.5.2 Required files and utilities 6
      2.5.3 Optional files and utilities 8

3. The Test Case Controller 11
   3.1 Introduction 11
   3.2 Modes of operation 11
   3.3 Initial processing 11
   3.4 Build mode processing 13
   3.5 Execute mode processing 16
   3.6 Clean mode processing 19
   3.7 Rerun and resume processing 20
      3.7.1 Introduction 20
      3.7.2 Resume processing 21
         3.7.2.1 Description 21
         3.7.2.2 Processing a **`parallel`** directive in resume mode 22
         3.7.2.3 Processing a **`random`** directive in resume mode 22
         3.7.2.4 Processing a **`timed_loop`** directive in resume mode 22
      3.7.3 Rerun processing 22
         3.7.3.1 Description 22
         3.7.3.2 Processing a **`random`** directive in rerun mode 23
         3.7.3.3 Processing a **`timed_loop`** directive in rerun mode 23
   3.8 Communication variables 23
   3.9 Journal entries 24
   3.10 Locking 24
   3.11 Using **`tcc`** to process a test suite on a read-only file system 25

4. The scenario file 27

4.1 Introduction . . . 27
4.2 The scenario language . . . 27
4.2.1 Introduction . . . 27
4.2.2 Scenario lines . . . 27
4.2.3 The scenario name . . . 28
4.2.4 Simple scenario elements . . . 28
4.2.4.1 Introduction . . . 28
4.2.4.2 Scenario information line . . . 28
4.2.4.3 Test case name . . . 28
4.2.4.4 Referenced scenario name . . . 29
4.2.4.5 File name . . . 30
4.2.5 Scenario directives . . . 30
4.2.5.1 Introduction . . . 30
4.2.5.2 **`repeat`** – process scenario elements a specified number of times . . . 31
4.2.5.3 **`timed_loop`** – process scenario elements until a specified period of time expires . . . 32
4.2.5.4 **`random`** – process a test case selected at random . . . 33
4.2.5.5 **`parallel`** – process scenario elements in parallel . . . 34
4.2.5.6 **`group`** – process scenario elements in parallel . . . 37
4.2.5.7 **`remote`** – process test cases on remote systems . . . 37
4.2.5.8 **`distributed`** – process distributed test cases . . . 38
4.2.5.9 **`include`** – process scenario elements listed in an include file . . . 39
4.2.6 Directive groups . . . 39
4.2.7 Directive nesting rules . . . 40
4.3 Example scenarios . . . 41

5. Configuration files . . . 53
5.1 Introduction . . . 53
5.2 Use of configuration variables . . . 53
5.3 Configuration file format . . . 53
5.4 Configuration variable processing in TETware-Lite . . . 54
5.5 Configuration variable processing in Distributed TETware . . . 54
5.6 Configuration variables which modify TETware's operation . . . 57
5.7 Distributed configuration variables used by Distributed TETware . . . 60

6. Other test suite files . . . 63
6.1 Introduction . . . 63
6.2 Result codes . . . 63
6.2.1 Description . . . 63
6.2.2 Result code definitions . . . 63
6.2.3 File format . . . 64
6.2.4 Example results code file . . . 65
6.3 System definitions . . . 65
6.3.1 Description . . . 65
6.3.2 File format . . . 65
6.3.3 Example **`systems`** files . . . 66

7. The Test Case Manager . . . 67

7.1 Introduction . . . 67
7.2 TCM flow of control . . . 67
7.3 C and C++ TCM options . . . 69
7.4 C and C++ TCMs in Distributed TETware . . . 69
7.5 Portability . . . 70

8. The C API . . . 73
8.1 Introduction . . . 73
8.2 C language binding . . . 73
8.3 TCC dependencies . . . 74
8.4 Test case structure and management . . . 75
8.4.1 Introduction . . . 75
8.4.2 Static test case interface – the **tet_testlist[]** array . . . 75
8.4.3 Dynamic test case interface – **tet_getmaxic()**, **tet_getminic()**, **tet_isdefic()**, **tet_gettpcount()**, **tet_gettestnum()** and **tet_invoketp()** . . . 76
8.4.4 **tet_startup** and **tet_cleanup** . . . 79
8.4.5 **tet_thistest**, **tet_nosigreset** and **tet_pname** . . . 79
8.5 Insulating from the test environment . . . 81
8.6 Error handling and reporting . . . 82
8.6.1 Introduction . . . 82
8.6.2 **tet_errno** . . . 82
8.6.3 **tet_errlist[]** and **tet_nerr** . . . 83
8.7 Making journal entries . . . 84
8.7.1 Introduction . . . 84
8.7.2 **tet_setcontext()** and **tet_setblock()** . . . 84
8.7.3 **tet_infoline()**, **tet_minfoline()**, **tet_printf()** and **tet_vprintf()** . . . 85
8.7.4 **tet_result()** . . . 86
8.8 Cancelling test purposes . . . 87
8.8.1 Introduction . . . 87
8.8.2 **tet_delete()** . . . 87
8.8.3 **tet_reason()** . . . 88
8.9 Accessing configuration variables . . . 89
8.9.1 Introduction . . . 89
8.9.2 **tet_getvar()** . . . 89
8.10 Generating and executing processes . . . 90
8.10.1 Introduction . . . 90
8.10.2 **tet_fork()**, **tet_exec()** and **tet_child** . . . 90
8.10.3 **tet_spawn()** . . . 92
8.10.4 **tet_wait()** . . . 93
8.10.5 **tet_kill()** . . . 93
8.11 Executed process functions . . . 94
8.11.1 Introduction . . . 94
8.11.2 **tet_main()** . . . 94
8.11.3 **tet_exit()** and **tet_logoff()** . . . 95
8.12 Test case synchronisation . . . 96
8.12.1 Introduction . . . 96
8.12.2 **tet_remsync()** . . . 96

8.12.3 **tet_sync()** and **tet_msync()** . . . 99
8.12.4 Control over sync error reporting . . . 100
8.13 Remote system information . . . 101
8.13.1 Introduction . . . 101
8.13.2 **tet_remgetlist()** . . . 101
8.13.3 **tet_remgetsys()** . . . 101
8.13.4 **tet_getsysbyid()** . . . 102
8.13.5 **tet_remtime()** . . . 102
8.14 Remote process control . . . 104
8.14.1 Introduction . . . 104
8.14.2 **tet_remexec()** . . . 104
8.14.3 **tet_remwait()** . . . 105
8.14.4 **tet_remkill()** . . . 107

9. The C++ API . . . 109
9.1 Introduction . . . 109
9.2 C++ language binding . . . 109
9.3 Using the C++ language binding . . . 110

10. The Thread-safe C and C++ APIs . . . 111
10.1 Introduction . . . 111
10.2 C language binding . . . 111
10.3 C++ language binding . . . 112
10.4 Functions that are specific to the Thread-safe APIs . . . 112
10.4.1 Introduction . . . 112
10.4.2 **tet_thr_create()** and **tet_pthread_create()** . . . 112
10.4.3 **tet_beginthreadex()** . . . 113
10.4.4 **tet_fork1()** . . . 114
10.5 Unavailable interfaces . . . 115
10.6 API differences . . . 115
10.6.1 Introduction . . . 115
10.6.2 Thread-specific data . . . 115
10.6.3 Block and sequence numbers . . . 115
10.6.4 **tet_spawn()** . . . 115
10.6.5 **tet_fork()** . . . 115
10.7 TCM differences . . . 116
10.7.1 Introduction . . . 116
10.7.2 Clean-up of left-over threads on UNIX systems . . . 116
10.7.3 Dealing with left-over threads on Win32 systems . . . 116
10.7.4 Signal handling . . . 117
10.8 Synchronisation requests in multi-threaded test cases . . . 117

11. The Shell and Korn Shell APIs . . . 119
11.1 Introduction . . . 119
11.2 Shell language binding . . . 119
11.3 Korn Shell language binding . . . 119
11.4 TCC dependencies . . . 120
11.5 Test case structure and management . . . 120
11.5.1 Introduction . . . 120
11.5.2 **iclist**, **ic*n***, **tet_startup** and **tet_cleanup** . . . 120

11.5.3 **tet_thistest** . . . 121
11.6 Insulating from the test environment . . . 121
11.7 Making journal entries . . . 122
11.7.1 Introduction . . . 122
11.7.2 **tet_setcontext** and **tet_setblock** . . . 122
11.7.3 **tet_infoline** . . . 123
11.7.4 **tet_result** . . . 123
11.8 Canceling test purposes . . . 124
11.8.1 Introduction . . . 124
11.8.2 **tet_delete** . . . 124
11.8.3 **tet_reason** . . . 124
11.9 Manipulating configuration variables . . . 124
11.10 Generation and execution of processes . . . 124
11.11 Executed process support . . . 125

12. The Perl API . . . 127
12.1 Introduction . . . 127
12.2 Description . . . 127

13. Test reporting and journaling . . . 129
13.1 Making journal entries . . . 129
13.1.1 Entries from the API . . . 129
13.1.2 Entries from test purposes . . . 130
13.2 Journal files . . . 130
13.2.1 Description . . . 130
13.2.2 Journal line parameters . . . 130
13.2.3 Journal line descriptions . . . 131
13.3 Result file processing . . . 132
13.3.1 Execution results from an API-conforming test case . . . 132
13.3.2 Processing results from a non API-conforming test case . . . 132
13.3.3 Processing results from a non-distributed API-conforming test case . . . 132
13.3.4 Processing results from a distributed API-conforming test case . . . 133
13.4 Support for user-supplied report writers . . . 133

14. Writing a C language API-conforming test suite . . . 135
14.1 Introduction . . . 135
14.2 Defining a test suite . . . 135
14.3 Defining common test case functions and variables . . . 138
14.4 Initialising test cases . . . 138
14.5 Controlling and recording test case execution results . . . 139
14.6 Results that must be verified by the user . . . 141
14.7 Child processes and subprograms . . . 142
14.8 Cleaning up test cases . . . 145

15. Writing a Shell language API-conforming test suite . . . 147
15.1 Introduction . . . 147
15.2 Defining a test suite . . . 147
15.3 Defining common test case functions and variables . . . 149
15.4 Initialising test cases . . . 152
15.5 Controlling and recording test case execution results . . . 152

15.6 Results that must be verified by the user . . . 155
15.7 Cleaning up test cases . . . 155

16. The distributed demonstration test suite . . . 157
16.1 Introduction . . . 157
16.2 Test suite files . . . 157
16.2.1 The **systems** file . . . 158
16.2.2 The **tet_code** file . . . 159
16.2.3 The **tet_scen** file . . . 159
16.2.4 The **tetbuild.cfg** file . . . 160
16.2.5 The **tetclean.cfg** file . . . 161
16.2.6 The **tetexec.cfg** file . . . 161
16.2.7 The **tetdist.cfg** file . . . 162
16.2.8 The **makefile** file . . . 163
16.2.9 The **tc1.c** file . . . 164
16.2.10 The **tc2.c** file . . . 164
16.2.11 The **tc3.c** file . . . 164

A. The TETware end-user licence . . . 169

B. Example C language API test suite source files . . . 171
B.1 Introduction . . . 171
B.2 **tet_code** . . . 171
B.3 **install** . . . 171
B.4 **cleantool** . . . 171
B.5 **tet_scen** . . . 171
B.6 **tetbuild.cfg** . . . 172
B.7 **tetexec.cfg** . . . 172
B.8 **tetclean.cfg** . . . 172
B.9 Makefile for **chmod-tc.c** . . . 172
B.10 **chmod-tc.c** . . . 173
B.11 Makefile for **fileno-tc.c** . . . 176
B.12 **fileno-tc.c** . . . 176
B.13 **fileno-t4.c** . . . 181
B.14 Makefile for **stat-tc.c** . . . 182
B.15 **stat-tc.c** . . . 182
B.16 Makefile for **uname-tc.c** . . . 189
B.17 **uname-tc.c** . . . 189

C. Example Shell API test suite source files . . . 191
C.1 Introduction . . . 191
C.2 **tet_code** . . . 191
C.3 **install** . . . 192
C.4 **buildtool** . . . 192
C.5 **cleantool** . . . 192
C.6 **tet_scen** . . . 192
C.7 **tetbuild.cfg** . . . 193
C.8 **tetexec.cfg** . . . 193
C.9 **tetclean.cfg** . . . 193
C.10 **shfuncs** — common functions used in the Shell API test suite . . . 193
C.11 Makefile for **chmod-tc.sh** . . . 195
C.12 **chmod-tc.sh** . . . 195

C.13 Makefile for **uname-tc.sh** . . . 197
C.14 **uname-tc.sh** . . . 197

D. Example distributed test case source files . . . 199
D.1 Introduction . . . 199
D.2 Files supplied on the master system . . . 199
D.2.1 **tet_code** . . . 199
D.2.2 **tet_scen** . . . 199
D.2.3 **tetbuild.cfg** . . . 200
D.2.4 **tetclean.cfg** . . . 200
D.2.5 **tetdist.cfg** . . . 201
D.2.6 **tetexec.cfg** . . . 201
D.2.7 **ts/makefile** . . . 201
D.2.8 **ts/tc1.c** . . . 202
D.2.9 **ts/tc2.c** . . . 202
D.2.10 **ts/tc3.c** . . . 203
D.3 Files supplied on the slave system . . . 204
D.3.1 **tetbuild.cfg** . . . 204
D.3.2 **tetclean.cfg** . . . 205
D.3.3 **tetexec.cfg** . . . 205
D.3.4 **ts/makefile** . . . 205
D.3.5 **ts/tc1.c** . . . 206
D.3.6 **ts/tc2.c** . . . 206
D.3.7 **ts/tc3.c** . . . 207
D.4 Files supplied both systems . . . 208
D.4.1 **systems** . . . 208
D.4.2 **ts/ntbuild.ksh** . . . 208
D.4.3 **ts/ntclean.ksh** . . . 209

E. Scenario language syntax summary . . . 211

F. Conceptual models used by TETware . . . 215
F.1 Introduction . . . 215
F.2 TETware-Lite conceptual model . . . 216
F.3 Distributed TETware conceptual model – local system with test cases . . . 217
F.4 Distributed TETware conceptual model – local system without test cases . . . 218
F.5 Distributed TETware conceptual model – remote system as master . . . 219
F.6 Distributed TETware conceptual model – remote system as slave . . . 220

G. Background and goals . . . 221
G.1 Introduction . . . 221
G.2 Previous TET implementations . . . 221
G.2.1 The Test Environment Toolkit . . . 221
G.2.2 The Distributed Test Environment Toolkit . . . 221
G.2.3 The Extended Test Environment Toolkit . . . 222
G.2.4 The Distributed Test Environment Toolkit Version 2 . . . 222
G.3 TETware . . . 222
G.4 Relationship between TETware and its predecessors . . . 223

H. Terminology . . . 225

H.1 Test case types 225
H.2 Glossary 225

LIST OF FIGURES

Figure 1. Test case interaction 5

Figure 2. Test case processing in build mode 15

Figure 3. Test case processing in execute mode 18

Figure 4. Test case processing in clean mode 20

Figure 5. Processing test cases in sequence 42

Figure 6. Processing test cases in parallel 43

Figure 7. Processing multiple instances of a single test case in parallel 44

Figure 8. Processing referenced scenario elements in parallel when in dTET2 mode 45

Figure 9. Processing referenced scenario elements in parallel when in ETET mode 46

Figure 10. Processing a `repeat` directive in execute mode 47

Figure 11. Processing `repeat` directives in parallel 48

Figure 12. Processing randomly selected test cases in parallel for a specified period of time 50

Figure 13. Processing remote and distributed test cases 51

Figure 14. Configuration variable processing in TETware-Lite 54

Figure 15. Configuration variable processing in Distributed TETware 56

Figure 16. Precedence of result code definitions 64

Figure 17. Directory structure for the example C language test suite 136

Figure 18. Directory structure for the example Shell language test suite 148

Figure 19. Directory structure for the distributed demonstration test suite 158

Figure 20. TETware-Lite conceptual model 216

Figure 21. Distributed TETware conceptual model – local system with test cases 217

Figure 22. Distributed TETware conceptual model – local system without test cases 218

Figure 23. Distributed TETware conceptual model – remote system as master 219

Figure 24. Distributed TETware conceptual model – remote system as slave 220

Figure 25. Relationship between TETware and its predecessors 223

1. Introduction

1.1 Preface

This document is the TETware Programmers Guide.

TETware is implemented on UNIX operating systems and also on the Windows NT and Windows 95 operating systems. It includes all of the functionality of the Test Environment Toolkit Release 1.10 (TET), the Distributed Test Environment Toolkit Version 2 Release 2.3 (dTET2) and the Extended Test Environment Toolkit Release 1.10.3 (ETET), together with a number of new features.

Throughout this document, the Windows NT and Windows 95 operating systems are referred to collectively as **Win32 systems**. The individual system names are only used when it is necessary to distinguish between them.

1.2 Product definition

TETware is a set of tools for the development and execution of system and unit tests. The goal behind creating TETware and its predecessors is to produce a test driver that accommodates present and future testing needs of the test development community. To achieve this goal, input from a wide sample of the test development community has been used for the specification and development of TETware's functionality and interfaces. A short account of the history of TETware and its predecessors is presented in the appendix entitled ''Background and goals'' at the end of this guide.

TETware is available in one of two versions. One version is called **TETware-Lite** and is able to process non-distributed test cases on a single computer system. The other version is called **Distributed TETware** and is able to process both distributed and non-distributed test cases on the local system and on one or more remote systems.

An overview of TETware, some simple architecture diagrams, and a description of what constitutes a **distributed** or a **non-distributed** test case, are presented in the chapter entitled ''TETware overview'' in the TETware User Guide. Diagrams illustrating the conceptual models used by TETware are presented in the appendix entitled ''Conceptual models used by TETware'' at the end of this guide.

TETware-Lite is supported on UNIX systems and on the Windows NT and Windows 95 operating systems. Distributed TETware is supported on UNIX systems and on the Windows NT operating system.

1.3 Audience

This document is intended to be read by test suite authors who will write or adapt test programs to run under the control of TETware.

Software testing engineers and system administrators should refer to the TETware User Guide for information about how to run TETware and to the TETware Installation Guide for information about how to install TETware on their computer systems.

1.4 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.

Long lines in some examples and computer-generated output have been folded at a \ character for formatting purposes. If you type such an example, you should type it in all on one line and omit the \ character.

1.5 Related documents

Refer to the following documents for additional information about TETware:

- *Test Environment Toolkit: TETware Installation Guide*
  There is one version of this document for each operating system family on which TETware is implemented.
- *Test Environment Toolkit: TETware User Guide*

In addition, the TETware Release Notes contain important information about how to install and use TETware. You should read the release notes thoroughly before attempting to install and use each new release of TETware.

1.6 Problem reporting

If you have subscribed to TETware support and you encounter a problem while installing and using TETware, you can send a support request by electronic mail to the address given in the TETware Release Notes. Please follow the instructions contained in the release notes about how to submit such a request; in particular, please be sure to include all the information asked for by these instructions when submitting the request.

2. Testing structure

2.1 Introduction

This chapter introduces the structure of a test suite that is to be processed by TETware. Examples of practical test suites which use this structure are presented in the chapters entitled ‘‘Writing a C language API-conforming test suite’’, ‘‘Writing a Shell language API-conforming test suite’’ and ‘‘The distributed demonstration test suite’’ later in this guide.

2.2 Test suite processing

A test suite is made up of one or more test cases. Each test case is an executable program.

The test cases in a test suite are processed by the TETware Test Case Controller (`tcc`), according to one or more chosen modes of operation. The available modes of operation are: build mode, execute mode and clean mode. For each test case that is to be processed, `tcc` builds the test case (if build mode has been specified), then executes the test case (if execute mode has been specified), then cleans up the test case (if clean mode has been specified).

The list of test cases that are to be processed by `tcc` is specified in the **test scenario**. The scenario may also contain directives which influence the way in which test cases are to be processed. The way in which the scenario is specified is described in the chapter entitled ‘‘The scenario file’’ later in this guide.

The way in which `tcc` processes a test suite may be influenced by the values of certain **configuration variables**. There is a set of configuration variables for each of `tcc`’s selected modes of operation. These variables are described in the chapter entitled ‘‘Configuration files’’ later in this guide.

As `tcc` processes a test suite it records information about the processing in a **journal**. Information generated by each test case is also recorded in the journal. Further information about the journal is presented in the chapter entitled ‘‘Test reporting and journaling’’ later in this guide.

The operation of `tcc` is described in the chapter entitled ‘‘The Test Case Controller’’ later in this guide.

2.3 Directory structure

TETware expects to operate within a defined directory structure. This structure includes the **tet root directory**, the **test suite root directory**, one or more **test case directories** and (optionally) the **alternate execution directory** hierarchy. Details of the function and purpose of each of these directories is presented in the section entitled ‘‘TETware directory layout’’ and in the appendix entitled ‘‘TETware directory structure’’, both in the TETware User Guide.

All the files in the test suite reside below the **test suite root** directory. The name of this directory is the same as the name of the test suite. The test suite root directory is usually located immediately below the **tet root** directory, although it is possible to locate it elsewhere if so required.

2.4 Test case structure

2.4.1 Introduction

`tcc` is able to execute test cases and tools that either use, or do not use, one of the TETware APIs (but not both types in a particular mode of operation).

A test case which uses one of the TETware APIs is known as an **API-conforming** test case, and a test case which does not use a TETware API is known as a **non API-conforming** test case. Likewise, a build tool or a clean tool can either be an API-conforming tool or a non API-conforming tool, whereas a prebuild tool or a build fail tool is always a non API-conforming tool.[1]

A build tool or a clean tool may be either API-conforming or non API-conforming, depending on the requirements of the test suite. However, **execution** of non API-conforming test cases is supported for compatibility only; new test cases should always use one of the TETware APIs.

2.4.2 Test Case Manager

When a test case uses one of the TETware APIs, its execution is supervised by a Test Case Manager (TCM).

The TCM is not a separate program, but instead is linked with user-supplied test code and the API library to produce an executable test case. There is a separate TCM module for each API that is supported by TETware. Each API is described in a separate chapter in this guide. Instructions for linking a test cases with an API and its TCM are presented in the chapter which describes the API.

The common functionality provided by each TCM is described in the chapter entitled ‘‘The Test Case Manager’’ later in this guide.

2.4.3 API-conforming test cases

An API-conforming test case is constructed by grouping together test functions (called **test purposes**) that test specific system features. These test purposes take advantage of support functions provided by the TETware APIs and are invoked by the TETware TCMs.

When you write a test case which uses a TETware API, you only need to supply the test purpose code that actually performs the required test operation. When a test case is executed, the TETware TCM calls each test purpose function that you write and ensures that each test purpose registers exactly one test result. A test purpose function may call one or more API functions during its execution and, when execution is finished, it returns control to the TCM.

1. The purposes of the various types of tool are described in a later section of this chapter.

This relationship is illustrated through the following picture:

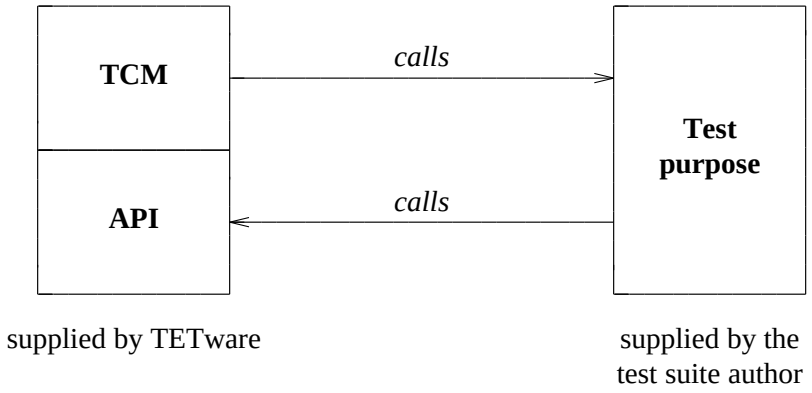

Figure 1. Test case interaction

The picture shows that the TCM calls test purpose functions, and that these functions in turn may call API functions. The API performs functions such as fetching configuration variable settings and writing messages into the journal.

Test purposes within a test case can be grouped together into **invocable components**. This ensures that a set of test purposes is always executed together and in the correct sequence. In most cases there is no practical limit to the number of test purposes that can be grouped in an invocable component and there is no practical limit to the number of invocable components that can be grouped within a single test case. However, there are (substantial) limits for these numbers when an API which supports distributed testing is used. These limits are a byproduct of the synchronisation between parts of a distributed test case which must be performed by the API; details of this synchronisation are presented in the chapter entitled ‘‘Test case synchronisation’’ in the TETware User Guide.

Information about the functionality provided by each API can be found in later chapters of this guide.

2.4.4 Non API-conforming test cases

This section describes how `tcc` processes a non API-conforming test case or tool. The processing described here applies equally to build, exec and clean tools as well as to test cases.

When executing a non API-conforming test case, `tcc` assumes that the test case writes journal output to `stdout` and `stderr` and regards the whole execution as if it were a single invocable component containing a single test purpose. When `tcc` executes the test case, it redirects test case `stdout` and `stderr` to a file that it creates. When processing a test case in execute mode, `tcc` generates the TCM start line that would be emitted by an API-conforming test case, and a result code based on the exit status of the test case. A zero exit status produces a result of `PASS` and any other value produces a result of `FAIL`. When processing of the test case is finished, `tcc` copies the captured output to the journal file.

`tcc` uses the values of certain configuration variables to determine whether it should execute test cases as API-conforming or non API-conforming test cases. The same variables are used by `tcc` to determine how it should execute the build tool and the clean tool. Different values for these

variables can be specified in each of `tcc`'s selected modes of operation if required.

The value of the `TET_API_COMPLIANT` configuration variable specifies whether or not a test case or tool uses the API. The value of the `TET_OUTPUT_CAPTURE` configuration variable specifies whether or not `tcc` should capture test case output and record this output in the journal file.

For convenience, when `TET_API_COMPLIANT` is not defined, it defaults to the inverse of `TET_OUTPUT_CAPTURE`. So, to indicate that you are executing non API-conforming test cases, or are using a non API-conforming build tool or clean tool, you should set `TET_OUTPUT_CAPTURE` to `True` and leave `TET_API_COMPLIANT` undefined. For more information on the use of this these configuration variables, see the section entitled ''Configuration variables which modify TETware's operation'' later in this guide.

2.5 Test suite structure

2.5.1 Introduction

TETware imposes minimum structural requirements on test suites. However, some specific files and utilities must be present. The formats of the data files described in the following sections are described in later chapters in this guide.

2.5.2 Required files and utilities

The following files and utilities must be included with each test suite:

Build tool

This tool is required when a test suite is to be processed in build mode.

The build tool is invoked when `tcc` processes a test case in build mode. In Distributed TETware it is invoked on every system on which the test case is to be processed. The build tool may either be an API-conforming or a non-API conforming tool.

This tool is used to perform the functions that are required to build the test case and, if the test suite makes use of an alternate execution directory, install the test case in its location below that directory. It is common to use `make` for this purpose. Since `make` is a non-API conforming build tool, it is necessary to set the `TET_OUTPUT_CAPTURE` variable to `True` in the build mode configuration.

If a build tool is required to access configuration variables for any reason, it must be an API-conforming tool since non-API conforming tools cannot access configuration variables.

Clean tool

This tool is required when a test suite is to be processed in clean mode.

The clean tool is invoked when `tcc` processes a test case in clean mode. In Distributed TETware it is invoked on every system on which the test case is to be processed. The clean tool may either be an API-conforming or a non-API conforming tool.

This tool is used to perform the functions that are required to clean up after a test case has been built and/or executed. In the trivial case where it is only required to remove the executable file that is created during the build stage, it is common to use `rm` as the clean tool. When this is done, it is necessary to set `TET_CLEAN_FILE` to `-f` and

`TET_OUTPUT_CAPTURE` to `True` in the clean mode configuration.

If a clean tool is required to access configuration variables for any reason, it must be an API-conforming tool since non-API conforming tools cannot access configuration variables.

Configuration variable settings

There is one configuration file for each of `tcc`'s modes of operation. When Distributed TETware is used, there is one of these files on the local system[2] and on each remote system on which test cases are to be processed. Each configuration file contains zero or more configuration variable assignments. Some of these variables affect the way in which `tcc` processes test cases, whereas other variables are meaningful to the test cases being processed. The API provides a mechanism by which variables defined in the configuration file for the current mode of operation may be accessed by test cases and tools.

By default the names of these files are `tetbuild.cfg`, `tetexec.cfg` and `tetclean.cfg`, corresponding to build, execute and clean modes, respectively. These files are located in the test suite root directory on each system. If an alternate execution directory is specified, the execute mode configuration file may (optionally) be located in that directory instead.

The format of configuration files and the meanings of the significant configuration variables are described in the chapter entitled ‘‘Configuration files’’ later in this guide.

Distributed configuration variable settings

When Distributed TETware is used, there is a file on the local system containing variables which specify the locations of test suite files and directories on remote systems. In addition, this file may be used to specify variables that are required by the network transport that is used for interprocess communication. Variables in the distributed configuration file cannot be accessed by test cases and tools.

The name of this file is `tetdist.cfg` and the file is located in the test suite root directory on the local system.

The format of the distributed configuration file is described in the chapter entitled ‘‘Configuration files’’ later in this guide.

This file is not required when TETware-Lite is used.

Test scenario definitions

Each test suite must provide at least one test scenario file. This file contains the definitions of one or more test scenarios. Many scenario files provide a scenario called `all` which typically causes all test cases and invocable components in a test suite to be processed. By default, the name of this file is `tet_scen` and the file is located in the test suite root directory on the local system.

The format of the scenario file is described in the chapter entitled ‘‘The scenario file’’ later in this guide.

---

2. That is: the system on which `tcc` is invoked.

Systems definitions file

In Distributed TETware, this file is used to define the mappings of logical system identifiers to physical machines. The name of the file is `systems` and the file is located in the *tet-root* on each system.

The format of the systems definitions file is described in the section entitled ‘‘System definitions’’ later in this guide.

This file is not required when TETware-Lite is used.

Results directory

Each test suite has a directory called `results` which is located in the test suite root directory. This directory is created by `tcc` if it does not exist. `tcc` creates a unique subdirectory in this directory on each run, into which it places the journal file and a hierarchy of files requested to be saved by the user.

2.5.3 Optional files and utilities

Test suite authors may provide optional files and utilities for use with test suites as follows:

Prebuild tool

When this tool is specified, it is invoked before `tcc` processes a test case in build mode. In Distributed TETware it is invoked only on the master system[3] when a test case is to be processed on more than one system. The prebuild tool should always be a non-API conforming tool.

When files for a remote or distributed test case are maintained only on one system, this tool might be used to propagate these files to the other participating systems before the test case is built.

Build fail tool

When this tool is specified, it is invoked after a prebuild or build tool fails[4] when `tcc` processes a test case in build mode. In Distributed TETware it is invoked on every system on which the test case is to be processed. The build fail tool should always be a non-API conforming tool.

One possible use for this tool might be to provide a skeleton test case that indicates that the real test case was not built successfully by returning a result of `UNINITIATED` for each test purpose that is to be executed.

Exec tool

When this tool is specified, it is is invoked when `tcc` processes a test case in execute mode. In Distributed TETware it is invoked on every system on which the test case is to be processed.

---

3. A test case that is to be processed on more than one system is specified in the scenario file within the scope of a `remote` or `distributed` directive. The first system which appears in the system list which is associated with this directive is known as the **master** system.

4. That is: the tool cannot be executed, the tool is timed out or returns non-zero exit status, or an API-conforming build tool does not report `PASS`.

Possible uses for this tool include:

— setting up environment variables before a test case is invoked

— running a compiled test case under the control of a debugger

— specifying which command interpreter[5] to use

— in Distributed TETware, attaching the test case to a controlling terminal.

Result codes file

TETware utilities which perform result code processing use code definitions which are contained in an internal table. Initially this table contains entries for the result codes which are defined in IEEE Standard 1003.3-1991. Test suite authors may provide files containing additional result codes which are to be added to the table. By default the name of these files is `tet_code` and the files may be located in the *tet-root* and test suite root directories.

The format of the result codes file is described in the section entitled ‘‘Result codes’’ later in this guide.

Treatment filters and report writers

TETware produces a journal file in a well defined format that has been designed so as to enable easy processing by treatment filters and report writers. Test suite authors may wish to provide treatment filters that produce reports in formats which are appropriate for the type of testing that is to be undertaken. The format of the journal file is described in the chapter entitled ‘‘Test reporting and journaling’’ later in this guide.

5. Such as `perl` or one of the shells; useful when the Shell, Korn Shell or Perl APIs are used on Win32 systems and other systems where the `#!` script interpreter convention is not implemented.

3. The Test Case Controller

3.1 Introduction

This chapter describes the operation of the TETware Test Case Controller `tcc`. A manual page for the `tcc` command is presented in the TETware User Guide.

`tcc` accepts user-specific and test suite-specific configuration options and enables the user control of test sessions. This control includes the building, execution, and clean up of test cases. In addition to the control of test sessions, `tcc` includes functionality to support internal mechanisms essential to the operation of TETware. These include managing interaction with the TCM, processing of results and the removal of temporary files.

The TETware-Lite version of `tcc` performs all of these operations itself on a single system. The Distributed version of `tcc` does not perform these operations itself; instead it sends requests to server (or **daemon**) processes which perform the required operations on each system on which test cases are to be processed. Unless stated to the contrary, the information presented in this chapter applies equally to both `tcc` versions.

3.2 Modes of operation

`tcc` processes test cases in one or more of the following modes of operation:

Build mode translates source test cases into executables.

Execute mode loads and executes test cases.

Clean mode removes unwanted files.

These modes of operation are selected by using options on the `tcc` command line.

The way in which `tcc` processes test cases in each mode of operation is affected by the settings of certain variables in the configuration for that mode. Readers should be aware that there is some interaction between the settings of certain variables in each mode. For example, if `TET_PASS_TC_NAME` is not defined, it takes its default value from the value of `TET_OUTPUT_CAPTURE`. Refer to the chapter entitled ‘‘Configuration variables’’ elsewhere in this guide for full details of the meanings of each configuration variable, their default values and the interactions between them.

3.3 Initial processing

Regardless of the mode selected, `tcc` performs the following actions before processing any test cases:

1. `tcc` records the value of the `TET_ROOT` environment variable, and also those of the `TET_SUITE_ROOT`, `TET_EXECUTE`, `TET_TMP_DIR` and `TET_RUN` environment variables if present.
2. `tcc` processes options specified on the command line.
3. `tcc` determines the name and location of the test suite to be processed. The top of the directory subtree in which the test suite resides becomes the **test suite root directory** for the current `tcc` invocation.

4. `tcc` determines the location of the **alternate execution directory** if one has been specified, either by means of the `–a` command-line option or by setting the `TET_EXECUTE` environment variable.
5. `tcc` reads in the configuration variables that are specified for each of the selected modes of operation. In Distributed TETware, this stage reads variables from the configuration files on the local system and establishes the master configurations for each of the selected modes of operation.
6. `tcc` reads in the scenario file (if one has been specified), checks the syntax of all the scenario specifications and identifies the chosen scenario.
7. In Distributed TETware, `tcc` identifies all the system IDs mentioned in the chosen scenario.
8. If a runtime directory has been specified using the `TET_RUN` environment variable, `tcc` copies the test suite root directory hierarchy to a position below the runtime directory. The directory subtree thus created becomes the new test suite root directory. In Distributed TETware this processing in only performed on the local system.
9. In Distributed TETware, when remote systems are mentioned in the chosen scenario or the network transport makes use of distributed configuration variables, `tcc` reads in variables from the distributed configuration file on the local system.
10. `tcc` creates the directory that is to contain the journal file and any saved files. In Distributed TETware this directory is only created on the local system.
11. `tcc` installs signal traps to ensure that an orderly shutdown is performed in the event that an unexpected signal is received.
12. In Distributed TETware, `tcc` starts up the **synchronisation daemon** and **execution results daemon** on the local system, and establishes a connection with the **TCC daemon** on each system mentioned in the chosen scenario.
13. In Distributed TETware, if a runtime directory has been specified using a `TET_REM`*nnn*`_TET_RUN` distributed configuration variable for a particular system, `tcc` copies the test suite root directory hierarchy on that system to a position below the runtime directory. The directory subtree thus created becomes the new test suite root directory for that system.
14. In Distributed TETware, `tcc` creates a saved files directory on each remote system that is mentioned in the chosen scenario.
15. `tcc` reads in any user-supplied result codes files, adds the user-defined results codes to the internal table containing standard results codes and makes the table available to other TETware components that need it. In Distributed TETware, user-supplied result codes files are only provided on the local system; `tcc` propagates the complete results code table to each remote system that is mentioned in the chosen scenario after any user-defined result codes have been added to the table.
16. If rerun or resume mode have been selected, `tcc` processes the old journal file that was produced by the previous `tcc` invocation and modifies the chosen scenario accordingly.

17. If the –y and/or –n command-line options have been specified, tcc prunes the chosen scenario to remove test cases not selected by these options.
18. tcc checks that each timed loop specified in the chosen scenario contains at least one test case to process.
19. If execute mode has been selected and TET_EXEC_IN_PLACE is false, tcc creates the temporary directory below which test case execution will take place. In Distributed TETware the temporary directory is created on each remote system that is mentioned in the chosen scenario.
20. If a journal file has been specified on the command-line, tcc verifies that it does not exist.
21. tcc tells the user the name of the journal file being used and writes a start-up message to the journal.
22. In Distributed TETware, tcc performs a configuration variable exchange for each of the selected modes of operation with each remote system mentioned in the chosen scenario. This stage establishes the per-system configurations for each of the selected modes of operation.
23. tcc reports the configuration variables for each of the selected modes of operation to the journal file. In Distributed TETware, tcc reports the per-system configurations for each of the systems mentioned in the chosen scenario, together with the distributed configuration variables.
24. In Distributed TETware, tcc sends certain communication variables to each system mentioned in the chosen scenario. These variables are put in the environment that is inherited by test cases and tools on that system.

If any of these operations should fail, tcc prints a diagnostic message and exits with non-zero status. When tcc encounters a non-fatal error while it is processing scenario lines or configuration variable assignments it does not exit immediately the first such error is identified. Instead, tcc attempts to perform a reasonable amount of additional processing in order to enable any further non-fatal scenario or configuration errors to be reported as well.

Diagnostic messages which are generated before the journal file has been opened are printed on the standard error stream. A few diagnostic messages which are generated after the journal file has been opened may be printed to the journal file; however, most messages are printed on the standard error stream.

If all of these operations are successful, tcc processes the chosen scenario according to the selected modes of operation. The following sections describe this processing in further detail.

3.4 Build mode processing

When a test suite is provided in source form, tcc is able to build executable files from the source code of each test case. There is no requirement that test suites be provided in source form. Therefore, use of build functionality is optional.

In build mode, tcc builds each test case in the specified scenario. In Distributed TETware it is possible to specify that processing takes place on more than one system at once.

The processing is as follows:

1. `tcc` delivers a Build Start message to the journal.
2. `tcc` obtains exclusive locks in the source and execution directories of the test case. In Distributed TETware these locks are obtained on each participating system.
3. If a `TET_PREBUILD_TOOL` is specified in the build mode configuration, `tcc` executes the prebuild tool in the test case source directory with arguments of `TET_PREBUILD_FILE` and the name of the test case, with output capture mode enabled. If the prebuild tool cannot be executed or returns a non-zero exit status, subsequent actions are not performed and processing resumes with the execution of the build fail tool as described below. In Distributed TETware, when more than one system is specified, the prebuild tool is only executed on the master system (that is: the first system mentioned in the system list).
4. `tcc` executes the build tool in the source directory of the test case with arguments of `TET_BUILD_FILE` and, if `TET_PASS_TC_NAME` is true, the name of the test case. If `TET_OUTPUT_CAPTURE` is true, the build tool is executed with output capture mode enabled. If the build tool cannot be executed, subsequent actions are not performed and processing resumes with the execution of the build fail tool as described below. In Distributed TETware the build tool is executed on each participating system.
5. If output capture mode is enabled, `tcc` transfers captured output to the journal file. If `TET_API_COMPLIANT` is true, `tcc` re-orders and copies the contents of the results file to the journal in the same way as it does when executing an API-conforming test case. In Distributed TETware captured output and the results file contents are gathered from each participating system and entered in the journal on the local system.
6. If the exit status of the build tool is non-zero or `TET_API_COMPLIANT` is true and the build tool did not report a `PASS` result, the build is considered to have failed and, if execute mode has been selected, arrangements are made not to process the test case in execute mode. If the build failed and a `TET_BUILD_FAIL_TOOL` is specified in the build mode configuration, `tcc` executes the build fail tool in the test case source directory with arguments of `TET_BUILD_FAIL_FILE` and the name of the test case, with output capture mode enabled. In Distributed TETware the build fail tool is executed on each system if the build operation failed on any of the participating systems.
7. `tcc` removes the locks obtained in the lock stage. In Distributed TETware locks are removed on each participating system.
8. `tcc` writes a Build End message to the journal.

The following diagram illustrates how `tcc` processes a test case in build mode:

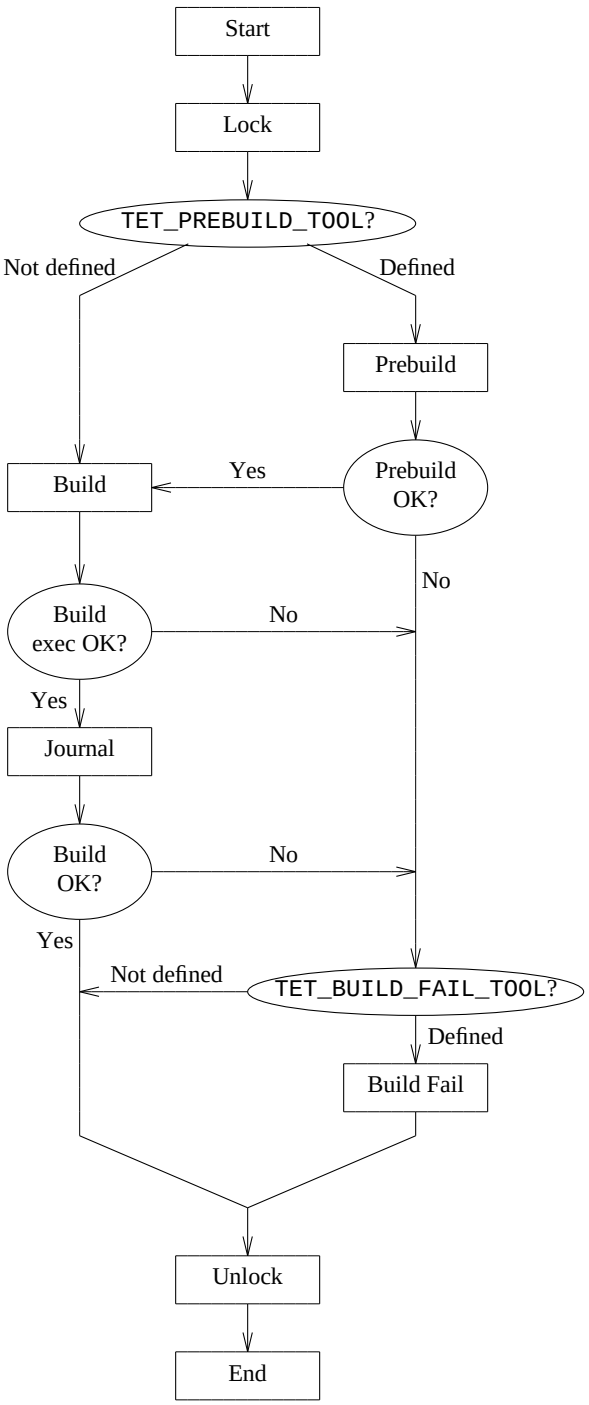

Figure 2. Test case processing in build mode

If the user has specified an alternate execution directory, `tcc` provides that information to the build tool via a communication variable. The test suite author should ensure that the build tool

copies all the files that are required for test case execution to the alternate execution directory.

3.5 Execute mode processing

In execute mode, `tcc` executes each test case in the specified scenario.

In Distributed TETware it is possible to execute instances of a non-distributed test case on more than one system at once, or co-operating parts of a distributed test cases on more than one system at once.

The processing is as follows:

1. `tcc` writes a Test Case Start message to the journal.
2. `tcc` obtains a lock in the test case execution directory. If `TET_EXEC_IN_PLACE` is false, the lock is shared; otherwise, the lock is exclusive. In Distributed TETware locks are obtained on each participating system.
3. If `TET_EXEC_IN_PLACE` is false, `tcc` creates a temporary directory in which execution can safely be performed, and copies the test case execution directory subtree to the location below the temporary directory. In Distributed TETware temporary directories are created and test case files are copied on each participating system.
4. If `TET_EXEC_IN_PLACE` is false, `tcc` removes the lock obtained in the lock stage. In Distributed TETware locks are removed on each participating system.
5. If `TET_API_COMPLIANT` is false, `tcc` writes the TCM Start, IC Start and TP Start messages to the journal that would have been written by an API-conforming test case or tool.
6. If `TET_EXEC_TOOL` is defined, `tcc` executes the exec tool with `TET_EXEC_FILE`, the test case name and the numbers of the invocable components to be executed as arguments; otherwise, `tcc` executes the test case directly with the numbers of the invocable components to execute as arguments. If `TET_EXEC_IN_PLACE` is true, this execution takes place in the test case execution directory; otherwise, execution takes place in the temporary directory. If `TET_OUTPUT_CAPTURE` is true, execution takes place with output capture mode enabled. In Distributed TETware execution takes place on each participating system.
7. If output capture mode is enabled, `tcc` transfers captured output to the journal file. If `TET_API_COMPLIANT` is true, `tcc` re-orders and copies the contents of the results file to the journal. If any test purpose has not generated a result, `tcc` supplies a result of `NORESULT`. If `TET_API_COMPLIANT` is false, `tcc` generates a TP Result line based on the exit status of the test case or exec tool, together with the IC End line that would have been generated by an API-conforming test case or tool.

   In Distributed TETware, captured output and the results file contents are gathered from each participating system and entered in the journal on the local system. When Distributed TETware executes a non-distributed test case on more than one system, results file contents from each system are re-ordered separately and entered in the journal file in turn. When Distributed TETware executes an API-conforming distributed test case, results file contents are not re-ordered; instead, results file contents other than results lines are entered in the journal file in chronological order. A single consolidated result line is generated for each test purpose by arbitrating between the partial result lines gathered from each system and is entered in the journal.

8. `tcc` copies each of the files specified by `TET_SAVE_FILES` to the saved files directory hierarchy. In Distributed TETware, if `TET_TRANSFER_SAVE_FILES` is false, files are copied to the saved files directory hierarchy on each participating system. However, if `TET_TRANSFER_SAVE_FILES` is true, files are copied from each system to a per-system saved files directory hierarchy on the local system. Different values of `TET_TRANSFER_SAVE_FILES` may be specified for each remote system if required.
9. If `TET_EXEC_IN_PLACE` is true, `tcc` removes the lock obtained in the lock stage. In Distributed TETware locks are removed on each participating system.
10. If `TET_EXEC_IN_PLACE` is false, `tcc` removes the temporary execution directory.
11. `tcc` writes a Test Case End message to the journal.

The following diagram illustrates how `tcc` processes a test case in execute mode:

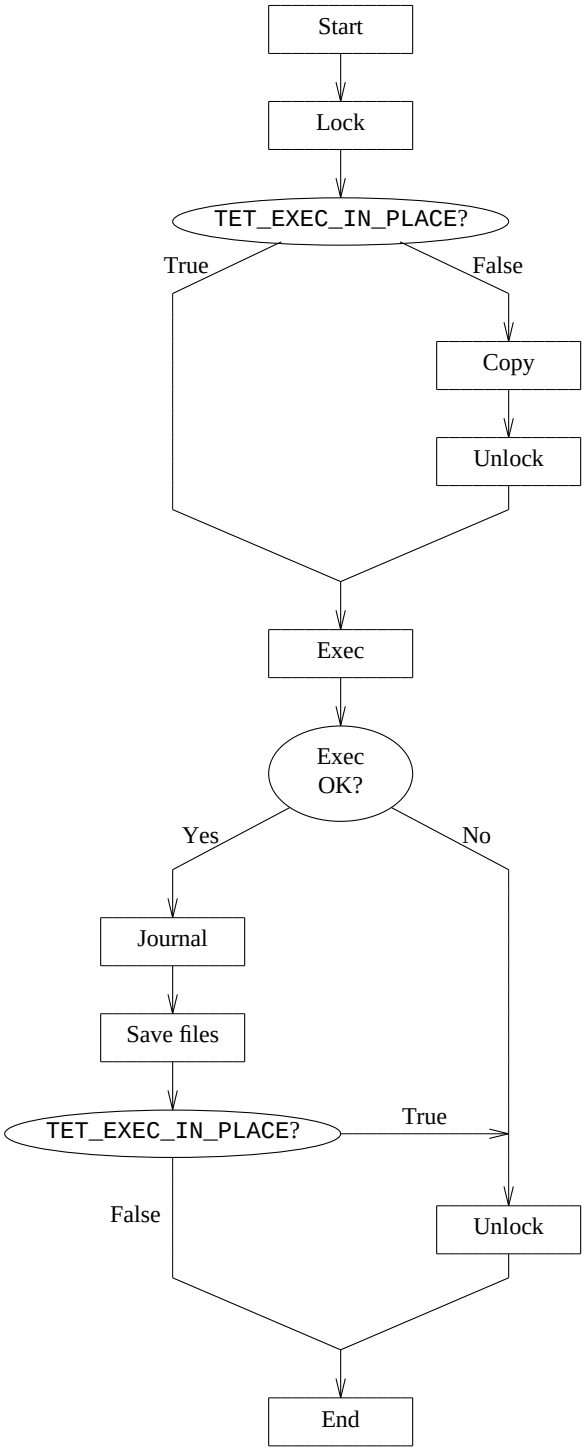

Figure 3. Test case processing in execute mode

3.6 Clean mode processing

Users can request `tcc` to remove unwanted files following test processing sessions. Clean mode processing does not affect the results of previous test runs.

In clean mode, `tcc` cleans up each test case in the chosen scenario. In Distributed TETware it is possible to specify that processing takes place on more than one system at once.

The processing is as follows:

1. `tcc` writes a Clean Start message to the journal.
2. `tcc` obtains exclusive locks in the source and execution directories of the test case. In Distributed TETware these locks are obtained on each participating system.
3. `tcc` executes the clean tool in the source directory of the test case with arguments of `TET_CLEAN_FILE` and, if `TET_PASS_TC_NAME` is true, the name of the test case. If `TET_OUTPUT_CAPTURE` is true, the clean tool is executed with output capture mode enabled. In Distributed TETware the clean tool is executed on each participating system.
4. If output capture mode is enabled, `tcc` transfers captured output to the journal file. If `TET_API_COMPLIANT` is true, `tcc` re-orders and copies the contents of the results file to the journal in the same way as it does when executing an API-conforming test case. In Distributed TETware captured output and the results file contents are gathered from each participating system and entered in the journal on the local system.
5. `tcc` removes the locks obtained in the lock stage. In Distributed TETware locks are removed on each participating system.
6. `tcc` writes a Clean End message to the journal.

The following diagram illustrates how `tcc` processes a test case in clean mode:

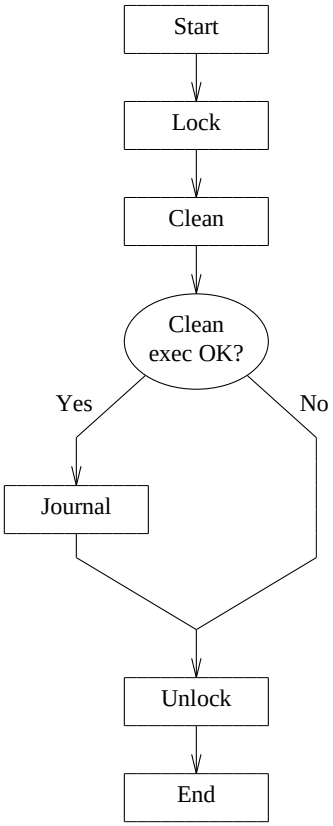

Figure 4. Test case processing in clean mode

3.7 Rerun and resume processing

3.7.1 Introduction

In addition to the normal processing described previously, `tcc` can rerun or resume processing of a previous test run. When you invoke `tcc` with the rerun or resume options, you specify the name of the scenario and journal file from the previous run, and a list of operation modes and/or test purpose result codes which are used to select test cases for reprocessing.

When either of these options are specified, `tcc` uses the list of operation modes specified with the option to select test cases for reprocessing. However, the selected test cases are always reprocessed according to the modes of operation selected for the current test run.

When `tcc` is invoked in rerun or resume mode, it extracts the command-line options used for the previous test run that were recorded in the old journal file. If the `-y` or `-n` options were used to select or reject particular test cases during the previous run, test cases that were not selected are removed from the scenario before the processing described below is performed. Then, after this processing is performed, test cases that are not selected as a result of any `-y` or `-n` options specified for the current test run are removed from the scenario before it is processed.

3.7.2 Resume processing

3.7.2.1 Description

When `tcc` is invoked with the resume option it analyses the old journal file, searching for result codes and/or completion statuses that match one of a user supplied set. The first such result code or completion status that is found identifies the **resume point** in the scenario.

When `tcc` finds the resume point as a result of searching the old journal file, it stores certain parameters which are used to identify the resume point when `tcc` processes the scenario, as follows:

1. The resume point is identified by a particular test case that is to be processed in a particular mode in the chosen scenario.
2. If the scenario is to be resumed at a particular test case in execute mode, the resume point is further identified by a particular IC number within that test case.
3. If the scenario is to be resumed at a particular IC in execute mode and the test case which contains this IC is within the scope of one or more looping directives,[6] the resume point is further identified by the iteration counts of each of the enclosing looping directives.

When `tcc` is invoked in resume mode, it steps through the scenario without processing any test cases until the resume point is found. Once the resume point is found, `tcc` processes the rest of the scenario according to the selected modes of operation in the usual way.

When you invoke `tcc` with the resume option, you specify the search criteria as a (comma-separated) list of result code names and operation mode key letters which select test cases as follows:

List element	Journal lines matched
result-code-name	Test Purpose Result lines giving execution results (not build or clean results) with the equivalent result code number.
`b`	Build End lines which contain a non-zero completion status; and, Test Purpose Result lines giving build results with any non-zero result code number.
`e`	Test Case End lines which contain a non-zero completion status; and, Test Purpose Result lines giving execution results with any non-zero result code number.
`c`	Clean End lines which contain a non-zero completion status; and, Test Purpose Result lines giving clean results with any non-zero result code number.

The names of test cases which occur before the resume point (and are therefore no longer required) are removed from the scenario once the resume point has been identified.

6. These directives are: the `repeat` and `timed_loop` directives.

When you invoke `tcc` with the resume option, you must select the same modes of operation that were selected for the previous test run. It is not possible to resume processing of a scenario using an old journal file that was itself produced by a previous invocation of `tcc` with the resume option.

3.7.2.2 Processing a `parallel` directive in resume mode

If the resume point is found within the scope of a `parallel` directive, the resume point is moved back to the start of the `parallel` directive. When ETET compatibility mode is in effect it is possible for a resume point to be within the scope of several `parallel` directives; in this case the resume point is moved to the start of the outermost enclosing `parallel` directive. It follows, therefore, that if an entire scenario consists of sequences of test cases that are executed in parallel, there is no benefit to be gained by processing the scenario in resume mode since any resume point that is identified is moved back to the start of the scenario.

3.7.2.3 Processing a `random` directive in resume mode

If the resume point is found within the scope of a `random` directive, the resume point is moved back to the start of that `random` directive. However, it should be understood that the test case selection which takes place when `tcc` processes a `random` directive in execute mode is, by definition, random. Thus it is unlikely that `tcc` will make the same selections after the resume point has been found as the selections that were made during the previous test run.

3.7.2.4 Processing a `timed_loop` directive in resume mode

When `tcc` analyses the old journal file in resume mode, it records the number of times that a timed loop starts in execute mode before the resume point is found.

When `tcc` processes a `timed_loop` directive in execute mode, it performs a test before the start of each loop iteration to determine whether or not the loop should be restarted. Ordinarily this test only compares the loop processing time against the time specified with the directive.

However, if this is the only test that is performed before the resume point is found, the possibility exists that a timed loop might iterate a very large number of times before the specified time expires, causing the system to thrash and creating a huge volume of unnecessary journal output. In order to prevent this, the test that is performed before the start of each loop iteration when the resume option is selected and the resume point has not yet been found also checks to ensure that the loop iteration count is less than the count which is derived from the old journal file.

3.7.3 Rerun processing

3.7.3.1 Description

The operation of `tcc` when the rerun option is specified is similar to that of the resume option with the exception that **only** invocable components with result codes matching one of the user supplied set will be processed according to the selected mode of operation.

The names of test cases which are not selected by the rerun option are removed from the scenario once all the test cases which are to be rerun have been identified.

When you invoke `tcc` with the rerun option, you need not select the same modes of operation that were selected for the previous test run. It is possible to rerun a scenario using an old journal file that was itself produced by a previous invocation of `tcc` with the resume or rerun option.

3.7.3.2 Processing a `random` directive in rerun mode

As indicated previously, after `tcc` has identified the test cases that must be reprocessed in rerun mode, it removes all the other test case names from the chosen scenario. This means that only test cases that match the rerun selection criteria remain within the scope of a `random` directive.

Thus, when `tcc` chooses a test case for processing in execute mode from the test cases within the scope of a `random` directive, the choice is made from the set of test cases that match the selection criteria. However, because the choice is made at random, it should be understood that it is unlikely that `tcc` will choose the same test case when processing a particular instance of the `random` directive as was chosen in the previous test run.

3.7.3.3 Processing a `timed_loop` directive in rerun mode

When `tcc` analyses the old journal file in rerun mode, it records the number of times that a timed loop starts in execute mode.

The way that `tcc` processes a `timed_loop` directive in rerun mode is similar to that described above for resume mode. However, the difference is that the test performed before the start of each loop iteration in execute mode always takes account of both the loop execution time and the loop iteration count.

3.8 Communication variables

`tcc` must be able to communicate with the other tools it executes (build tool, clean tool, exec tool, and the test cases). `tcc` does this by using communication variables. Communication variables are environment variables, so environment variables starting with `TET_` are reserved for use by TETware. The communication variables defined include:

`TET_ACTIVITY`
: The number of activities performed thus far by the TCC. Activities include executions of build tool, clean tool, exec tool, and test cases.

`TET_CODE`
: The path name of the current result code definition file.

`TET_CONFIG`
: The path name of the current configuration variable file.

`TET_EXECUTE`
: The path name of the top of the alternate execution directory hierarchy if one has been specified.

`TET_ROOT`
: The path name of the TETware root directory.

`TET_RUN`
: The path name of the runtime directory if one has been specified.

`TET_SUITE_ROOT`
: The path name of the alternate location in which test suite root directories reside, if one has been specified.

3.9 Journal entries

`tcc` manages updates to the journal file. Upon completion of a test case build, execution or clean, the results are transferred from a temporary file into the journal file. During this transfer, `tcc` ensures that each executed test purpose generated one (and only one) result.

The way in which `tcc` processes an execution results file is described in the section entitled ‘‘Result file processing’’ elsewhere in this guide.

3.10 Locking

`tcc` employs a locking system which prevents concurrently executing TCCs from interfering with each other’s processing of test cases. In the following description, ‘‘execution directory’’ means the execution directory under the alternate execution directory hierarchy if one is in use, otherwise the source directory. The discussion applies equally to master and slave systems.

In build and clean modes, `tcc` obtains an exclusive lock in both the source and execution directories as follows:

- A file `tet_lock` is created in the source directory, using an atomic operation which will fail if a file or directory of that name already exists. If the file cannot be created the locking operation fails.
- If an alternate execution directory hierarchy is in use, a file `tet_lock` is created in the execution directory in the same manner. If the file cannot be created, the lock is removed from the source directory and the locking operation fails.

When the build or clean has completed the lock files are removed, execution directory first.

In execute mode with `TET_EXEC_IN_PLACE` true, `tcc` obtains an exclusive lock in the execution directory using the same method as for build mode.

In execute mode with `TET_EXEC_IN_PLACE` false, `tcc` obtains a shared lock in the execution directory as follows:

- A directory `tet_lock` is created in the execution directory, with read and write permission for all users, using an atomic operation which will fail if a file or directory of that name already exists. If the directory cannot be created because a plain file exists, the locking operation fails. If the directory cannot be created because a directory already exists, the failure is ignored.
- A unique file is created in the directory `tet_lock`. If the file cannot be created because `tet_lock` either does not exist or is a plain file, then the locking attempt is re-started.

When the execute has completed the lock is removed as follows:

- The file created when the lock was obtained is removed from the `tet_lock` directory.
- The `tet_lock` directory is removed, using an operation which will fail if the directory is not empty. Failure of this operation is ignored.

3.11 Using `tcc` to process a test suite on a read-only file system

When `tcc` processes a test suite which resides on a read-only file system, it is unable to obtain the locks described in the previous section. However, if an attempt to obtain a lock fails because the file system is read-only, `tcc` ignores the failure. Thus, it is possible to use `tcc` to process a test suite which is provided on a read-only file system. It is possible that such a file system might be mounted read-only from a central file server or might reside on a read-only medium such as a CD ROM.

If the read-only file system contains only source files, `tcc` must first copy the test suite source files to a runtime directory and build the test suite there before it can be executed. The `TET_RUN` environment variable may be used to instruct `tcc` to perform this operation. In Distributed TETware the `TET_REM`*nnn*`_TET_RUN` distributed configuration variables may be used to specify runtime directories on remote systems.

If the read-only file system contains executable test suite files, the `TET_RUN` environment variable may be used in the same way as for a test suite provided in source form. Alternatively, the `TET_TMP_DIR` environment variable may be used to specify a temporary directory location on a writable file store and the test suite can be processed in execute mode with the `TET_EXEC_IN_PLACE` configuration variable set to `False`. In Distributed TETware, the `TET_REM`*nnn*`_TET_TMP_DIR` distributed configuration variables may be used to specify the locations of temporary execution directories on remote systems.

4. The scenario file

4.1 Introduction

When `tcc` processes test cases, it does so by reading instructions contained in a test scenario. Each test suite should include a scenario file which contains one or more test scenarios. This chapter describes the format of the scenario file and the language that is used to specify a test scenario.

When you invoke `tcc`, one of the parameters that you can supply is the name of the scenario to process. If you do not specify this parameter, `tcc` processes a scenario called `all`. Alternatively you can specify a simple test scenario independent of any scenario file by means of one or more `–l` command-line options.

4.2 The scenario language

4.2.1 Introduction

A scenario consists of a sequence of elements. In most cases each element is separated from the next by white space. However, in certain cases, it is possible for two elements to appear together without being separated by white space. When this is done, the second element is said to be **attached** to the first one; the significance of this type of construct is described in a later section.

The first element in the scenario is the **scenario name**. Subsequent elements in the scenario are **directives** and **simple elements**.

A summary of the scenario language is presented in the appendix entitled ‘‘Scenario language syntax summary’’ at the end of this guide.

4.2.2 Scenario lines

Conceptually, a scenario consists of a sequence of elements on a single line. However, in practice it is usually necessary to divide up the elements over several lines in order to limit each line to a manageable length. `tcc` silently imposes a maximum length of 1024 characters (including the newline) on a single physical line read from the scenario file. However, the number of elements that can appear in a scenario and the number of scenarios that may be specified in a scenario file are limited only by the amount of memory that is available to `tcc`.

The start of a scenario is indicated when an element appears at the start of a line. Continuation lines are indicated by placing white space at the start of each line. A comment is introduced by a # character and continues until the end of the line. Blank lines and comments are ignored.

For example, the following scenario:

scenario-name $element_1$ $element_2$ ...

is identical in meaning to:

scenario-name
    $element_1$
    $element_2$
    ...

When continuation lines are used in a scenario, it should be understood that the newline character which ends each line is regarded as part of the white space which separates one scenario element

from the next. Therefore it is not possible to split an individual element over more than one line by using continuation lines.

4.2.3 The scenario name

The first element in the scenario is the scenario name. A scenario name may contain between 1 and 31 characters. Characters in the scenario name are taken from the POSIX portable character set. The first character in the name may be an alphabetic character or a _ character (an underscore). Each of the other characters in the name may be an alphanumeric character or one of the `_-./` characters (an underscore, a hyphen, a period and a forward slash).

4.2.4 Simple scenario elements

4.2.4.1 Introduction

Each simple scenario element is complete in itself and has no effect on other elements in the scenario.

When reading the descriptions that follow, it should be understood that a scenario directive is anything between a pair of `:` characters.

For example:

:*directive*:

In addition, each reference to a **directive** in these descriptions applies equally to a **directive group**. The meanings of the directives themselves and the concept of a directive group are described in later sections in this chapter.

All the simple elements are supported in both TETware-Lite and Distributed TETware. The simple scenario elements are described in the following sections.

4.2.4.2 Scenario information line

A scenario information line is a text string enclosed by a pair of `"` characters (double quotes).

For example:

```
"this is a scenario information line"
```

When `tcc` processes a scenario information line, it simply prints the string (including the double quotes) to the journal.

A scenario information line is treated as a single scenario element; therefore it cannot be split over more than one line by using continuation lines. A scenario information line is the only simple scenario element which may contain embedded spaces.

4.2.4.3 Test case name

A test case name may appear by itself or may be attached to a directive. When a test case name appears by itself, it starts with a `/` character.

For example:

/*test-case-name*

When a test case name is attached to a directive, it starts with a `@/` sequence.[7] There must be no space between the directive's terminating `:` character and the `@` character.

For example:

:*directive*:@/*test-case-name*

A test case name may have an optional list of invocable components (or **IC list**) associated with it. When an IC list is specified, it is enclosed between `{` and `}` characters and attached to the end of the test case name. There must be no space between the test case name and the IC list.

For example:

/*test-case-name*{*ic-list*}

or:

:*directive*:@/*test-case-name*{*ic-list*}

An IC list consists of one or more **numbers** or **number ranges**. Each number or number range is separated from the next by a `,` character (a comma). A number range consists of two numbers separated by a `-` character (a hyphen). A number in the IC list refers to a single invocable component in the test case. A number range refers to a range of invocable components in the test case. An IC list must not contain embedded spaces.

When a test case name appears in a scenario, `tcc` processes the test case according to the selected modes of operation. When `tcc` processes a test case name with an IC list in execute mode, it passes the IC list as an argument to the test case or exec tool. When the TCM receives the IC list argument, it only calls the invocable components that are specified in the list. When no IC list argument is specified, the TCM calls all the invocable components in the list.

For example, if a test case is specified in the scenario as:

/*test-case-name*{2,4,7-10}

`tcc` passes an argument of `2,4,7-10` when it executes the test case. This argument instructs the TCM to call only the user-supplied test purpose functions specified by invocable component numbers 2, 4, 7, 8, 9 and 10. The TCM prints a diagnostic if an invocable component specified in the IC list is not defined in the test case.

A test case name is always interpreted relative to the **test suite root** directory.

4.2.4.4 Referenced scenario name

The name of another scenario (also known as a **referenced scenario name**). A referenced scenario name may appear by itself or may be attached to a directive. In each case the scenario name starts with a `^` character.

---

7. Note that the `@` character is used to distinguish between the attached */test-case-name* described here and the attached */file-name* that is described in a later section.

For example:

^*scenario-name*

or:

:*directive*:^*scenario-name*

When a referenced scenario name appears by itself, `tcc` processes each of the elements contained in the named scenario as if they had appeared where *scenario-name* appears.

When a referenced scenario name is attached to a directive, `tcc` processes each of the elements contained in the named scenario within the scope of the directive to which the *scenario-name* is attached.

4.2.4.5 File name

The name of a file which contains a list of test case names (also known as an **include file** name).

A file name is always attached to a directive and starts with a `/` character.

For example:

:*directive*:/*file-name*

Note that there is no space between the directive's terminating `:` character and the `/` character.

The named file should contain a list of test case names and/or scenario information lines, one per line. Lines in the file should not contain directives or referenced scenario names. Leading white space on a line is permitted but ignored. Comments in the file are introduced with a # character and end at the end of the line. Blank lines in the file and comments are ignored.

An include file is used to associate a list of test case names and/or scenario information lines with a particular directive. When a file name appears in a scenario, `tcc` processes each test case and scenario information line listed in the file within the scope of the directive to which the file name is attached, according to the selected modes of operation. A file name is always interpreted relative to the **test suite root** directory.

4.2.5 Scenario directives

4.2.5.1 Introduction

A directive is a scenario element which has **scope**. It affects the way in which `tcc` processes other elements within its scope.

Each directive is enclosed between a pair of `:` characters, thus:

:*directive*:

A directive may have one or more **parameters** associated with it. Parameters also appear within the pair of `:` characters and are separated from the directive keyword and each other by a `,` character (a comma), thus:

:*directive*,*parameter*...:

A directive may have a simple scenario element **attached** to it. An attached element appears immediately after the `:` character which ends the directive, thus:

:*directive*:*attached-element*

There must be no space between the directive's terminating `:` character and the attached element.

When a directive has a simple element attached to it, the attached element is processed within the scope of the directive. Subsequent elements in the scenario are not processed within the scope of the directive.

When a directive does not have an element attached to it, there must be a matching **end** directive at some point before the end of the scenario. All the scenario elements between the directive and its matching end directive are processed within the scope of the directive. The end directive keyword is formed by prefixing the directive keyword with `end`, thus:

`:end`*directive*`:`

An end directive does not take parameters or have an element attached to it.

Directives may be **nested**; that is: one directive may appear within the scope of another directive. There are rules which determine whether or not a particular directive may appear within another directive's scope. These rules are presented in the section entitled ''Directive nesting rules'' later in this chapter.

Some directives are supported in both TETware-Lite and Distributed TETware, whereas others are supported only in Distributed TETware. The scenario directives are described in the following sections.

4.2.5.2 `repeat` – process scenario elements a specified number of times

Synopsis

```
:repeat[,count]:
    element
    ...
:endrepeat:
```

or:

`:repeat[,`*count*`]:@/`*test-case-name*

or:

`:repeat[,`*count*`]:/`*file-name*

or:

`:repeat[,`*count*`]:^`*scenario-name*

Description

The `repeat` directive is processed by `tcc` as follows:

- If build mode has been selected, `tcc` processes the sequence of elements within the scope of the `repeat` directive once in build mode.
- Then, if execute mode has been selected, `tcc` processes the sequence of elements within the scope of the `repeat` directive *count* times in execute mode.
- Finally, if clean mode has been selected, `tcc` processes the sequence of elements within the scope of the `repeat` directive once in clean mode.

If *count* is specified, it should be a positive number. If *count* is not specified, it defaults to 1.

4.2.5.3 `timed_loop` – process scenario elements until a specified period of time expires

Synopsis

`:timed_loop,`*seconds*`:`
element
...
`:endtimed_loop:`

or:

`:timed_loop,`*seconds*`:@/`*test-case-name*

or:

`:timed_loop,`*seconds*`:/`*file-name*

or:

`:timed_loop,`*seconds*`:^`*scenario-name*

Description

The `timed_loop` directive is processed by `tcc` as follows:

- If build mode has been selected, `tcc` processes the sequence of elements within the scope of the `timed_loop` directive once in build mode.
- Then, if execute mode has been selected, `tcc` performs a test before processing the sequence of elements within the scope of the `timed_loop` directive in execute mode. The sequence of elements is processed repeatedly until the test fails.

  Normally the test performed fails if the time specified by the *seconds* parameter has expired. However, when `tcc` is invoked with the rerun option, or before the resume point is found when `tcc` is invoked with the resume option, the test fails if the time specified by the *seconds* parameter has expired **or** the sequence of elements has already been processed as many times as the same sequence was processed in the course of the test session recorded in the old journal file.
- Finally, if clean mode has been selected, `tcc` processes the sequence of elements within the scope of the `timed_loop` directive once in clean mode.

The *seconds* parameter must be a positive number.

4.2.5.4 `random` – process a test case selected at random

Synopsis

```
:random:
    element
    ...
:endrandom:
```

or:

```
:random:@/test-case-name
```

or:

```
:random:/file-name
```

or:

```
:random:^scenario-name
```

Description

The way in which `tcc` processes the `random` directive depends on which modes of operation have been selected and whether or not this directive appears within the scope of a looping directive,[8] as follows:

- When execute mode has not been selected and the `random` directive is not within the scope of a looping directive:
  - `tcc` processes each of the elements within the scope of the `random` directive in build and/or clean mode according to the selected mode(s) of operation.
- When execute mode has not been selected and the `random` directive is within the scope of a looping directive:
  - If build mode has been selected, `tcc` processes each of the elements within the scope of the `random` directive in build mode.
  - Then, if clean mode has been selected, `tcc` processes each of the elements within the scope of the `random` directive in clean mode.
- When execute mode has been selected and the `random` directive is not within the scope of a looping directive:
  - `tcc` selects a test case at random from within the scope of the `random` directive and builds and/or executes and/or cleans the test case according to the selected mode(s) of operation.
- When execute mode has been selected and the `random` directive is within the scope of at least one looping directive:
  - If build mode has been selected, `tcc` processes each of the elements within the scope of the `random` directive in build mode.

8. The looping directives are: the `repeat` and `timed_loop` directives.

— Then, if execute mode has been selected:

- For each iteration of each enclosing looping directive, `tcc` selects a test case at random from within the scope of the `random` directive and executes it.

— Finally, if clean mode has been selected, `tcc` processes each of the elements within the scope of the `random` directive in clean mode.

It can be seen from this description that when `tcc` processes all the elements within the scope of a `random` directive, both test cases and scenario information lines are processed. However, when `tcc` processes a randomly selected element within the scope of a `random` directive, the selection is made only from test case elements. Therefore, scenario information lines are not processed when elements are selected at random.

When considering the operation of the `random` directive when `tcc` is invoked with the rerun or resume options, it should be understood that the selection of a single test case from within the scope of a `random` directive is, by definition, random. Therefore, when `tcc` is invoked with either of these options, it is likely that a different test case to the one selected in the previous run will be selected in the current `tcc` invocation.

When `tcc` is invoked with the rerun option and must select a test case at random, it selects the test case from the set of test cases within the scope of the `random` directive that are identified by the rerun options and not from the set that appears in the scenario file. Therefore, the **chance** of any particular test case being selected in the current invocation is at least as great as it was in the previous `tcc` run.

Likewise, when `tcc` is invoked with the resume option and identifies the resume point within the scope of a `random` directive, it moves the resume point to the start of the random directive before processing the scenario in the current invocation. Therefore, although the same test case may not be selected from within the scope of the `random` directive after the resume point has been found as was selected in the previous `tcc` run, the **chance** of a particular test case being selected in the current invocation is the same as it was in the previous `tcc` run.

4.2.5.5 `parallel` – process scenario elements in parallel

Synopsis

```
:parallel[,count]:
    element
    ...
:endparallel:
```

or:

```
:parallel[,count]:@/test-case-name
```

or:

```
:parallel[,count]:/file-name
```

or:

```
:parallel[,count]:^scenario-name
```

Compatibility with previous TET implementations

Previous TET implementations have processed the `parallel` directive in different ways.

In dTET2, a `parallel` directive may not enclose a `remote` or another `parallel` directive within its scope, and the `timed_loop` and `random` directives and referenced scenario names are not supported. All the elements within the scope of a `parallel` directive are processed in parallel; that is, processing of each element starts at the same time.[9]

In ETET, a `parallel` directive may enclose other directives and referenced scenario names within its scope. Elements of these types that appear immediately below a `parallel` directive are not truly processed in parallel; instead, for each element below a `parallel` directive the ETET `tcc` forks a child to process the element. Thus, if an element other than a simple scenario element appears below a `parallel` directive, the child processes these subordinate elements **in sequence**. However, all the child processes thus created themselves execute in parallel.

When a directive or referenced scenario name appears within the scope of a `parallel` directive in a context which might be processed differently in previous TET implementations, TETware uses the `TET_COMPAT` configuration variable to resolve the ambiguity in order to provide backwards compatibility with both of these implementations. There is no default value for `TET_COMPAT`. Therefore, if `tcc` needs to refer to this variable when the variable is not defined, it prints a diagnostic and exits.

When `tcc` operates in ETET compatibility mode and needs to process a **sequence** of scenario elements within the scope of a `parallel` directive, it does so by inserting an **implied sequential** directive at the head of the sequence. When `tcc` processes the scenario, it processes the sequences thus defined in parallel; that is: processing of each sequence starts at the same time. However, within each element sequence, processing of elements is sequential; that is: processing of each successive element in the sequence starts as soon as processing of the previous element has finished. This strategy enables TETware to provide ETET compatibility even on operating systems where the `fork()` system call — necessary for ETET's support of the `parallel` directive — is not implemented.

The way in which `tcc` processes elements within the scope of the `parallel` directive is affected by the compatibility mode that is specified by the test suite author using the `TET_COMPAT` configuration variable, as follows:

— When in ETET mode:

- An implied sequential directive is inserted between a `parallel` directive and a subordinate `repeat`, `timed_loop` or `random` directive. Therefore these directives are permitted to appear within the scope of a `parallel` directive.
- If a referenced scenario name appears immediately below a `parallel` directive, the top level of the referenced scenario is searched for `repeat`, `timed_loop` and `random` directives and other referenced scenario names. If one of these elements is found, the referenced scenario name immediately below the `parallel` directive is replaced by a copy of the referenced scenario. Then an implied sequential directive is inserted between the `parallel` directive and each of the subordinate `repeat`,

9. In dTET2, when more than one mode of operation is selected, test cases may be built in parallel, then executed in parallel, then cleaned in parallel, according to the selected modes of operation.

`timed_loop` and `random` directives and other referenced scenario names in the copy of the referenced scenario.

— When in dTET2 mode:

- A `repeat`, `timed_loop` or `random` directive may not appear within the scope of a `parallel` directive.
- Any number of referenced scenario names may appear within the scope of a `parallel` directive, nested to any level, provided that the directive nesting rules are not violated when the contents of each referenced scenario is interpolated.

Description

The `parallel` directive is processed by `tcc` as follows:

- If build mode has been selected, `tcc` processes in build mode a single copy of all the elements (when in dTET2 mode) or sequences of elements (when in ETET mode) within the scope of the `parallel` directive in parallel.
- Then, if execute mode has been selected, `tcc` processes in execute mode *count* copies of all the elements (when in dTET2 mode) or sequences of elements (when in ETET mode) within the scope of the `parallel` directive in parallel.
- Finally, if clean mode has been selected, `tcc` processes in clean mode a single copy of all the elements (when in dTET2 mode) or sequences of elements (when in ETET mode) within the scope of the `parallel` directive in parallel.

If *count* is specified, it should be a positive number. If *count* is not specified, it defaults to 1.

When `tcc` processes a test case, it may obtain locks in the test case source and execution directories in order to prevent unwelcome interference between concurrent test case processing. When `tcc` processes a `parallel` directive, it attempts to obtain all the locks that it needs at the same time. Therefore, it is necessary for the test suite author to organise the test suite in such a way that a locking conflict does not occur when test cases are processed in parallel. Usually this organisation is best achieved by locating each test case in its own directory within the test suite hierarchy.

When `tcc` is invoked with the resume option and identifies the resume point within the scope of a `parallel` directive, the resume point is moved back to the start of the directive. In ETET mode the use of implied sequential directives makes it possible for for a resume point to be found within the scope of more than one `parallel` directives; in this case the resume point is moved back to the start of the outermost enclosing `parallel` directive. A consequence of this is that if an entire scenario is contained within the scope of a `parallel` directive, `tcc` cannot effectively be invoked with the resume option to process such a scenario.

4.2.5.6 `group` – process scenario elements in parallel

Synopsis

```
:group[,count]:
    element
    ...
:endgroup:
```

or:

```
:group[,count]:@/test-case-name
```

or:

```
:group[,count]:/file-name
```

or:

```
:group[,count]:^scenario-name
```

Description

The `group` directive operates in the same way as does the `parallel` directive. This directive is supported only for compatibility with previous TET implementations and should not be used in new test cases.

4.2.5.7 `remote` – process test cases on remote systems

Synopsis

```
:remote,sysid...:
    element
    ...
:endremote:
```

or:

```
:remote,sysid...:@/test-case-name
```

or:

```
:remote,sysid...:/file-name
```

or:

```
:remote,sysid...:^scenario-name
```

Description

The `remote` directive is not supported by TETware-Lite.

In Distributed TETware, `tcc` processes test cases within the scope of the `remote` directive on the systems specified by the *sysid* parameters. At least one *sysid* must be specified. A *sysid* of zero refers to the local system[10] and other positive *sysid* values refer to remote systems.

When the local system is not specified, `tcc` processes test cases within the scope of a `remote` directive as **non-distributed** test cases. When the local system is specified, `tcc` processes test cases within the scope of a `remote` directive as **distributed** test cases. `tcc` supports the

10. That is: the system on which `tcc` is invoked.

processing of distributed test cases when the local system is specified only for backward compatibility with dTET2. Authors of new test suites should use the `distributed` directive to specify distributed test cases.

Distributed test cases must use an API which supports distributed testing; at present these are the C and C++ APIs in Distributed TETware. Non-distributed test cases may use any TETware API or be non API-conforming test cases.

4.2.5.8 `distributed` – process distributed test cases

Synopsis

`:distributed,`*sysid*`...:`
element
...
`:enddistributed:`

or:

`:distributed,`*sysid*`...:@/`*test-case-name*

or:

`:distributed,`*sysid*`...:/`*file-name*

or:

`:distributed,`*sysid*`...:^`*scenario-name*

Description

The `distributed` directive is not supported by TETware-Lite.

In Distributed TETware, `tcc` processes test cases within the scope of the `distributed` directive on the systems specified by the *sysid* parameters. At least one *sysid* must be specified. A *sysid* of zero refers to the local system and other positive *sysid* values refer to remote systems.

`tcc` always processes test cases within the scope of a `remote` directive as **distributed** test cases. Thus it is possible to use this directive to specify a distributed test case which is processed entirely on remote systems.

Distributed test cases must use an API which supports distributed testing; at present these are the C and C++ APIs in Distributed TETware. Test cases which use other TETware APIs and non API-conforming test cases cannot be processed by TETware as distributed test cases.

4.2.5.9 `include` – process scenario elements listed in an include file

Synopsis

`:include:/`*file-name*

Description

The `include` directive is not a true directive in that it does not have scope; that is: it does not affect the way in which `tcc` processes scenario elements in the named file. Instead it is provided simply to enable test suite authors to specify a file containing certain types of simple scenario element to be processed by `tcc` outside the scope of any directives.

Note that the rules that govern the format and contents of the file associated with the `include` directive are the same as those which apply to include files associated with other directives. These rules are presented in the section entitled ‘‘File name’’ earlier in this chapter.

4.2.6 Directive groups

A directive group is constructed from two or more directives that are permitted by the scenario language syntax to appear adjacent to each other in a test scenario.

A directive group is enclosed between a pair of `:` characters, and each directive is separated from the next by a `;` character, thus:

`:`*directive*$_1$`;`*directive*$_2$`...:`

As with individual directives, a directive within a group may have parameters associated with it.

So, the complete formal syntax specification for a directive group which contains one or more directives is as follows:

`:`*directive*`[,`*parameter*`[,...]][;...]:`

As with individual directives, a directive group may have a simple element attached to it, thus:

`:`*directive*$_1$`;`*directive*$_2$`:`*attached-element*

When this is done, the attached element is processed within the scope of all the directives in the group.

When a directive group does not have an element attached to it, there must be matching **end** directives in the correct order at some point before the end of the scenario. Often, each directive in a group without an attached element will be matched by an **end** directive in another group.

For example:

`:`*directive*$_1$`;`*directive*$_2$`:`
element
...
`:end`*directive*$_2$`;end`*directive*$_1$`:`

Note that this example could also be written as follows:

:*directive*$_1$:
:*directive*$_2$:
element
...
:end*directive*$_2$:
:end*directive*$_1$:

or even on a single line, as follows:

:*directive*$_1$;*directive*$_2$: *element* ... :end*directive*$_2$;end*directive*$_1$:

This format is particularly useful when specifying a simple scenario on the command-line by means of the `-l` option to `tcc`.

4.2.7 Directive nesting rules

It is possible for a directive to appear within the scope of another directive. When this is done, the directives are said to be **nested**. However, there are rules which limit the way in which directives may be nested. These rules are defined in terms of whether or not a particular directive may appear within the scope of another directive of the same or a different type.

These rules are complicated by the way in which the `parallel` directive is processed. This processing is described in the section entitled ''`parallel` – process scenario elements in parallel'' earlier in this chapter. In particular, this section describes how `tcc` may insert an **implied sequential** directive in a scenario when processing the scenario in ETET compatibility mode. The placement of these **implied sequential** directives is significant when the directive nesting rules are interpreted by `tcc`.

The directive nesting rules are described in the following table:

Permitted directive combinations						
Inner directive	**Outer directive**					
	`timed_loop`	`repeat`	`random`	`parallel`	`remote` and `distributed`	implied sequential
`timed_loop`	OK	OK	Error	Error	OK	OK
`repeat`	OK	OK	Error	Error	OK	OK
`random`	OK	OK	Error	Error	OK	OK
`parallel`	OK	OK	Error	Error	OK	OK
`remote` and `distributed`	OK	OK	OK	OK	Error	OK
implied sequential	OK	OK	OK	OK	OK	OK

When interpreting these rules it should be understood that the effect of an **implied sequential** directive is to hide a `parallel` directive when directives are nested. That is: for the purposes of these rules the scope of a `parallel` directive is considered to end when an **implied sequential** directive is encountered.

For example, it can be seen from the table above that the the rules do not permit a `repeat` directive to appear within the scope of a `parallel` directive. When dTET2 compatibility mode is in effect, `tcc` does not insert **implied sequential** directives into the scenario. Therefore, the directive nesting rules are violated if a `repeat` directive appears within the scope of a

`parallel` directive.

However, when ETET compatibility mode is in effect and a `repeat` directive appears immediately below a `parallel` directive, `tcc` inserts an **implied sequential** directive between them. The effect of this is to exclude the `repeat` directive from the scope of the `parallel` directive, and so the directive nesting rules are not violated.

4.3 Example scenarios

This section contains some examples of the different ways in which simple elements and directives can be used to define test scenarios. Alternative ways of defining the same scenario are illustrated in some of the more simple examples.

A diagram is used to illustrate each example. Each diagram is presented as a simple flow chart in which time advances from top to bottom.

Example 1

In this scenario the named test cases are simply processed in sequence. One of the test cases has a list of invocable components associated with it.

This scenario can be written in several ways as follows:

```
# simple scenario example
all
        "this is a simple scenario"
        /ts/tc1{1-3,6}
        /ts/tc2
        /ts/tc3
```

or:

```
# simple example using a referenced scenario name
all
        ^scen1

scen1
        "this is a simple scenario"
        /ts/tc1{1-3,6}
        /ts/tc2
        /ts/tc3
```

or:

```
# simple example using an include file
all
        :include:/ts/tclist
```

In this case the file *test-suite-root*`/ts/tclist` contains the following lines:

```
"this is a simple scenario"
/ts/tc1{1-3,6}
/ts/tc2
/ts/tc3
```

The way in which `tcc` processes this scenario may be represented by the following diagram:

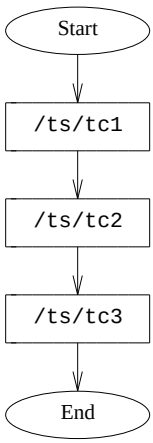

Figure 5. Processing test cases in sequence

Example 2

In this scenario the named test cases are processed in parallel.

This scenario can be written in several ways as follows:

```
# example of parallel processing
all
        "these test cases are processed in parallel"
        :parallel:
        /ts/tc1/tc1
        /ts/tc2/tc2
        /ts/tc3/tc3
        :endparallel:
```

or:

```
# example of parallel processing using an attached element
all
        "the test cases in scenario 'scen1' are processed in parallel"
        :parallel:^scen1

scen1
        /ts/tc1/tc1
        /ts/tc2/tc2
        /ts/tc3/tc3
```

or:

```
# another example of parallel processing using a referenced scenario name
all
        "the test cases in scenario 'scen1' are processed in parallel"
        :parallel:
        ^scen1
        :endparallel:

scen1
        /ts/tc1/tc1
        /ts/tc2/tc2
        /ts/tc3/tc3
```

or:

```
# example of parallel processing using an include file
all
        "the test cases listed in the include file are processed in parallel"
        :parallel:/ts/tclist
```

or:

```
# another example of parallel processing using an include file
all
        "the test cases listed in the include file are processed in parallel"
        :parallel:
        :include:/ts/tclist
        :endparallel:
```

Note that the test suite is organised so that each test case resides in its own directory when a `parallel` directive is used.

The way in which `tcc` processes this scenario may be represented by the following diagram:

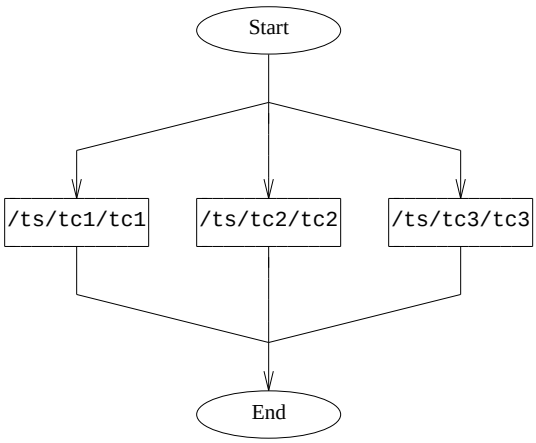

Figure 6. Processing test cases in parallel

Example 3

In this example four instances of a single test case are executed at the same time. This scenario must be processed with `TET_EXEC_IN_PLACE` set to `False` so as to ensure that each test case instance executes in its own directory.

The scenario is defined as follows:

```
all
        :parallel,4:@/ts/tc1
```

When `tcc` processes this scenario in build or clean mode, the test case is processed once. However, when `tcc` processes this scenario in execute mode, four instances of the test case are executed at the same time.

The way in which `tcc` processes this scenario in execute mode may be represented by the following diagram:

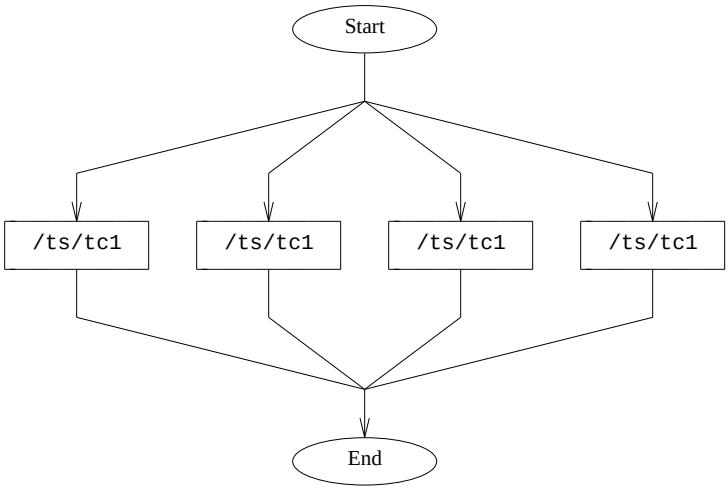

Figure 7. Processing multiple instances of a single test case in parallel

Example 4

This example illustrates how `tcc` processes nested referenced scenario names within the scope of a `parallel` directive differently in dTET2 and ETET compatibility modes.

The scenario is defined as follows:

```
all
        :parallel:^scen1

scen1
        /ts/tc1/tc1
        ^scen2

scen2
        /ts/tc2/tc2
        /ts/tc3/tc3
```

In dTET2 mode all the test cases are processed in parallel. In ETET mode the objects defined at the top level of `scen1` are processed in parallel. However, if an object expands to more than one element, these elements are processed in sequence.

The way in which `tcc` processes this scenario in dTET2 mode may be represented by the following diagram:

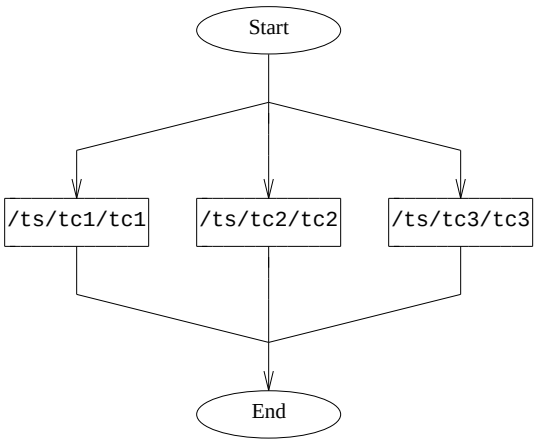

Figure 8. Processing referenced scenario elements in parallel when in dTET2 mode

The way in which `tcc` processes this scenario in ETET mode may be represented by the following diagram:

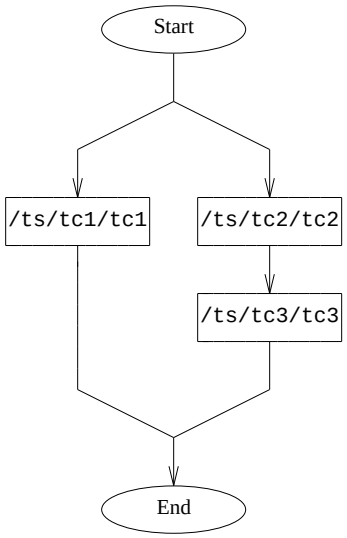

Figure 9. Processing referenced scenario elements in parallel when in ETET mode

Example 5

In this example the named test cases are processed within the scope of a `repeat` directive.

The scenario is defined as follows:

```
all
        :repeat,10:
        /ts/tc1
        /ts/tc2
        /ts/tc3
        :endrepeat:
```

When `tcc` processes this scenario in build or clean mode, the sequence of test cases is processed once. However, when `tcc` processes this scenario in execute mode, the sequence of test cases is executed 10 times.

The way in which `tcc` processes this scenario in execute mode may be represented by the following diagram:

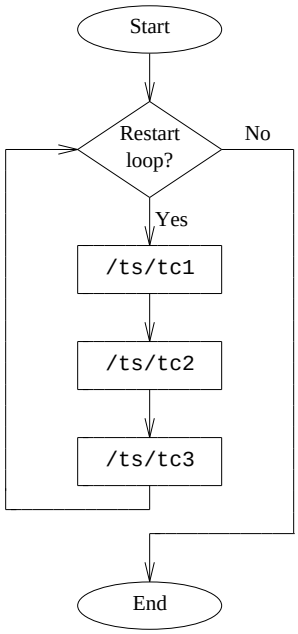

Figure 10. Processing a `repeat` directive in execute mode

Example 6

In this example two instances of the list of test cases within the scope of a `repeat` directive are processed in parallel.

The scenario is defined as follows:

```
all
        :parallel,2;repeat,10:^scen1

scen1
        /ts/tc1
        /ts/tc2
        /ts/tc3
```

Since this scenario contains a looping directive within the scope of a `parallel` directive, the compatibility mode must be specified in order to enable `tcc` to interpret the scenario correctly. When `tcc` reads this scenario in ETET mode, it inserts an **implied sequential** directive between the `parallel` and `repeat` directives. However, no directives are added when `tcc` reads this scenario in dTET2 mode. It can be seen that without the **implied sequential** directive the directive nesting rules have been violated so `tcc` cannot process this scenario in dTET2 mode.

Note that since it is possible for more than one instance of a test case to execute at once, `TET_EXEC_IN_PLACE` must be set to `False` when this scenario is executed.

When `tcc` processes this scenario in build or clean mode, each test case in the list is processed once in sequence. However, when `tcc` processes this scenario in execute mode, two sequences of test cases are initiated at the same time and each sequence is executed 10 times.

The way in which `tcc` processes this scenario in execute mode may be represented by the following diagram:

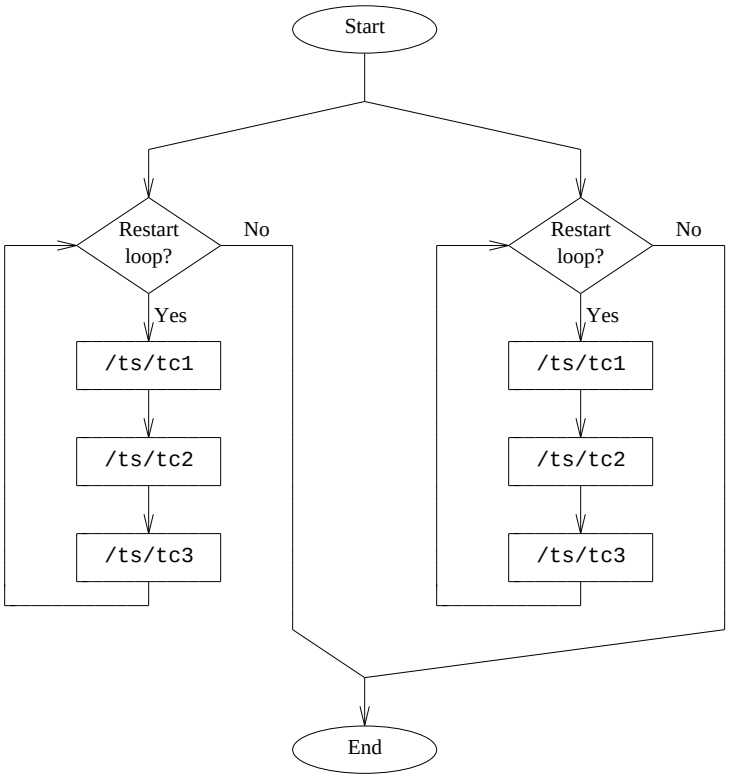

Figure 11. Processing `repeat` directives in parallel

Example 7

In this example two instances of a timed loop execute in parallel. In each loop a single test case is chosen at random from the list of test cases. Each loop is repeated until its execution time has exceeded 300 seconds.

The scenario may be defined as follows:

```
all
        :parallel,2;timed_loop,300;random:^scen1

scen1
        /ts/tc1
        /ts/tc2
        /ts/tc3
```

or as follows:

```
all
        :parallel,2;timed_loop,300;random:
        /ts/tc1
        /ts/tc2
        /ts/tc3
        :endrandom;endtimed_loop;endparallel:
```

or as follows:

```
all
        :parallel,2:
        :timed_loop,300:
        :random:
        /ts/tc1
        /ts/tc2
        /ts/tc3
        :endrandom:
        :endtimed_loop:
        :endparallel:
```

The versions of this scenario shown here illustrate how the same scenario may be written with or without the use of directive groups.

Like the scenario in the previous example, this scenario must be processed in ETET mode and with `TET_EXEC_IN_PLACE` set to `False`.

When `tcc` processes this scenario in build or clean mode, each test case in the list is processed once in sequence. However, when `tcc` processes this scenario in execute mode, two instances of the timed loop are initiated at the same time. Each timed loop instance iterates until 300 seconds have expired. During each iteration of each loop instance, a single test case is selected at random from the list and executed.

The way in which `tcc` processes this scenario in execute mode may be represented by the following diagram:

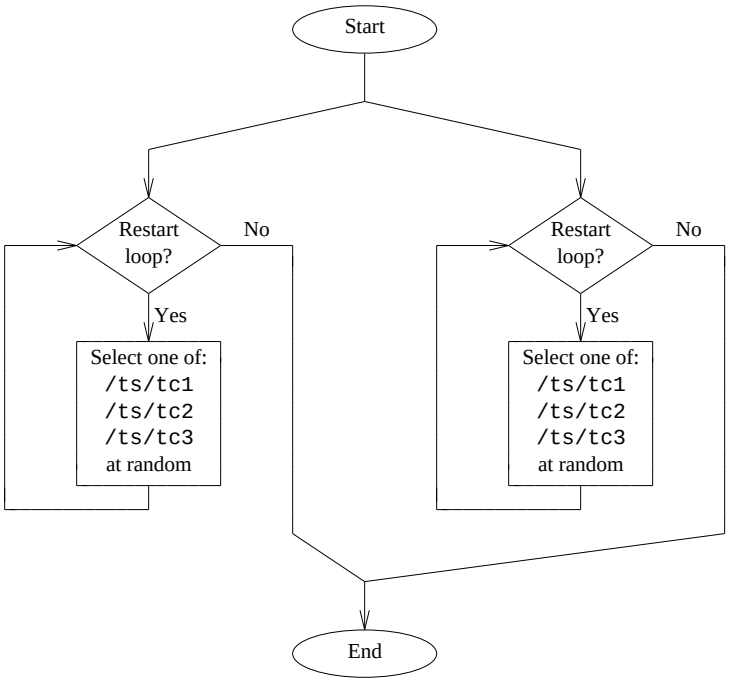

Figure 12. Processing randomly selected test cases in parallel for a specified period of time

Example 8

In this example the named test cases are processed as non-distributed test cases on several remote systems.

The scenario is defined as follows:

```
all
        :remote,001,002:
        /ts/tc1
        /ts/tc2
        /ts/tc3
        :endremote:
```

This scenario cannot be processed by TETware-Lite.

When `tcc` processes each test case in this scenario, it starts the processing of instances of the test case on each system specified by the `remote` directive at the same time. Then, `tcc` waits for the test case instance on each system to finish processing before it starts processing the next test case.

The way in which `tcc` processes this scenario may be represented by the following diagram:

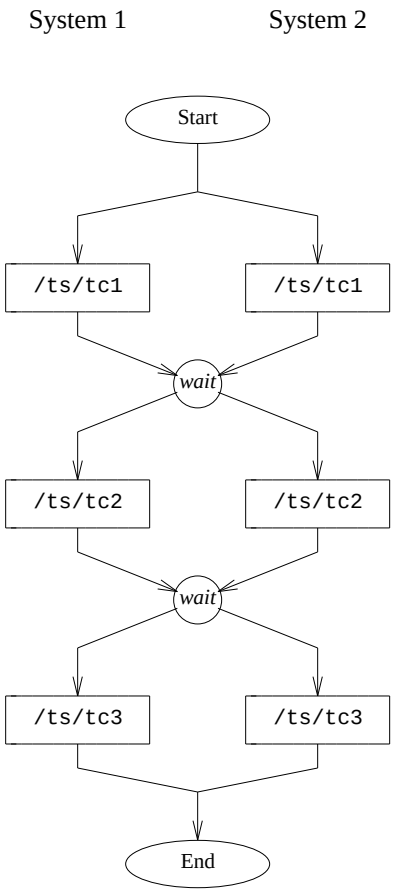

Figure 13. Processing remote and distributed test cases

Example 9

In this example the named test cases are processed as distributed test cases on several remote systems.

The scenario is defined as follows:

```
all
        :distributed,001,002:
        /ts/tc1
        /ts/tc2
        /ts/tc3
        :enddistributed:
```

This scenario cannot be processed by TETware-Lite. The way in which `tcc` processes this scenario may be represented by the same diagram as was used to represent the previous example.

5. Configuration files

5.1 Introduction

Each test suite has one or more configuration files associated with it. These files contain configuration variable assignments which are specified by the test suite author. When `tcc` processes test cases in a particular mode of operation, it reads variables from the configuration file for that mode. So, each test suite should include a build configuration file, an execute mode configuration file and a clean mode configuration file.

By default, configuration files for each test suite are located in the test suite root directory. However, if an alternate execution directory is specified, the execute mode configuration file may be located there instead if so desired. The name of the build mode configuration file is `tetbuild.cfg`, that of the execute mode configuration file is `tetexec.cfg` and that of the clean mode configuration file is `tetclean.cfg`. However, the names of these files may be overridden by `tcc` command-line options if so desired.

In Distributed TETware, configuration files for each mode must be provided on the local system and on each remote system on which tests are to be processed. The names and locations of these files on each remote system are the same as the ones described above for the local system. In addition, a file containing distributed configuration variables must be provided only on the local system when test cases are to be processed on remote systems or when the TETware network code uses such variables. The name of distributed configuration file is `tetdist.cfg` and it is located in the test suite root directory.

5.2 Use of configuration variables

Test suite authors may define variables in the per-mode configuration files which are to be used by API-conforming tools and test cases. (Note that in Distributed TETware, variables defined in the distributed configuration file cannot be accessed by test cases.)

TETware does not provide default values for user-defined variables. Therefore, test suite authors should allow for the possibility that test suite variables may not be defined and ensure that test cases behave sensibly in the event that a required variable is undefined.

In addition to user-defined configuration variables, test suite authors may define certain variables that are used by TETware to determine how test cases are to be processed. These variables are described in the sections that follow.

5.3 Configuration file format

Each (non-blank, non-comment) line in a configuration file specifies a configuration variable assignment in the following format:

variable=*value*

Lines beginning with # and blank lines are ignored.

The first character in a variable's name should be an alphabetic character. Subsequent characters in the name should be an alphanumeric character or a _ character (an underscore). Names beginning with the prefix `TET_` are reserved for use by TETware.

5.4 Configuration variable processing in TETware-Lite

In addition to variables specified in configuration files, configuration variables may be specified on the `tcc` command-line by means of one or more `-v` options. When a configuration variable is specified in this way, it is added to the configuration for each selected mode of operation.

Variables specified using the `-v` command-line option have higher precedence than variables specified in configuration files.

The way in which TETware-Lite processes configuration variables in each mode of operation is illustrated in the following diagram:

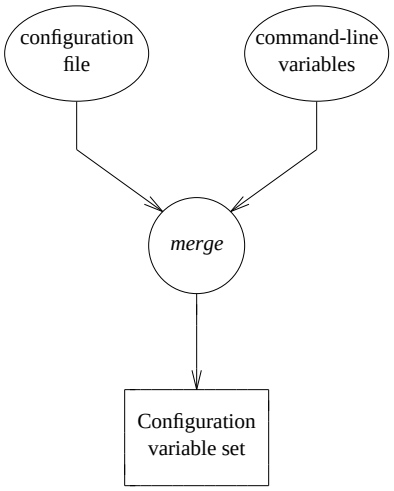

Figure 14. Configuration variable processing in TETware-Lite

5.5 Configuration variable processing in Distributed TETware

As indicated previously, when Distributed TETware is used, configuration variables for each mode of operation may be specified on remote systems as well as on the local system. In addition, it is possible to prefix a variable's name with `TET_REM`*nnn*`_` in order to associate a variable with a particular system.

The Distributed `tcc` processes the configuration for each mode of operation by performing the following actions:

1. `tcc` determines the location of the configuration file on the local system and reads in the variables defined in the file.
2. `tcc` adds in any variables defined on the command-line, giving them precedence over variables defined in the configuration file. The set of variables derived in this way is known as the **master configuration** for the particular mode of operation.
3. If the local system is mentioned in the chosen scenario, `tcc` uses the master configuration to generate a configuration for the local system using the following precedence (highest first):
   — variables with a `TET_REM000_` prefix defined on the `tcc` command line

— other variables defined on the `tcc` command line

— variables with a `TET_REM000_` prefix defined in the local configuration file

— other variables defined in the local configuration file

Then `tcc` removes any `TET_REM000_` from each variable in the local system's configuration. The set of variables derived in this way is known as the **per-system configuration** for the local system.

4. For each remote system that is mentioned in the chosen scenario, `tcc` performs a **configuration variable exchange** with `tccd` on that system, using the master configuration. When performing this operation, `tcc` indicates which variables in the master configuration originated from the command-line.
5. `tccd` reads in the variables defined in the configuration file on that system.
6. Then `tccd` merges these variables with the master configuration received from the local system using the following precedence (highest first). In the text that follows, ''local'' and ''remote'' describe systems from `tcc`'s point of view and ''a matching `TET_REM`*nnn*`_` prefix'' is a prefix in which *nnn* matches the system ID of the remote system.

   — variables with a matching `TET_REM`*nnn*`_` prefix defined on the `tcc` command line

   — other variables defined on the `tcc` command line

   — variables with a matching `TET_REM`*nnn*`_` prefix defined in the configuration file on the remote system

   — variables with a matching `TET_REM`*nnn*`_` prefix defined in the master configuration received from `tcc`

   — other variables defined in the configuration file on the remote system

   — other variables defined in the master configuration received from `tcc`
7. Finally, `tccd` removes any matching `TET_REM`*nnn*`_` prefix from each variable and returns the merged configuration back to `tcc.` The set of variables derived in this way is known as the **per-system configuration** for that system.

From this description it will be seen that it is possible to define a variable on the local system that is to appear in the master configuration and in the per-system configurations for both the local and remote systems. Such variables may be defined in a configuration file on the local system or on the `tcc` command line. In addition, it is possible to define a variable in a configuration file on a remote system that is to appear in the per-system configuration for that system.

However, it is not possible to define a variable in a configuration file on a remote system that is to appear in the master configuration or in the per-system configuration for another system.

The way in which Distributed TETware processes configuration variables in each mode of operation is illustrated in the following diagram:

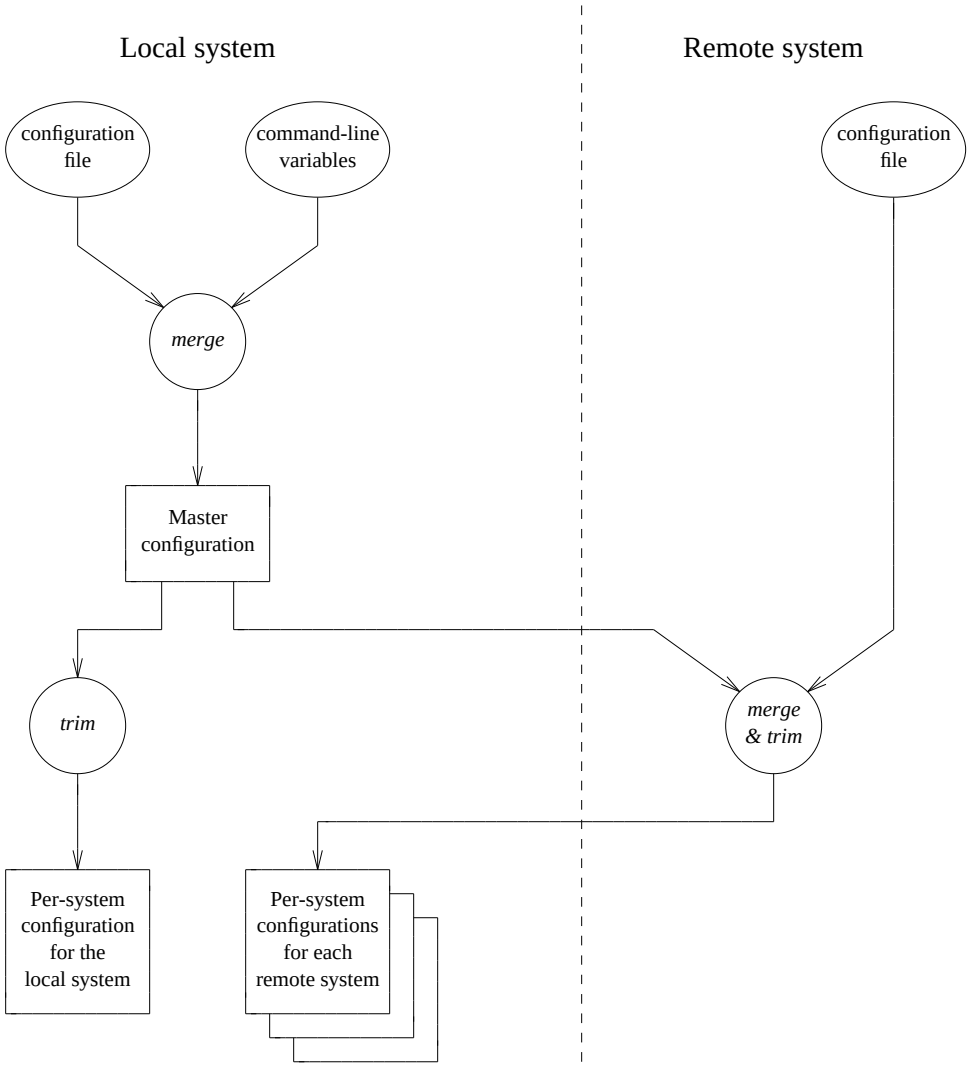

Figure 15. Configuration variable processing in Distributed TETware

The result of all this is that when Distributed TETware is used you can specify variables that are read from each of the per-system configurations in several ways.

Configuration variables may be specified both on the local system and on any remote systems that are to participate in remote or distributed testing. In this context, the **local** system is the system on which `tcc` is run (whether or not any test cases run on this system), and **remote** systems are other systems on which test cases or test case parts are run. When reading the discussion that follows, you should bear in mind that the local system always has a system ID of zero; other system IDs always refer to remote systems.

Configuration variable assignments made on the local system are propagated to each of the remote systems; however, configuration variable assignments made on a remote system normally have precedence over those that are propagated from the local system.

For example, if the following assignment is made on the local system:

```
TET_BUILD_TOOL=make
```

then, the value of TET_BUILD_TOOL will be set to make on the local system and on all the remote systems.

If the following assignment is made on one of the remote systems:

```
TET_BUILD_TOOL=augmake
```

then the value of TET_BUILD_TOOL is changed to augmake only on that remote system, and remains unchanged on all of the other systems.

It is possible to direct a variable assignment made on the local system to a particular system by prefixing its name with TET_REM*nnn*_ where *nnn* is the ID of the system that is to receive the variable.

So, if the following assignments are made on the local system:

```
TET_BUILD_TOOL=make
TET_REM002_TET_BUILD_TOOL=augmake
```

then the value of TET_BUILD_TOOL on remote system 002 is set to augmake and the value of TET_BUILD_TOOL on the local system and all the other remote systems is set to make.

Furthermore, the value of a TET_REM*nnn*_ variable assignment made on the local system overrides any assignment to the corresponding variable that may be made on system *nnn*. So, in this case, the value of TET_BUILD_TOOL on remote system 002 is set to augmake irrespective of any assignment that might be made on that remote system.

Finally, if the following assignments are made on the local system:

```
TET_BUILD_TOOL=augmake
TET_REM000_TET_BUILD_TOOL=make
```

then the value of TET_BUILD_TOOL on the local system will be set to make and the value of TET_BUILD_TOOL on all the remote systems will be set to augmake (provided that no assignment for TET_BUILD_TOOL is made on any of the remote systems).

5.6 Configuration variables which modify TETware's operation

This section describes configuration variables which affect the way in which TETware processes a test suite. The variables described here should be set in the per-mode configurations.

In Distributed TETware, the values of some variables are read from the master configuration and affect the way in which TETware processes test cases on all systems. By contrast, the values of other variables are read from each per-system configuration and affect the way in which TETware processes test cases on each individual system.

Some variables used by TETware are boolean variables, whereas others are string variables. TETware provides default values are provided for all the boolean variables and some of the string variables.

The following table lists all the variables used by TETware, the type of each value and the default value supplied (if any). The last column indicates whether Distributed TETware obtains the variable's value from the master or the per-system configuration.

Configuration variables used by TETware			
Variable name	**Type**	**Default value**	**Source in Distributed TETware**
TET_API_COMPLIANT	boolean	inverse of TET_OUTPUT_CAPTURE	master
TET_BUILD_FAIL_FILE	string	undefined	per-system
TET_BUILD_FAIL_TOOL	string	undefined	per-system
TET_BUILD_FILE	string	undefined	per-system
TET_BUILD_TOOL	string	undefined	per-system
TET_CLEAN_FILE	string	undefined	per-system
TET_CLEAN_TOOL	string	undefined	per-system
TET_COMPAT	string	undefined	master
TET_EXEC_FILE	string	undefined	per-system
TET_EXEC_IN_PLACE	boolean	False	master
TET_EXEC_TOOL	string	undefined	per-system
TET_OUTPUT_CAPTURE	boolean	False	master
TET_PASS_TC_NAME	boolean	same as TET_OUTPUT_CAPTURE	master
TET_PREBUILD_FILE	string	undefined	per-system
TET_PREBUILD_TOOL	string	undefined	per-system
TET_RESCODES_FILE	string	tet_code	master
TET_SAVE_FILES	string	undefined	per-system
TET_TRANSFER_SAVE_FILES	boolean	False	per-system

The meaning of each variable is as follows:

TET_API_COMPLIANT — Specifies whether or not test cases and tools use a TETware API. If true, test cases and tools are expected to use the API to print diagnostics and register results. If false, tcc treats the test case or tool as if it consists of a single invocable component containing a single test purpose. When tcc processes the test case in execute mode, it prints the messages to the journal file that would be printed by an API-conforming test case and generates a test purpose result based on the test case's exit status (zero = PASS, non-zero = FAIL).

TET_BUILD_FAIL_FILE — Names the file of instructions for the build fail tool. The use of this variable is optional.

TET_BUILD_FAIL_TOOL — Names the utility to be executed if a prebuild or build operation fails. The use of this variable is optional.

TET_BUILD_FILE — Names the file of instructions for the build tool. The use of this variable is optional.

TET_BUILD_TOOL — Names the utility to be executed when processing a test case in build mode. This variable must be specified if build mode is selected.

TET_CLEAN_FILE — Names the file of instructions for the clean tool. The use of this variable is optional.

TET_CLEAN_TOOL	Names the utility to be executed when processing a test case in clean mode. This variable must be specified if clean mode is selected.
TET_COMPAT	Specifies the compatibility mode to be used when interpreting a scenario. Possible values are: `dtet2` to select dTET2 compatibility mode or `etet` to select ETET compatibility mode. This variable must be specified if the scenario contains ambiguous syntax.
TET_EXEC_FILE	Names the file of instructions for the exec tool. The use of this variable is optional.
TET_EXEC_IN_PLACE	Specifies whether or not `tcc` should execute test cases ‘‘in place’’. If false, `tcc` copies test case files to a temporary directory before executing them. The setting of this variable is only meaningful in execute mode.
TET_EXEC_TOOL	Names the utility to be executed when processing a test case in exec mode. Normally this variable is not specified, in which case the test case is executed directly.
TET_OUTPUT_CAPTURE	Specifies whether or not `tcc` should capture standard output and standard error output from test cases and record it in the journal. For historical reasons the value of this variable also provides default values for the `TET_API_COMPLIANT` and `TET_PASS_TC_NAME` configuration variables.
TET_PASS_TC_NAME	If true, `tcc` passes the name of the test case to be processed on the command-line when executing a build or clean tool. If false, `tcc` does not pass a test case name to a build or clean tool. Note that `tcc` always passes a test case name to a prebuild, buildfail or exec tool.
TET_PREBUILD_FILE	Names the file of instructions for the prebuild tool. The use of this variable is optional.
TET_PREBUILD_TOOL	Names the utility to be executed before processing a test case in build mode. In Distributed TETware, if the test case to be processed in within the scope of a `remote` or `distributed` directive which specifies more than one system, the prebuild tool is only executed on the first system in the list. The use of this variable is optional.
TET_RESCODES_FILE	This variable specifies the name of the result code file. When more than one mode of operation is selected and this variable is defined in more than one per-mode configuration, only the first definition is significant. Thus the use of this variable to specify a results code file is per `tcc` invocation and not per mode of operation. The use of this variable is optional.
TET_SAVE_FILES	This variable specifies a (comma separated) list of file names. If, after `tcc` executes a test case, a file matching one of these names is found below the execution directory hierarchy, that file is transferred to the saved file directory tree on the same

system. If a directory is found that matches one of the names, then its contents are transferred recursively. Shell file name matching syntax may be used in the list of file names. The use of this variable is optional.

`TET_TRANSFER_SAVE_FILES` If true, files processed by Distributed TETware on a remote system in accordance with the description of `TET_SAVE_FILES` above are transferred to the saved file directory on the local system instead of being saved on that remote system. The use of this variable is optional. This variable is not used in TETware-Lite.

5.7 Distributed configuration variables used by Distributed TETware

This section describes distributed configuration variables. When Distributed TETware processes test cases on remote systems, these variables inform `tcc` of the locations of test case files and directories on each remote system. In addition, when Distributed TETware is built to use the XTI network interface, certain distributed configuration variables are used by `tcc`'s network code. The variables described here should be set in the distributed configuration file on the local system.

The following table lists all of the distributed configuration variables used by Distributed TETware. Separate values of each variable with a `TET_REM`*nnn*`_` prefix must be supplied for each remote system mentioned in the chosen scenario.

Distributed configuration variables used by Distributed TETware

Variable name	Type	Default value
`TET_REM`*nnn*`_TET_EXECUTE`	string	undefined
`TET_REM`*nnn*`_TET_ROOT`	string	undefined
`TET_REM`*nnn*`_TET_RUN`	string	undefined
`TET_REM`*nnn*`_TET_SUITE_ROOT`	string	same as the value of `TET_REM`*nnn*`_TET_ROOT`
`TET_REM`*nnn*`_TET_TMP_DIR`	string	`tet_tmp_dir`
`TET_REM`*nnn*`_TET_TSROOT`	string	undefined
`TET_LOCALHOST`	string	undefined
`TET_XTI_MODE`	string	`tcp`
`TET_XTI_TPI`	string	`/dev/tcp`

The meaning of each variable is as follows:

`TET_REM`*nnn*`_TET_EXECUTE` The values of these variables specify the locations of alternate execution directories on remote systems. The use of these variables is optional but, if they appear, they perform the equivalent functions on remote systems to that performed by the value of the `TET_EXECUTE` environment variable on the local system. The values of these variables are passed to test cases and tools in the environment as communication variables on each system.

`TET_REM`*nnn*`_TET_ROOT` The values of these variables specify the locations of **tet root** directories on remote systems. One of these variable assignments must be made for each remote system that may participate in remote or distributed testing. The values of these variables are passed to test cases and tools in the environment as communication variables on each system.

`TET_REM`*nnn*`_TET_RUN` The values of these variables specify the locations of runtime directories on remote systems. The use of these variables is optional but, if they appear, they perform the equivalent functions on remote systems to that performed by the value of the `TET_RUN` environment variable on the local system (refer to the section entitled ‘‘Environment variables’’ earlier in this chapter). The values of these variables are passed to test cases and tools in the environment as communication variables on each system.

`TET_REM`*nnn*`_TET_SUITE_ROOT`

These variables are not used by TETware but, when specified, are passed to test cases and tools in the environment as communication variables on each system. This is done in order to enable existing ETET test cases which rely on the presence of a `TET_SUITE_ROOT` environment variable to be processed on a remote system by Distributed TETware.

`TET_REM`*nnn*`_TET_TMP_DIR` The values of these variables specify the locations of temporary directories on remote systems which are used instead of the default location when `TET_EXEC_IN_PLACE` is false. The use of these variables is optional but, if they appear, they perform the equivalent functions on remote systems to that performed by the value of the `TET_TMP_DIR` environment variable on the local system.

`TET_REM`*nnn*`_TET_TSROOT` The values of these variables specify the locations of test suite root directories on remote systems. One of these variable assignments must be made for each remote system that may participate in remote or distributed testing.

In addition, the following distributed configuration variables are accessed by `tcc`’s network transport code when the XTI network interface is used:

`TET_LOCALHOST` This variable must be specified when the XTI network interface is used and the underlying transport provider is TCP/IP. The value of this variable should be the Internet address of the local system. This address is presented in dot notation and must be an address that can be used to access the local system from remote systems (i.e., it should not be the address of the loopback interface). All four fields in the address must be specified.

`TET_XTI_MODE`	Possible values: `tcp` (to indicate TCP/IP) or `osico` (to indicate the OSI connection-oriented transport). The value of this variable indicates the underlying transport provider to be used.
`TET_XTI_TPI`	The name of the XTI transport provider identifier on the local system.

6. Other test suite files

6.1 Introduction

This chapter describes the formats of other files that may be provided with each test suite.

6.2 Result codes

6.2.1 Description

A mapping mechanism is provided to enable TETware processes to interpret results from test suites. This mapping is contained in a result codes file. When a test purpose returns a particular result, the TCM determines the action required for each result code and writes an entry in the journal. The API library ensures that the test purposes only generate standard or test suite specified results.

TETware provides the default set of result codes that are defined in IEEE Std 1003.3-1991. Additional result codes may be defined on a per installation or per test suite basis.

6.2.2 Result code definitions

TETware provides the standard result code definitions. The user can supply additional result code definitions. The standard codes are defined in an internal table provided by TETware. It is an error for the user to assign different meanings to the standard codes.

The provision of user-supplied result code files is optional. User-supplied codes for use by all test suites may be defined in a file which is located in the **tet root** directory. User-supplied codes for use by a particular test suite may be defined in a file which is located in that test suite's **test suite root** directory. When Distributed TETware is used these file reside on the local system.[11]

By default, the names of each user-supplied file is `tet_code`. However, a different name may be defined by use of the `TET_RESCODES_FILE` configuration variable. `tcc` determines the name of each user-supplied result code file using the following algorithm:

- If build mode has been selected and `TET_RESCODES_FILE` is defined in the build configuration file, then that value is used.
- If no file name has yet been determined and execute mode has been selected and `TET_RESCODES_FILE` is defined in the execute configuration file, then that value is used.
- If no file name has yet been determined and clean mode has been selected and `TET_RESCODES_FILE` is defined in the clean configuration file, then that value is used.
- If no file name has yet been determined then `tet_code` is used.

11. That is: the system on which `tcc` is invoked.

When result code files are supplied, the precedence of result definitions is as follows (highest precedence first):

- Codes defined in the file at the **test suite root** level.
- Codes defined in the file at the **tet root** level.
- Codes defined in the internal table provided by TETware.

This precedence is illustrated in the following diagram:

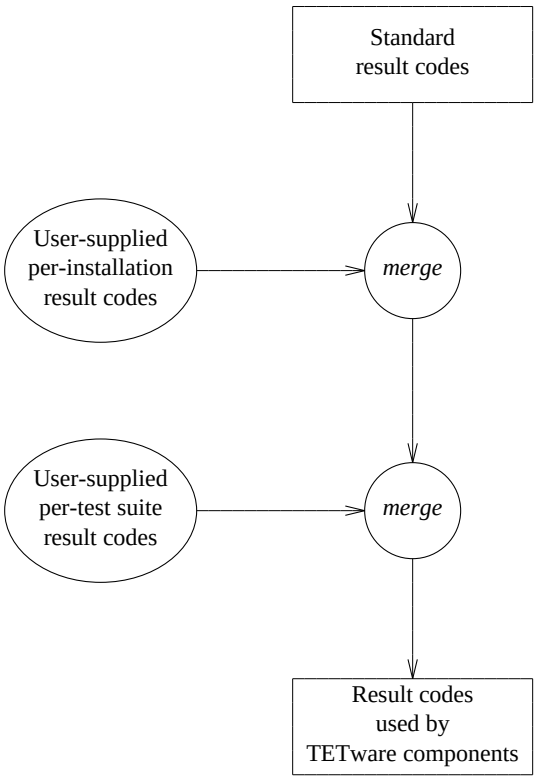

Figure 16. Precedence of result code definitions

6.2.3 File format

Blank lines and lines starting with a # are ignored. Other lines in this file contain up to three blank separated fields, defined as follows:

1. The result code. This is a non-negative decimal integer between 0 and 127, inclusive. Result codes from 0 to 31 (inclusive) are reserved for use by TETware. The remainder are available for use by the test suite author.
2. The name of this result. This is a field delimited by double quotes which contains a text string describing the result. This field may contain embedded spaces.
3. The action to take when this result is encountered. This is an indication of what the TCM should do when the result is returned by a test purpose. Possible values are `Continue`

and `Abort`. The default value for this field is `Continue`.

6.2.4 Example results code file

The following is an example result codes definition file. It contains some user-defined result codes as well as the standard result codes provided by TETware.

```
# first, the standard result codes
0   "PASS"                  Continue
1   "FAIL"                  Continue
2   "UNRESOLVED"            Continue
3   "NOTINUSE"              Continue
4   "UNSUPPORTED"           Continue
5   "UNTESTED"              Continue
6   "UNINITIATED"           Continue
7   "NORESULT"              Continue
# then, some codes for use with this test suite
32  "INSPECT"               Continue
33  "STOP RUN"              Abort
```

6.3 System definitions

6.3.1 Description

Distributed TETware uses a systems definition file to define the mapping of a TETware system ID to a host name or other parameter which may be used by the network code to establish a connection with that system. The systems definition file is not used by TETware-Lite.

The name of this file is `systems` and it is located in the *tet-root* directory on each system.

When remote or distributed testing is to be performed, TETware components on each participating system each refer to the `systems` file on that system when mapping a TETware system ID to a network address. You must ensure that the same mappings are defined on all participating systems, otherwise unpredictable behaviour will occur.

In addition, test cases can access entries in the system definition file by calling the `tet_getsysbyid()` API function.

6.3.2 File format

Blank lines and lines starting with a # are ignored. Other lines in this file contain up to three blank separated fields. The first field contains the TETware system identifier. System zero must always refer to the system on which `tcc` is to be invoked (the **master** or **local** system). Other (**remote** or **slave**) systems are specified by system identifiers with positive values. The value of the system identifier for a remote system must be in the range 1 through 999.

When Distributed TETware is built to use the socket network interface, each entry in the `systems` file takes the following form:

sysid host

The system is identified by the value in the *host* field. The host name lookup functions on each system must be able to perform address resolution on each host name listed in the `systems` file. Note that it is usually an error to specify a host name as `localhost` since that name cannot be used to connect to another system.

When Distributed TETware is built to use the X/Open Transport Interface (XTI), each entry in the `systems` file takes the following form:

sysid host address-string

The *host* field is not used when Distributed TETware is built to use XTI.[12]

The *address-string* field contains a hexadecimal string representation of a data item which is used to identify the address of a network endpoint. The network endpoint thus identified must refer to the entity which is used by the Test Case Controller daemon (`tccd`) on the named system to accept connections from client processes. Each byte in the data item is represented by a pair of hexadecimal digits in the address string; for example, a byte in the data item with decimal value 13 is specified in the address string as `0d`. The precise format of the data item depends on which transport provider is being used.

6.3.3 Example `systems` files

Here are some example `systems` files. Note that, in each example, one of the machines is referenced by more than one logical system ID.

The following example is for a machine on which TETware has been built to use the socket network interface:

```
# Example system file for INET implementation on host 'ozone'

000     ozone
001     neon
002     argon
003     ozone
```

Entries for all the host names mentioned in this file should appear in the hosts database on each system.

The following example is for a machine on which TETware has been built to use the XTI network interface:

```
# Example system file for XTI implementation on host 'ozone'

000     ozone           000204010a010200000000000000000000
001     neon            000204010a010300000000000000000000
002     argon           000204010a010400000000000000000000
003     ozone           000204010a010200000000000000000000
```

The contents of the XTI address string depends on the transport being used, the network implementation and the architecture of the machine on which the file resides. Therefore, although the XTI address strings specified for a particular system in the `systems` files on each machine must describe the same transport address, the contents of this field for a particular entry may be different on different types of machine.

12. However, test cases may still access the value in this field by calling the `tet_getsysbyid()` API function.

7. The Test Case Manager

7.1 Introduction

TETware supplies functionality to support the development of test cases. This chapter describes that functionality for a Test Case Manager (TCM) or ‘‘wrapper’’ that provides a suitable environment for the execution of invocable components as requested by the Test Case Controller (`tcc`).

The TCM is not a separate program but instead is part of each TETware API. Whichever language binding is used, the appropriate version of the TCM and API are linked with the user-supplied test code to produce each test case.

TETware-Lite and Distributed TETware provide different versions of the C and C++ TCMs. However, the same versions of the Shell, Korn Shell and Perl TCMs are supplied with both Distributed TETware and TETware-Lite. This is because in Distributed TETware the C and C++ TCMs support distributed test cases whereas the others do not.

Through the TCM, developers gain support in doing the following:

- Initialising and cleaning-up test cases.
- Selecting invocable components and test purposes.
- Insulating from the test environment.
- Making journal entries.

7.2 TCM flow of control

This section describes the flow of control in a TETware TCM. When Distributed TETware is used to execute parts of a distributed test case, this processing takes place on each participating system except where noted otherwise. When TETware-Lite or TCMs that do not support distributed testing are used, processing that is described as being associated with distributed test cases is not performed.

The general flow of control for test cases written to one of the TETware APIs is as follows:

1. The TCM arranges for access to its respective configuration information.
2. The TCM outputs a Test Case Manager start message to the execution results file. When processing a distributed test case this operation is only performed by the master TCM.
3. The TCM builds a list of test purposes to be executed from the list of requested invocable components. If no invocable components were requested, or if the special invocable component `all` is requested, the TCM builds this list from all of the invocable components in the test case.
4. The TCM arranges for the processing of asynchronous events.[13]

13. But see the section entitled ‘‘Portability’’ later in this chapter.

5. When processing a distributed test case, all the TCMs synchronise with each other before executing their respective start-up procedures (if any).
6. When processing a distributed test case, all the TCMs synchronise at the commencement of each invocable component.
7. The TCM prints an Invocable Component Start message to the execution results file before it executes each invocable component. When processing a distributed test case this message is printed by the master TCM.
8. When processing a distributed test case, all the TCMs synchronise at the commencement of each test purpose. During this synchronisation process the TCMs ensure that they are executing a common test purpose.
9. The TCM prints a Test Purpose Start message to the execution results file before it executes each test purpose. When processing a distributed test case this message is printed by the master TCM.
10. The TCM executes each test purpose in the invocable component.
11. During the test purpose any test case information lines generated by the user-supplied test code are entered into the execution results file.
12. If an event occurs which interrupts the processing of a test purpose, the interrupted process immediately proceeds to the end of the test purpose and outputs a test result of `UNRESOLVED`. If the event is a termination message from the TCC, the TCM reports receipt of the message to the execution results file, executes the specified clean-up procedure (if any) and exits;[14] otherwise it continues by processing the next test purpose.
13. If a TCM is about to execute a test purpose that has been marked as cancelled, it instead reports the test purpose as `UNINITIATED` and continues to process the next test purpose (if any). If a test purpose has been marked as cancelled in one part of a distributed test case, the TCM informs all the others of the cancellation during the automatic synchronisation at the start of the test purpose.
14. When processing a non-distributed test case, the TCM prints a test purpose result to the execution results file at the end of each test purpose. When processing a distributed test case, all the TCMs synchronise at the end of the test purpose. The execution results daemon gathers the partial results recorded by each test purpose part, arbitrates between them and prints the consolidated result to the execution results file.
15. Once all of the test purposes in an invocable component have been executed, the TCM outputs an Invocable Component End message to the execution results file and moves on to the next invocable component. When processing a distributed test case this message is only printed by the master TCM.
16. After all test purposes for all requested invocable components have been executed, the TCM executes the specified clean-up procedure (if any) and exits normally.

14. When the TCC terminates the TCM on a Win32 system, the TCM exits immediately without reporting the event or executing the clean-up procedure. See the section entitled ‘‘Portability’’ later in this chapter.

17. If for any reason, a particular TCM fails to execute the requested set of invocable components or the specified start-up or clean-up procedure, the relevant API will exit abnormally. When a distributed test case is being processed and one test case part terminates abnormally, this event is communicated to the other parts at the next automatic synchronisation point; whereupon they also terminate without executing any more test purposes.

7.3 C and C++ TCM options

In addition to the ordinary TCMs, each version of TETware provides versions of the C and C++ TCMs which are suitable for use in multi-threaded environments. When either version of TETware is built on a UNIX system, the user may choose to configure the thread-safe versions of the C and C++ TCMs to support either UI or POSIX threads (but not both at the same time).

It can be seen that there are a number of TCM options from which to choose. The test suite author should ensure that the correct API is used with each TCM. The following TCM/API options are available:

- TETware-Lite, C TCM/API, single-threaded version.
- TETware-Lite, C++ TCM, single-threaded version, use with the single-threaded C API library.
- TETware-Lite, C TCM/API, multi-threaded version.
- TETware-Lite, C++ TCM, multi-threaded version, use with the multi-threaded C API library.
- Distributed TETware, C TCM/API, single-threaded version.
- Distributed TETware, C++ TCM, single-threaded version, use with the single-threaded C API library.
- Distributed TETware, C TCM/API, multi-threaded version.
- Distributed TETware, C++ TCM, multi-threaded version, use with the multi-threaded C API library.
- TETware-Lite or Distributed TETware, Shell TCM/API.
- TETware-Lite or Distributed TETware, Korn Shell TCM/API.
- TETware-Lite or Distributed TETware, Perl TCM/API.

Note that test cases using the Distributed versions of the C and C++ TCMs must always be run under control of the Distributed TETware TCC. They cannot be run stand-alone or under the control of the TETware-Lite TCC. Test cases which use the other TCMs may be run stand-alone or under control of either TCC.

7.4 C and C++ TCMs in Distributed TETware

The information presented in this section applies only when the C and C++ TCMs are used with Distributed TETware. It does not apply when other TCMs or TETware-Lite is used.

When the Distributed version of a TCM is used to manage parts of the same distributed test case running on different systems, one TCM assumes the role of master and the others assume the role of slaves. However, unlike previous DTET implementations, the master system is not

constrained to be the system on which `tcc` is invoked. Indeed, it is possible for the Distributed TCC to control both distributed and non-distributed test cases which are processed entirely on remote systems.

The identity of master and slave TCM is determined as follows: when the Distributed TCC processes a test case, it does so with reference to a list of systems on which the test case is to be processed. Initially this list contains only one entry for the local system (system ID 0). However, this list is updated when `tcc` processes test cases specified within the scope of a `remote` or `distributed` directive in the test scenario. When `tcc` executes either a distributed or a non-distributed test case on a particular system, it communicates the system's ID to the TCM. When `tcc` determines that a test case is a distributed test case, it communicates the list of all the participating systems to the TCM that is managing each test case part. However, when `tcc` determines that a test case is a non-distributed test case, it communicates a system list to the TCM which contains only that TCM's system ID.

So a TCM always knows the ID of the system on which it is running and can determine whether or not it is processing a distributed test case by counting the number of system IDs in the list that it receives from `tcc`. A Distributed TCM acts as a master TCM if its system ID is the first (or only) ID in the list. Conversely, a Distributed TCM acts as a slave TCM if its system ID is the second or subsequent ID in the list.

Since all the parts of a distributed test case share the same execution results file, it is appropriate for only one of the test case parts to print TCM Start, IC Start, IC End and TP Start messages to the execution results file. This function is only performed by the master TCM in a distributed test case.

In addition, Distributed TCMs that are managing parts of a particular distributed test case on different systems all synchronise with each other at certain points during test case execution. These synchronisation points occur at the following times:

- At TCM startup time.
- Before the user-supplied startup function is called by the TCM.
- Before the first test purpose function in an invocable component is called by the TCM.
- Before each test purpose function is called by the TCM.
- When each test purpose function returns control to the TCM.
- Before the user-supplied cleanup function is called by the TCM.

Thus it can be seen that TCMs which support distributed testing ensure that each part of a distributed test case keeps in step with all of its peers throughout the execution process. Refer to the chapter entitled ‘‘Test case synchronisation’’ in the TETware User Guide for details of synchronisation between TCMs that are managing the different parts of a distributed test case.

7.5 Portability

On Win32 operating systems the C runtime support library does not really support the asynchronous event handling that is provided by the signal mechanism on a UNIX system. Therefore the Win32 versions of the C TCM do not attempt to handle asynchronous events of this type.

This and other issues related to the processing of test cases on Win32 systems are discussed in the appendix entitled ‘‘Implementation notes for TETware on Win32 systems’’ in the TETware User

Guide.

8. The C API

8.1 Introduction

This chapter describes the TETware C API. Different versions of the C API are supplied with TETware-Lite and Distributed TETware. The Distributed version of the C API may be used when writing both distributed and non-distributed test cases, whereas the Lite version of the C API may only be used when writing non-distributed test cases. The types of test case supported by each API version corresponds to the types of test case which may be processed by the TCC included with each TETware version.

The synopses here are described in accordance with the International C Standard ISO 9899. An ISO 9899 or Common Usage C (as defined in ISO 9945–1) conforming compiler is required to develop test cases using these interfaces. See the chapter entitled ‘‘Writing a C language API-conforming test suite’’ for an example of how to write a C language based test suite.

8.2 C language binding

On UNIX systems, test cases written to this language binding attach themselves to it through the following files:

- *tet-root*`/lib/tet3/libapi.a` contains the support routines for test purposes.
- *tet-root*`/lib/tet3/tcm.o` contains the TCM. This file contains the routine `main()` and associated support routines for the sequencing and control of invocable components and test purposes.
- *tet-root*`/lib/tet3/tcmchild.o` contains the child process controller. This file contains a `main()` routine which can be used by test suites when building processes which test purposes will launch using the `tet_exec()` and `tet_spawn()` interfaces.
- *tet-root*`/lib/tet3/tcmrem.o` contains the remote executed process controller. This file contains a `main()` routine which can be used by test suites when building processes which test purposes will launch using the `tet_remexec()` interface. Note that in Distributed TETware the use of `tet_remexec()` (and therefore the use of this file) is deprecated. It is possible that this file may be removed from a future TETware release. This file is not supplied in TETware-Lite.
- *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.

The names of these files are similar on Win32 systems; the differences are that object files (`.o` files) instead have a `.obj` suffix and library files (`.a` files) instead have a `.lib` suffix.

A test suite should access each of these files by means of its build tool, in a way which is appropriate for the available Software Generation System. Test suite authors are advised to allow easy specification of alternate path names for these files (possibly through TETware configuration variables), thus improving the flexibility of their suites.

A thread-safe version of the C API is supplied in addition to the standard (that is: single-threaded) version described here. Distinct versions of the thread-safe C API are supplied with Distributed TETware and TETware-Lite. Differences between the standard and thread-safe APIs are described in the chapter entitled ‘‘The Thread-safe C and C++ APIs’’ elsewhere in this guide.

The interfaces described in this chapter can also be used by test cases written to the C++ language binding, although the names of the files containing these interfaces are different. As with the C API, the C++ language binding is provided in both standard and thread-safe versions, and distinct versions of each are supplied with Distributed TETware and TETware-Lite. The C++ API is described in more detail in the chapter entitled ‘‘The C++ API’’ elsewhere in this guide.

8.3 TCC dependencies

Test cases built to the Lite version of this API may be either be executed stand-alone or under the control of either TCC version. Test cases built to the Distributed version of this API require the Distributed TCC to execute; they cannot be executed stand-alone. This is because the amount of effort required to establish an environment in which test cases could execute without the TCC is substantial. This applies especially to the requirement for test purpose synchronisation and result arbitration.

The TCC uses communication variables to pass information to the API. If the communication variables normally set by the TCC are not set when a test case is executed, `TET_ACTIVITY` defaults to 0 and `TET_CONFIG` to none. If `TET_CODE` is undefined or the file specified by `TET_CODE` does not exist in the current directory, the default set of result codes are used.

If the test case requires configuration variables or additional result codes, those communication variables should be set accordingly when a test case is executed stand-alone.

8.4 Test case structure and management

8.4.1 Introduction

These functions and variables are used when test cases are initialised and cleaned up, and in selecting invocable components and test purposes to execute. Some of these elements are provided by the API, whereas others must be defined in each test case.

There are two methods that may be used to specify the list of invocable components and test purposes in a test case, as follows:

- When the static method is used, the list of invocable components and test purpose functions in the test case must be specified by the test suite author at compile time in a static array which is part of the user-supplied test code. The TCM determines which invocable components and test purposes to execute by inspecting the contents of this array.
- When the dynamic method is used, the list of invocable components and test purpose functions in the test case may be specified at run time by the test case itself. The TCM calls certain test case interface functions to determine which invocable components and test purpose functions exist within the test case, and to invoke each test purpose function which is to be executed. When the dynamic method is used, all of the test case interface functions must be provided in the user-supplied test code.

The method used by the TCM to determine which invocable components and test purpose functions have been specified in the test case depends on whether or not the test case contains these interface functions. If the test case contains the set of interface functions that are called by the TCM, then those functions are used. Otherwise, the TCM uses a default set of interface functions that are part of the API library; this set implements the interface to the static `tet_testlist[]` array.

The two methods that may be used to specify invocable components and test purpose functions are described in more detail in the two subsections that follow.

8.4.2 Static test case interface – the `tet_testlist[]` array

Synopsis

```
struct tet_testlist {
        void (*testfunc)(void);
        int icref;
};

struct tet_testlist tet_testlist[];
```

Description

The `tet_testlist[]` array consists of an array of `tet_testlist` structures. When the static method is used to specify the invocable components and test purpose functions that must be defined in the user-supplied test code. Each element in this array specifies a user-supplied test purpose function that may be called by the TCM.

Members of the `tet_testlist` structure have meanings as follows:

`testfunc` A pointer to the test purpose function.

`icref` The number of the invocable component to which this function belongs.

The `tet_testlist[]` array is terminated by a structure with the `testfunc` element set to `NULL`. No other element of the array will use the value `NULL` for this element.

For each requested invocable component, the TCM scans the `tet_testlist[]` array and executes, in order, each test purpose that is associated with that invocable component. When `all` invocable components are requested, the TCM executes all ICs for which entries are defined in the `tet_testlist[]` array, in order of ascending IC number. In both cases the TCM will calculate the number of test purposes that are to be executed for each requested invocable component.

The TCM does not perform any error checking on the contents of the `tet_testlist[]` array. It is the test author's responsibility to ensure that the contents of the array is correctly specified. In particular, it should be noted that in a distributed test case the `tet_testlist[]` structure must be exactly replicated on each system that is to participate in the test and, therefore, contain the same number of members. This may require the inclusion of test purposes on some systems that do nothing except register a result of `PASS`.

Application notes

When a test case contains a `tet_testlist[]` array, it may not contain any of the interface functions described in the next subsection.

8.4.3 Dynamic test case interface – `tet_getmaxic()`, `tet_getminic()`, `tet_isdefic()`, `tet_gettpcount()`, `tet_gettestnum()` and `tet_invoketp()`

Synopsis

```
int tet_getmaxic(void);
int tet_getminic(void);

int tet_isdefic(int icnum);

int tet_gettpcount(int icnum);

int tet_gettestnum(int icnum, int tpnum);

int tet_invoketp(int icnum, int tpnum);
```

Description

The TCM calls these functions to determine which invocable components and test purpose functions have been specified in the test case. If any of these functions are provided in the user-supplied code, they must all be provided. If the user-supplied code does not contain this set of functions, the TCM uses a default set that is provided in the API library.

The `tet_getmaxic()` and `tet_getminic()` functions should return the highest and lowest invocable component number that are defined in the test case. Invocable component numbers should be positive values starting at 1. It is permissible for gaps to exist in the range of invocable component numbers that are defined in the test case, but the TCM operates less efficiently when this is the case.

The TCM calls `tet_getmaxic()` and `tet_getminic()` when building the list of invocable components to be executed. This operation is performed soon after the start of processing, before the test case startup function[15] is called.

The `tet_isdefic()` function should return 1 if the invocable component specified by `icnum` is defined in the test case, or 0 if the invocable component is not defined.

The TCM calls `tet_isdefic()` when building the list of invocable components to be executed, and before executing each invocable component in the list. It is likely that the TCM will call this function more than once for each invocable component number between the values returned by calls to `tet_getmaxic()` and `tet_getminic()`. It is the responsibility of the test suite author to ensure that a call to `tet_isdefic()` for a particular `icnum` always returns the same value.

The `tet_gettpcount()` function should return the number of test purposes that have been defined in the invocable component specified by `icnum`, or 0 if the invocable component specified by `icnum` has not been defined in the test case.

The TCM calls `tet_gettpcount()` once for each invocable component that is to be executed, before invoking each test purpose function defined for that invocable component.

The `tet_gettestnum()` function should return the absolute test number for the test purpose specified by `tpnum` within the scope of the invocable component specified by `icnum`. If this test purpose has not been defined in the test case, `tet_gettestnum()` should return 0.

The test purpose number specified by `tpnum` and the absolute test number returned by this function both start at 1. For example, consider a test case which contains three invocable components. The first and third invocable components each contain two test purposes, and the second invocable component contains four test purposes. The following table shows the absolute test number that should be returned by a call to `tet_gettestnum(`*icnum*`,` *tpnum*`)` for each of the defined test purposes in this hypothetical test case:

Invocable component number (*icnum*)	**Test purpose number** (*tpnum*)	**Value to be returned by** `tet_gettestnum()`
1	1	1
1	2	2
2	1	3
2	2	4
2	3	5
2	4	6
3	1	7
3	2	8
any other (*icnum*, *tpnum*) combination		0

The TCM calls `tet_gettestnum()` immediately before invoking the specified test purpose function. The value returned is used to initialise the global variable `tet_thistest` which is described in a later subsection.

15. That is: the function specified by `(*tet_startup)()`.

The `tet_invoketp()` function should invoke the test purpose function specified by `tpnum` within the scope of the invocable component specified by `icnum`. In this context the first test purpose in a particular invocable component *icnum* has a `tpnum` of 1, and the last test purpose in the same invocable component has a `tpnum` of `tet_gettpcount(`*icnum*`)`.

The TCM calls `tet_invoketp()` in order to invoke the specified test purpose function. The return value of this function is reserved for future use; for now, it should always return 0.

Application notes

When any of the interface functions described in this section are provided in a test case, they must all be provided.

When a test case contains the set of interface functions that are described in this section, it may not contain the `tet_testlist[]` array that is described in the previous section.

It is possible to use the API functions which write information lines to the journal and access the values of configuration variables from these functions.

It is the responsibility of the provider of these functions to ensure that they behave in a consistent manner. For example, the results are undefined if a call to `tet_isdefic()` for a particular `icnum` returns 1 and a subsequent call for the same `icnum` returns 0. Likewise, the results are undefined if a call to `tet_getmaxic()` or `tet_getminic()` indicates that a particular invocable component is defined and a subsequent call to `tet_isdefic()` or `tet_gettpcount()` indicates that the invocable component is not defined.

When these functions are used in parts of a distributed test case, total chaos will break out if when called with particular arguments they behave differently in the different test case parts.

By convention the absolute test number should increase uniformly for each test purpose defined within the test case, just as it does when the static test case interface is used. Although the TCM does not itself rely on this behaviour, test suite authors are reminded that failure to observe this convention might confuse a report writer.

The flexibility provided by the dynamic test case interface makes it possible for a test case to alter its execution path on-the-fly, possibly depending on factors which can change between test runs. When this capability is used it is possible that the journal produced might yield unexpected results when used as input to a subsequent `tcc` run when the Rerun or Resume options are specified.

The TCM will not call any of the functions `tet_gettpcount()`, `tet_gettestnum()` and `tet_invoketp()` with an `icnum` value which has not been returned by a previous call to `tet_getmaxic()` or `tet_getminic()`, or indicated as valid by a previous call to `tet_isdefic(`*icnum*`)`. Likewise, the TCM will not call any of the functions `tet_gettestnum()` and `tet_invoketp()` with a known good `icnum` value and a `tpnum` value which is outside of the range 1 through `tet_gettpcount(`*icnum*`)`.

In TETware-Lite there is no practical limit to the number of invocable components and test purpose functions that may be defined in a test case. In Distributed TETware the maximum value of an `icnum` is limited to 32766 and the maximum value of a `tpnum` is limited to 32767.

8.4.4 `tet_startup` and `tet_cleanup`

Synopsis

```
void (*tet_startup)(void);
void (*tet_cleanup)(void);
```

Description

The function pointers `tet_startup` and `tet_cleanup` must be defined in the user-supplied test code. These pointers may be initialised with the addresses of the functions to be used to perform test case specific start up and clean up procedures, respectively. The start up procedure is executed before the first requested invocable component and the clean up procedure is executed on completion of the last requested invocable component. These routines are executed irrespective of which invocable components are requested. If a test case does not need to perform actions on start up and/or clean up, the corresponding pointer should be initialised to `TET_NULLFP` (a NULL function pointer, defined in `tet_api.h`).

8.4.5 `tet_thistest`, `tet_nosigreset` and `tet_pname`

Synopsis

```
int tet_thistest;
int tet_nosigreset;
char *tet_pname;
```

Description

These variables are provided by the API.

When the static test case interface is used, the `tet_thistest` variable contains the sequence number (starting at 1) of the element in the `tet_testlist[]` array that is associated with the currently executing test purpose.

When the dynamic test case interface is used, the `tet_thistest` variable contains the absolute test number that is returned by the `tet_gettestnum()` function as described in a previous subsection.

During execution of the start up and clean up functions, `tet_thistest` is set to zero.

The value of `tet_nosigreset` determines whether or not the TCM reinstates signal handlers for unexpected signals before each test purpose function is called. Initially this variable contains a value of zero but this may be changed by the user-supplied test code. The default value of zero means that signal handlers will be reinstated before each test purpose, in order to ensure that unexpected signals do not go unnoticed if an earlier test purpose installed a local handler but does not restore the original handler before returning control to the TCM.

If `tet_nosigreset` is set to a non-zero value in the start up function called via `(*tet_startup)()`, then signal handlers will be left in place between test purposes. In test cases where stray signals constitute a test failure, it is recommended that `tet_nosigreset` is left with its default value of zero. This is because, even if test purposes contain code to restore the signal handling, this code will not be executed if an unexpected signal arrives and the TCM skips to the start of the next test purpose.

The `tet_pname` variable points to the process name as given on the test case command line. This variable is also provided in subprograms that are linked with one of the TETware child

process controllers.

Portability

Setting the value of `tet_nosigreset` has no effect on a Win32 system.

8.5 Insulating from the test environment

Description

The following configuration variables are used by the C language TCM to help determine which events should be handled for the test case, and which should be passed through. They are used by the TCM to support functionality to insulate test cases from the test environment.

`TET_SIG_IGN` defines (by comma separated number) the set of signals that are to be ignored during test purpose execution. Any signal that is not set to be ignored or to be left (see `TET_SIG_LEAVE` below) with its current disposition, will be caught when raised and the result of the test purpose will be set to `UNRESOLVED` because of the receipt of an unexpected signal. A test purpose may undertake its own signal handling as required for the execution of that test purpose. The disposition of signals will be reset after the test purpose has completed, unless the global variable `tet_nosigreset` is non-zero. The TCM needs to know how many signals the implementation supports in order to set up catching functions for these signals.

`TET_SIG_LEAVE` defines (by number) the set of signals that are to be left unchanged during test execution. In most cases this will mean that the signal takes its default action. However, the user can change the disposition of the signal (to ignore) before executing the TCC if this signal is to remain ignored during the execution of the test purposes.

The implementation on UNIX systems does not allow the signals defined by POSIX.1 (ISO 9945–1) to be set to be ignored or left unchanged, as this may pervert test results.

Portability

The facilities described here are not provided on Win32 systems.

8.6 Error handling and reporting

8.6.1 Introduction

Many of the API functions return error indications. The API provides the following variables for use when determining and reporting the cause of these errors.

8.6.2 `tet_errno`

Synopsis

```
int tet_errno;
```

Description

When an API function returns a value which indicates that an error has occurred, the API stores a value in the global `tet_errno` variable which indicates the cause of the error. The API does not alter the value of this variable when a call to an API function is successful.

Distributed TETware uses a client/server architecture and calls to several of the API functions cause the API to send requests to server processes. A server sends a reply code in response to each request that it receives. When the reply code indicates that a request has failed, the value stored in `tet_errno` is derived from the server reply code. A list of the server reply codes and their meanings is presented in the appendix entitled ‘‘Server reply codes’’ in the TETware User Guide.

The following error codes may be used by the API. These codes are defined in `tet_api.h`. Note that not all of them may be visible outside of the API.

`TET_ER_2BIG`	Argument list too long.
`TET_ER_ABORT`	Abort TCM on TP end.
`TET_ER_CONTEXT`	Request out of context.
`TET_ER_DONE`	Event finished or already happened.
`TET_ER_DUPS`	Request contained duplicate IDs.
`TET_ER_ERR`	General error code.
`TET_ER_FID`	Bad identifier in file i/o request.
`TET_ER_FORK`	Can’t fork.
`TET_ER_INPROGRESS`	Event in progress.
`TET_ER_INTERN`	Server internal error.
`TET_ER_INVAL`	Invalid parameter.
`TET_ER_LOGON`	Not logged on to server.
`TET_ER_MAGIC`	Bad magic number in server request.
`TET_ER_NOENT`	No such file or directory.
`TET_ER_PERM`	Privilege request/kill error.
`TET_ER_PID`	No such process.
`TET_ER_RCVERR`	Receive message error.
`TET_ER_REQ`	Unknown request code.
`TET_ER_SIGNUM`	Bad signal number.
`TET_ER_SNID`	Bad sync identifier in SYNCD request.
`TET_ER_SYNCERR`	Sync completed unsuccessfully.
`TET_ER_SYSID`	System identifier not in system name list.
`TET_ER_TIMEDOUT`	Request or system call timed out.
`TET_ER_TRACE`	Tracing not configured.

TET_ER_WAIT	Process not yet terminated.
TET_ER_XRID	Bad execution results file identifier in XRESD request.

Whenever an unsuccessful API call sets tet_errno to TET_ER_ERR (the general error code), a diagnostic message is generated somewhere which contains more precise details of the cause of the error. If an error of this type occurs in the API library, the diagnostic is printed to the execution results file as a TCM/API message if possible; if this is not possible, the diagnostic is printed on the test case's standard error stream.

However, in Distributed TETware, an error of this type can also occur in a server process. In this case the more detailed error message is printed on the server's standard error stream. The result of this is that when an API call is unsuccessful in Distributed TETware and tet_errno is set to TET_ER_ERR, the more detailed error message often appears in a TCCD log file on the local system or on one of the remote systems that is participating in the test run.

8.6.3 tet_errlist[] and tet_nerr

Synopsis

```
char *tet_errlist[];

int tet_nerr;
```

Description

The tet_errlist[] array contains short text strings, similar to those listed in the previous section, which describe each of the values defined for tet_errno.

When a call to an API function is unsuccessful, the string obtained when the value of tet_errno is used to index the tet_errlist[] array may be used when an information line is printed to the execution results file by the test case.

The global variable tet_nerr is initialised to the number of strings provided in the tet_errlist[] array. The value of tet_errno should be checked against tet_nerr before using it to index the array in order to guard against the possibility that a new error code is added to the API before the corresponding message is added to the array.

8.7 Making journal entries

8.7.1 Introduction

These functions are provided for use by test cases when making entries in the execution result file.

8.7.2 `tet_setcontext()` and `tet_setblock()`

Synopsis

```
void tet_setcontext(void);
void tet_setblock(void);
```

Description

The `tet_setcontext()` function sets the current context to the value of the current process ID, and resets the block and sequence numbers to 1. A call to `tet_setcontext()` should be made by any application which executes a `fork()` to create a new process and which wishes to write entries from both processes. The call to `tet_setcontext()` must be made from the child process, not from the parent.

The `tet_setblock()` function increments the current block ID. The value of the current block ID is reset to one at the start of every test purpose or after a call to `tet_setcontext()` which altered the current context. The sequence ID of the next entry, a number which is automatically incremented as each entry is output to the execution results file, is set to one at the start of each new block.

Return value

These functions do not return a value.

Portability

The thread-safe version of `tet_setcontext()` does not reset the block and sequence numbers.

8.7.3 `tet_infoline()`, `tet_minfoline()`, `tet_printf()` and `tet_vprintf()`

Synopsis

```
void tet_infoline(char *line);
int tet_minfoline(char **lines, int lines);
int tet_printf(char *format, /* [arg,] */ ...);
int tet_vprintf(char *format, va_list ap);
```

Description

A call to `tet_infoline()` prints the information line specified by `line` to the execution results file. The sequence number is incremented by one after the line is output. If the current context and the current block ID have not been set, the call to `tet_infoline()` causes the current context to be set to the value of the calling process ID and the current block ID to be set to one.

A call to `tet_minfoline()` prints groups of information lines to the execution results file. In Distributed TETware these lines are printed using a single operation which guarantees that lines from other test case parts do not appear in between lines printed by a particular call to this function. `lines` points to the first in a list of pointers to strings which are to be written to the execution results file in a single operation. A NULL pointer in the list is ignored. `nlines` specifies the number of pointers in the list.

A call to `tet_printf()` formats the string specified by `format` which may contain `printf()`-like conversion specifications and prints it to the execution results file as one or more test case information lines. If after formatting the string is to contain more than one information line, each line except the last should be delimited by a newline character. If the formatted string contains a line that is longer than the maximum permitted for a journal information line, the API adds extra newlines in order to break the long line into two or more shorter lines. If possible, a newline added by the API will replace a blank character in the string so that the string is broken on a word boundary. When formatting is complete, the lines are written to the execution results file as if by a call to `tet_minfoline()`.

The operation of `tet_vprintf()` is the same as that described for `tet_printf()` except that, instead of being called with a variable number of arguments, it is called with a variable argument list.

Return value

A call to `tet_infoline()` does not return a value.

A successful call to `tet_minfoline()` returns zero. If a call to `tet_minfoline()` is unsuccessful, –1 is returned and `tet_errno` is set to indicate the cause of the error.

A successful call to `tet_printf()` or `tet_vprintf()` returns the number of bytes written to the execution results file. If a call to `tet_printf()` or `tet_vprintf()` is unsuccessful, –1 is returned and `tet_errno` is set to indicate the cause of the error.

8.7.4 tet_result()

Synopsis

```
void tet_result(int result);
```

Description

A call to `tet_result()` informs the API of the result of the test purpose from which it is called. The API generates a TP result line which is printed to the execution results file by the TCM upon test purpose completion. This ensures that all informational messages are written out before the test purpose result, and that there is one (and only one) result generated per test purpose. If the result code specified by `result` is one for which the action specified in the result codes file is to abort testing, then the TCM will exit after the test purpose has completed. If an immediate abort is desired, then the test purpose should execute a `return` statement immediately after the call to `tet_result()`.

If a test purpose does not call `tet_result()`, the TCM will generate a result of `NORESULT`. If more than one call to `tet_result()` is made with different result codes, the TCM determines the final result code by use of precedence rules. The precedence order (highest first) is:

`FAIL`
`UNRESOLVED`, `UNINITIATED`
`NORESULT` (i.e., invalid result codes)
Test suite supplied codes
`UNSUPPORTED`, `UNTESTED`, `NOTINUSE`
`PASS`

Where two or more codes have the same precedence then all calls to `tet_result()` with one of those codes are ignored except the first such call.

The `tet_result()` function should only be called from within the scope of a test purpose function. It must not be called from a test case start up or clean up function.

Return value

This function does not return a value.

8.8 Cancelling test purposes

8.8.1 Introduction

These functions are provided for use when cancelling test purposes.

8.8.2 `tet_delete()`

Synopsis

```
void tet_delete(int testno, char *reason);
```

Description

A call to `tet_delete()` marks the test purpose specified by the absolute test number `testno` as cancelled. When the static test case interface is used, the test purpose to be cancelled is the one defined in the element specified by `tet_testlist[`*testno* – 1`]`. When the dynamic test case interface is used, the test purpose to be cancelled is the one identified by the `tet_gettestnum()` function. If the test purpose specified by `testno` is not defined in the test case, a call to `tet_delete()` has no effect.

`reason` should point to a text string which describes the reason why the test purpose is to be marked as cancelled. This string should be contained in a static area.

When the TCM prepares to call a test purpose function, it first checks to see if the function has been marked as cancelled by a call to `tet_delete()`. If the test purpose has been marked as cancelled, the TCM does not call the function but instead prints the line pointed to by `reason` to the execution results file and records a result of `UNINITIATED`.

If a call to `tet_delete()` names a `testno` that has been marked as cancelled by a previous `tet_delete()` call, the reason for cancellation is changed to the `reason` specified in the current call.

If `tet_delete()` is called with a NULL `reason` parameter, the test purpose specified by `testno` is reactivated if it has previously been marked as cancelled.

Note that the string pointed to by a non-NULL `reason` parameter is not copied by the API when `tet_delete()` is called. Therefore it must point to static data, as the calling function will have terminated when the reason string is accessed by the TCM. Also, care should be taken not to re-use a buffer that has previously been passed to `tet_delete()`.

If `tet_delete()` is called in a distributed test case, the API notifies other participating TCMs of the cancellation. This notification occurs when the TCMs synchronise with each other before attempting to execute the cancelled test purpose. Thus, none of the TCMs execute a distributed test purpose which has been cancelled on any of the participating systems.

Return value

This function does not return a value.

Application notes

The `tet_delete()` function can only usefully be called from a top-level process; that is, a process which has been linked with the TCM module. It has no effect when called from a child process; that is, in the `(*childproc)()` function after a call to `tet_fork()` or in a process which has been linked with one of the child process controllers.

8.8.3 `tet_reason()`

Synopsis

```
char *tet_reason(int testno);
```

Description

The function `tet_reason()` returns a pointer to a string which contains the reason why the test purpose with the specified absolute test number has been cancelled by a previous call to `tet_delete()`. If this test purpose is not defined in the test case or is not marked as cancelled, a value of `(char *) NULL` is returned.

Return value

If the specified test purpose exists and has been cancelled by a previous call to `tet_delete()`, a call to `tet_reason()` returns the `reason` parameter supplied with the `tet_delete()` call; otherwise, a NULL pointer is returned.

Application notes

The `tet_reason()` function can only usefully be called from a top-level process; that is, a process which has been linked with the TCM module. The return value of a call to `tet_reason()` is undefined in a child process; that is, in the `(*childproc)()` function after a call to `tet_fork()` or in a process which has been linked with one of the child process controllers.

It is not possible to use `tet_reason()` in a distributed test case to determine whether or not a remote test purpose part has been cancelled.

8.9 Accessing configuration variables

8.9.1 Introduction

This function provides access to configuration variables. A description of how configuration variables are defined is presented in the chapter entitled ‘‘Configuration files’’ elsewhere in this guide. Note that when a test case or tool is processed by the TCC, this function only provides access to variables that are defined for the current mode of operation.

When Distributed TETware is used, this function provides access to the per-system configuration defined for the system on which the calling process is running. This function cannot be used to access configuration variables defined on other systems or distributed configuration variables.

8.9.2 `tet_getvar()`

Synopsis

```
char *tet_getvar(char *name);
```

Description

A call to `tet_getvar()` returns a pointer to the value of the configuration variable `name`. This pointer will remain valid for the life of the process, regardless of subsequent calls to `tet_getvar()`.

If the variable specified by `name` is defined but has no setting, `tet_getvar()` returns a pointer to an empty string. If the variable specified by `name` is undefined, `tet_getvar()` returns a `NULL` pointer.

Return value

This function returns the values described above.

8.10 Generating and executing processes

8.10.1 Introduction

These functions enable API-conforming child processes and subprograms to be created and administered.

8.10.2 `tet_fork()`, `tet_exec()` and `tet_child`

Synopsis

```
int tet_fork(void (*childproc)(void), void (*parentproc)(void),
        int waittime, int validresults);

int tet_exec(char *file, char *argv[], char *envp[]);

extern pid_t tet_child;
```

Description

The `tet_fork()` function creates a new process which is a copy of the calling process and, unless a negative `waittime` is specified, modifies the signal disposition in the newly created process such that any signals that were being caught in the parent process are set to their default values in the child process. Then the function specified by `(*childproc)()` is called in the child process. If this function returns, the child process terminates with an exit status of 0. Alternatively, the `(*childproc)()` function may terminate the child process with a specific exit status by means of a call to `tet_exit()` or overlay the child process by means of a call to `tet_exec()`.

If `(*parentproc)()` is not set to `NULL`, the function specified by `(*parentproc)()` is called in the parent process. Then the parent process waits for the child process to terminate and obtains the child's exit status. Then, the bits which are set in `validresults` are cleared in the child's exit status value. If the result of this operation is zero, `tet_fork()` assumes that the child process terminated with a valid (or expected) exit status. Otherwise, `tet_fork()` assumes that the child process terminated with an unexpected exit status and reports this exit status to the execution results file.

If the value of the child's exit status is one of the expected values, `tet_fork()` returns the child's exit status; otherwise, `tet_fork()` returns a value of −1 if the child's exit status is unexpected or some other error occurs. When `tet_fork()` returns −1, it reports the nature of the error using `tet_infoline()` and sets the test purpose result code to `UNRESOLVED` by calling `tet_result()`.

If a positive `waittime` is specified, the parent process will ensure that the child process does not continue to execute for more than `waittime` seconds after the completion of the optional `(*parentproc)()` function. If `waittime` is zero, the parent process will wait indefinitely for the child process to terminate. If a negative `waittime` is specified, the signal dispositions in the child process are not modified, the parent process does not wait for the child process to terminate and the value of `validresults` is ignored. When a negative `waittime` is specified, it is the responsibility of the `(*parentproc)()` function to wait for the child process and interpret its exit status.

`tet_exec()` may be called from a `(*childproc)()` routine of a child process generated by a call to `tet_fork()`. `tet_exec()` passes the argument data specified by `argv[]` and the environment data specified by `envp[]` to the process specified by `file`. The usage of the

`tet_exec()` is equivalent to that of the ISO 9945–1 `execve()` function, except that the API adds arguments and environment data that are to be interpreted by the driver of the executed `file`.

The interface between `tet_exec()` and the subprogram launched by it has been designed to enable the subprogram to use the API. Therefore the subprogram that is launched by a call to `tet_exec()` must be built with the child process controller `tcmchild.o`.

If `tet_exec()` is called without first calling `tet_fork()`, the results are undefined. This is because the `tet_fork()` function makes calls to `tet_setcontext()` in the child and `tet_setblock()` in the parent to distinguish output from the child and from the parent before, during and after execution of the `(*parentproc)()` function.

The global variable `tet_child` is provided by the API for use in the `(*parentproc)()` function called from `tet_fork()`. It is set to the process ID of the child.

Return value

A successful call to `tet_fork()` returns the exit status of the child process. If an error occurs, the child process terminates abnormally[16] or the child's exit status is not one of the values specified by `validresults`, –1 is returned and `tet_errno` is set to indicate the cause of the error.

A successful call to `tet_exec()` does not return. If a call to `tet_exec()` is unsuccessful, –1 is returned and `tet_errno` is set to indicate the cause of the error.

Portability

`tet_fork()`, `tet_exec()` and `tet_child` are not provided on Win32 systems. Test suite authors should instead use `tet_spawn()` and `tet_wait()` when writing portable test cases.

16. That is: `WIFEXITED(`*wait-status*`)` in the parent process is false.

8.10.3 `tet_spawn()`

Synopsis

```
pid_t tet_spawn(char *file, char *argv[], char *envp[]);
```

Description

A call to `tet_spawn()` creates a subprogram without the need to call `tet_fork()` first. The meanings of the arguments to `tet_spawn()` are the same as the meanings of the arguments to `tet_exec()`, described previously.

The interface between `tet_spawn()` and the subprogram launched by it has been designed to enable the subprogram to use the API. Therefore the subprogram that is launched by a call to `tet_spawn()` must be built with the child process controller `tcmchild.o` (on UNIX systems) or `tcmchild.obj` (on Win32 systems).

Return value

A successful call to `tet_spawn()` returns the process identifier of the newly created process. If a call to `tet_spawn()` is unsuccessful, –1 is returned and `tet_errno` is set to indicate the cause of the error.

Portability

Test case authors are reminded that process identifiers are reallocated rather more frequently on a Win32 system than they are on a typical UNIX system. Therefore, the use of the value returned by `tet_spawn()` to generate a unique quantity (such as a temporary file name) is likely to be less successful on a Win32 system than on a UNIX system.

On a Win32 system the type of the return value of this function (`pid_t`) is defined as an `int` in `tet_api.h`. This corresponds to the type of the value returned by the underlying `_spawnve()` function in the C runtime support library. This value is actually a `HANDLE` value, so it is only meaningful within the context of the calling process. Therefore it should be remembered that this value is not the same as the value that would be returned by a call to `_getpid()` in the newly created process.

8.10.4 `tet_wait()`

Synopsis

```
int tet_wait(pid_t pid, int *statp);
```

Description

A call to `tet_wait()` waits for the process identified by `pid` to terminate and returns that process's exit status indirectly through `*statp`. `pid` is the process identifier returned by a previous successful call to `tet_spawn()`.

Return value

A successful call to `tet_wait()` returns zero. If a call to `tet_wait()` is unsuccessful, –1 is returned and `tet_errno` is set to indicate the cause of the error.

Portability

On a UNIX system, the value returned indirectly through `*statp` is obtained from the `waitpid()` system call. On a Win32 system, the value returned indirectly through `*statp` is obtained from a call to the `_cwait()` function in the C runtime support library. Test suite authors are reminded that the encodings of the process exit status values returned by these two functions are likely to be different.

8.10.5 `tet_kill()`

Synopsis

```
int tet_kill(pid_t pid, int sig);
```

Description

A call to `tet_kill()` sends the signal specified by `sig` to the process specified by `pid`, which should be the process identifier returned by a previous successful call to `tet_spawn()`.

Return value

A successful call to `tet_kill()` returns zero. If a call to `tet_kill()` is unsuccessful, –1 is returned and `tet_errno` is set to indicate the cause of the error.

Portability

The `sig` parameter is ignored on a Win32 system; instead, a different method is used to terminate the process specified by `pid`.

Application notes

Test case authors are discouraged from using `tet_kill()` to terminate a process which is running on a Win32 system. The reasons for this are discussed in the appendix entitled ''Implementation notes for TETware on Win32 systems'' in the TETware User Guide.

8.11 Executed process functions

8.11.1 Introduction

These functions are provided for use by API-conforming processes that are launched by calls to the `tet_exec()`, `tet_spawn()` and `tet_remexec()` functions.

8.11.2 `tet_main()`

Synopsis

```
int tet_main(int argc, char *argv[]);
int tet_thistest;
char *tet_pname;
```

Description

The function `tet_main()` must be supplied by the test suite developer. This function is called by the `main()` function of the TETware child process controller. Prior to calling `tet_main()`, the child process controller sets the value of the `tet_thistest` variable to the value of `tet_thistest` in the process that called `tet_exec()`, `tet_spawn()` or `tet_remexec()`. This value should not be changed by the executed process.

The current context is preserved from the calling process and the current block is incremented by one before `tet_main()` is called.

If `tet_main()` returns, its return value becomes the child process's exit status. If the child process was started by a call to `tet_exec()`, the child process's exit status will be returned to the process which called the `tet_fork()` function; in this case, the value returned from `tet_main()` will usually match one of the valid result values specified in the call to `tet_fork()`. If the child process was started by a call to `tet_spawn()`, the child process's exit status may be accessed in the parent by a call to `tet_wait()`. If the child process was started by a call to `tet_remexec()`, the child process's exit status may be accessed in the parent by a call to `tet_remwait()`.

The `tet_pname` variable in the child process contains the process name as given in the `argv[0]` parameter to `tet_main()`.

Return value

If the user-supplied `tet_main()` function returns a value to the child process controller, this value becomes the child process's exit status.

8.11.3 `tet_exit()` and `tet_logoff()`

Synopsis

```
void tet_exit(int status);
void tet_logoff(void);
```

Description

The function `tet_exit()` should be used instead of `exit()` by a child process that is started by a call to `tet_fork()`, `tet_exec()`, `tet_spawn()` or `tet_remexec()`. In Distributed TETware this function logs off all TETware servers, then calls `exit()` with the specified `status` as argument. `tet_exit()` should only be called from the child process that is started by `tet_fork()`, `tet_exec()`, `tet_spawn()` or `tet_remexec()` and not by any of its children.

The function `tet_logoff()` may be called by a child process that is started by a call to `tet_fork()`, `tet_exec()`, `tet_spawn()` or `tet_remexec()` which does not need to make any further TETware API calls and is not able to call `tet_exit()` at process termination time (for example: if one of the flavours of `exec()` is about to be called in the child process). `tet_logoff()` should only be called once from the child process. In Distributed TETware the results are undefined if a process or any of its descendents makes any TETware API calls after `tet_logoff()` is called.

Return value

A successful call to `tet_exit()` does not return.

A call to `tet_logoff()` does not return a value.

Portability

In TETware-Lite a call to `tet_exit()` simply calls `exit()` and a call to `tet_logoff()` has no effect.

8.12 Test case synchronisation

8.12.1 Introduction

These functions enable parts of a distributed test purpose or a user-supplied startup or cleanup function that are running on different systems to synchronise to an agreed point in the executing code. They are only available for use in distributed test cases.

Refer to the chapter entitled ‘‘Test case synchronisation’’ in the TETware User Guide for an overview of TETware synchronisation and a description of how to interpret journal messages that are generated by the default sync error handling function.

8.12.2 `tet_remsync()`

Synopsis

```
int tet_remsync(long syncptno, int *sysnames, int nsysname,
        int waittime, int vote, struct tet_synmsg *msgp);
```

Description

A call to `tet_remsync()` causes the calling process’s system to synchronise with one or more of the other systems that are participating in the same distributed test case. The call can only succeed if each of the systems specified in the call also expect to synchronise with each other and with the calling process.

`sysnames` points to a list of IDs of the other systems with which the calling process wishes to synchronise. `nsysname` specifies the number of systems in the list. The system ID of the calling process is ignored if it appears in the list pointed to by `sysnames`.

`syncptno` specifies the sync point number to which the calling process wishes to synchronise. If `syncptno` is zero, a successful call to `tet_remsync()` returns as soon as all participating systems have synchronised to the next sync point. If `syncptno` is greater than zero, a successful call to `tet_remsync()` returns as soon as all participating systems have synchronised using a sync point number which is not less than `syncptno`. When `syncptno` is greater than zero, a call to `tet_remsync()` will fail if a sync point has already occurred during the lifetime of the current test case whose number is greater than or equal to `syncptno`. The results are undefined if a negative `syncptno` is specified.

`waittime` specifies the number of seconds that may elapse between synchronisation requests from other participating systems before the calling process times out. If `waittime` is greater than zero, a call to `tet_remsync()` will be successful if all the participating systems synchronise to the specified sync point with no more than `waittime` seconds between each request. If `waittime` is zero, a call to `tet_remsync()` will return immediately, whether or not it is successful. If `waittime` is negative, a call to `tet_remsync()` will wait indefinitely for the specified sync point to occur or until the request fails for some reason. Test suite authors should be aware of the potential for deadlock if a negative `waittime` is specified.

`vote` specifies how the calling system wishes to vote in the synchronisation event. This parameter should be set to one of the defined constants `TET_SV_YES` or `TET_SV_NO`, to indicate a **yes** vote or a **no** vote, respectively. When the calling process specifies a **yes** vote, a call to `tet_remsync()` can only be successful if all the other participating systems also specify a **yes** vote. When the calling process specifies a **no** vote, the API does not use the votes specified by the other participating systems when determining whether or not a call to `tet_remsync()` in that process is successful.

It is possible for a process which calls `tet_remsync()` to exchange sync message data with other participating systems which synchronise exactly to the sync point specified by `syncptno`. This is done by calling `tet_remsync()` with a non-NULL value of `msgp`. When `msgp` is non-NULL, it points to a user-supplied `tet_synmsg` structure which contains the following elements:

```
struct tet_synmsg {
        char *tsm_data;
        int tsm_dlen;
        int tsm_sysid;
        int tsm_flags;
};
```

When `tet_remsync()` is called by parts of a distributed test purpose, one system sends data which may be received by other systems. The API associates the sync message data with the particular sync point specified by the `syncptno` parameter used in the `tet_remsync()` call on the sending system. In order to receive the message data, the `syncptno` parameter in calls to `tet_remsync()` on receiving systems must reference this sync point exactly, either by specifying the same value for `syncptno` as that used on the sending system, or by specifying a zero `syncptno`.

The test purpose part on the sending system should indicate a desire to send sync message data by initialising members of the `tet_synmsg` structure as follows before `tet_remsync()` is called:

- `tsm_data` points to the message to be sent.
- `tsm_dlen` is set to the number of bytes of message data to be sent.
- `tsm_flags` is set to `TET_SMSNDMSG`.

The test purpose part(s) on the receiving system(s) should indicate their willingness to receive sync message data by initialising members of the `tet_synmsg` structure as follows before `tet_remsync()` is called:

- `tsm_data` points to a user-supplied buffer in which the message data is to be received.
- `tsm_dlen` is set to the length of the receiving buffer.
- `tsm_flags` is set to `TET_SMRCVMSG`.

If the call to `tet_remsync()` is successful, then on return the API modifies members of the `tet_synmsg` structure on the receiving systems(s) as follows:

- Up to `tsm_dlen` bytes of sync message data are copied to the receiving buffer pointed to by `tsm_data`.
- `tsm_dlen` is set to the number of bytes of sync message data actually copied.
- `tsm_sysid` is set to the system ID of the system that sent the data, or to −1 if there is no message data associated with the sync point specified by `syncptno`.
- If the API must truncate the message because the receiving buffer is not big enough, the `TET_SMTRUNC` bit is set in `tsm_flags`.

If more than one system tries to send sync message data for a particular sync point, the API performs the following operations:

1. Decide from which system to accept data and redesignate the other sending systems as receiving systems.
2. Process the redesignated systems as described above.
3. Clear the `TET_SMSNDMSG` bit and set the `TET_SMRCVMSG` bit in `tsm_flags` on the redesignated systems.
4. Set the `TET_SMDUP` bit in `tsm_flags` on all systems.

If a process tries to send a message which is larger than the maximum permitted message size (as defined by the value `TET_SMMSGMAX`), the API perform the following actions:

1. Truncate the message to the maximum size before accepting it.
2. Set the `TET_SMTRUNC` bit in `tsm_flags` on all systems.

In most cases when a call to `tet_remsync()` is unsuccessful, the values of members of the `tet_synmsg` structure are undefined when the call returns. However, if the only reason that a call to `tet_remsync()` is unsuccessful is that other systems specified a **no** sync vote, the `tet_synmsg` structure is processed in the normal way. This enables a process both to send message data and to specify a **no** vote in a single `tet_remsync()` call.

If a process running on a particular system calls `tet_remsync()` with a `msgp` of `NULL`, the API regards it as a receiving system but does not return any message data to it.

Return value

The call to `tet_remsync()` returns zero as soon as all the participating systems synchronise at least as far as the specified sync point without timing out.

The call to `tet_remsync()` returns –1 when one of the following conditions occur:

- More than `waittime` seconds elapse between synchronisation requests from participating systems.
- A related synchronisation request times out on one of the other participating systems.
- The user-supplied function in a test case on one of the other participating systems returns control to its TCM before synchronising.
- The sync point specified by `syncptno` has already occurred.
- A **yes** sync vote is specified in the call but another participating system specifies a **no** vote for this sync point.
- `sysnames` is `NULL` or `nsysname` specifies an empty system ID list.
- A system ID appears more than once in the array pointed to by `sysnames`.
- An invalid parameter is specified in the call.
- The API encounters a problem while processing the request.

When a call to `tet_remsync()` is unsuccessful, the API sets `tet_errno` to indicate the cause of the error before calling the sync error handling function specified by `tet_syncerr`.

Portability

`tet_remsync()` is only provided in Distributed TETware. It is not provided in TETware-Lite.

The API treats sync message data as opaque and does not perform byte-swapping or other processing when data is exchanged between machines with different architectures. So it is best only to send ASCII strings in messages that are to be exchanged between systems which might run on different machines.

Application notes

The values of user-defined sync point numbers must increase throughout the lifetime of an entire test case and not just during the lifetime of a particular test purpose function within the test case.

Since synchronisation with other systems is defined in terms of system IDs (rather than individual process IDs), it is the responsibility of the test suite author to ensure that only one process running on a particular (logical) system calls `tet_remsync()` at one time. The results are undefined if processes running on the same system make overlapping `tet_remsync()` calls.

If a multi-threaded test case makes overlapping calls to `tet_remsync()` from more than one thread at once, one thread will be blocked by the API until the call in the other thread completes. Then the call in the blocked thread will fail with an `ER_DONE` error.

8.12.3 `tet_sync()` and `tet_msync()`

Synopsis

```
int tet_sync(long syncptno, int *sysnames, int waittime);

int tet_msync(long syncptno, int *sysnames, int waittime,
        struct tet_synmsg *msgp);
```

Description

`tet_sync()` and `tet_msync()` are provided for backward compatibility with previous DTET implementations and their use is deprecated in TETware. It is possible that support for these functions may be removed from a future TETware release. Test case authors should use `tet_remsync()` when writing new test cases.

In TETware `tet_sync()` and `tet_msync()` are implemented by calling `tet_remsync()`. When `sysnames` is non-NULL, it points to a zero-terminated list of system IDs. If either function is called with a NULL `sysnames` parameter, a default system list containing only system ID zero is used; otherwise, the zero-terminated system list pointed to by the `sysnames` parameter is used. A pointer to the resulting system list and the number of systems in the list (including the terminating zero) are passed to the underlying `tet_remsync()` call.

When a call to `tet_sync()` results in a call to `tet_remsync()`, a `vote` of `TET_SV_YES` and a `msgp` of NULL are used. Likewise, when a call to `tet_msync()` results in a call to `tet_remsync()`, a `vote` of `TET_SV_YES` is used.

When calls to `tet_sync()` or `tet_msync()` are unsuccessful, the API places an entry in the journal file indicating the cause of the failure. If the call is unsuccessful because one or more of the participating systems fails to synchronise, or the related process times out or terminates before the specified sync point occurs, a call is made to the sync error handling function specified by `tet_syncerr`. This variable is initialised with the address of a function which prints messages similar to those printed by the API in previous DTET implementations.

Return value

The return value of `tet_sync()` and `tet_msync()` is the same as that of the underlying `tet_remsync()` call.

Portability

`tet_sync()` and `tet_msync()` are only provided in Distributed TETware. They are not provided in TETware-Lite.

8.12.4 Control over sync error reporting

Synopsis

```
void (*tet_syncerr)(long syncptno, struct tet_syncstat *statp, int nstat);
void tet_syncreport(long syncptno, struct tet_syncstat *statp, int nstat);
```

Description

If a call to `tet_remsync()` is unsuccessful, the API calls the sync error handling function pointed to by the global variable `tet_syncerr` before the `tet_remsync()` call returns.

When `(*tet_syncerr)()` is called by the API, `syncptno` contains the number of the sync point that has failed, `statp` points to the first in an array of structures, each of which describes the sync status of each of the other systems participating in the event and `nstat` specifies the number of structures in the list.

The sync status structure is defined as follows:

```
struct tet_syncstat {
        int tsy_sysid;  /* system ID */
        int tsy_state;  /* sync state */
};
```

Possible values for the `tsy_state` member of this structure are as follows:

Symbolic constant	Meaning
TET_SS_NOTSYNCED	sync request not received
TET_SS_SYNCYES	system voted YES
TET_SS_SYNCNO	system voted NO
TET_SS_TIMEDOUT	system timed out
TET_SS_DEAD	process exited

The global variable `tet_errno` is set to indicate the cause of the error before `(*tet_syncerr)()` is called.

`tet_syncerr` is initialised to point to the API's default sync error reporting function `tet_syncreport()`, but may be changed by the test suite author to point to a user-supplied sync error handling function.

8.13 Remote system information

8.13.1 Introduction

These functions are provided in Distributed TETware to enable a test purpose to retrieve information about remote systems.

8.13.2 `tet_remgetlist()`

Synopsis

```
int tet_remgetlist(int **sysnames);
```

Description

A call to `tet_remgetlist()` from a process which is part of a distributed test case returns the number of other systems which are participating in the test case. In addition, a pointer to a zero-terminated array containing the names of the other systems is returned indirectly through `*sysnames`.

A call to `tet_remgetlist()` from a process which is not part of a distributed test case returns zero.

Return value

This function returns the values described above.

Portability

In TETware-Lite a call to `tet_remgetlist()` always returns zero and a pointer to a single zero-value system ID is returned indirectly through `*sysnames`.

8.13.3 `tet_remgetsys()`

Synopsis

```
int tet_remgetsys(void);
```

Description

A call to `tet_remgetsys()` returns the system ID of the system on which the calling process is executing.

Return value

This function returns the value described above.

Portability

In TETware-Lite a call to `tet_remgetsys()` always returns zero.

8.13.4 `tet_getsysbyid()`

Synopsis

```
int tet_getsysbyid(int sysid, struct tet_sysent *sysp);
```

Description

The `tet_getsysbyid()` function enables a test case to access information contained in the system definition file. If an entry for the system specified by `sysid` can be found in the file, information from the entry is placed in the user-supplied `tet_sysent` structure pointed to by `sysp`.

This function enables part of a distributed test case to determine the host (or node) names of other systems participating in the test.

The `tet_sysent` structure contains the following members:

```
struct tet_sysent {
        int ts_sysid;                   /* TETware system ID */
        char ts_name[TET_SNAMELEN];     /* system's host name */
};
```

Refer to the section entitled ‘‘System definitions’’ elsewhere in this guide for details of the system definition file.

Return value

A successful call to `tet_getsysbyid()` returns zero. If a call to `tet_getsysbyid()` is unsuccessful, –1 is returned and `tet_errno` is set to indicate the cause of the error.

Portability

This function is not provided in TETware-Lite.

8.13.5 `tet_remtime()`

Synopsis

```
int tet_remtime(int sysid, time_t *tp);
```

Description

A call to `tet_remtime()` obtains the system time on the system specified by `sysid` and returns it indirectly through `*tp`.

When `sysid` specifies the system ID of the calling process, the time is obtained by using an appropriate system call. However, when `sysid` specifies a different system ID, the time is obtained from an instance of TCCD that is running on the specified system.

Return value

A successful call to `tet_remtime()` returns zero. If a call to `tet_remtime()` is unsuccessful, –1 is returned and `tet_errno` is set to indicate the cause of the error.

Portability

This function is not provided in TETware-Lite.

8.14 Remote process control

8.14.1 Introduction

In Distributed TETware these functions enable a part of a distributed test case running on one system to generate a remote process on another system.

Note: The use of these functions is deprecated; they are only supported in order to provide backward compatibility with previous DTET implementations. It is possible that support for these functions may be removed from a future TETware release.

If it is necessary for one part of a distributed test case to start a process on a remote system, it is recommended that the test case should instead be structured so that the process is started by the part of the test case which is running on that system. If necessary, the two systems can make calls to `tet_remsync()` in order to ensure that the process is executed and waited for at the correct time.

If it is necessary for a non-distributed test case to start a process on a remote system, it is strongly recommended that the test case should instead be structured as a distributed test case; when this is done the method mentioned in the previous paragraph may be used.

8.14.2 `tet_remexec()`

Synopsis

```
int tet_remexec(int sysname, char *file, char *argv[]);
```

Description

A call to `tet_remexec()` starts a new process on the remote system specified by `sysid`. The calling process waits wait until the new process has been started and has synchronised with it.

`file` specifies the name of the file to be executed. The location of `file` is relative to the remote system's `TET_EXECUTE` directory if set, otherwise, it is relative to *tet-root* on the remote system. Since the request is performed by a server process, it is not necessary for a test case to call `tet_fork()` before calling `tet_remexec()`.

The `tet_remexec()` function passes the argument data as specified by `argv[]` to the process specified by `file`. The usage of `tet_remexec()` is similar to the ISO 9945–1 `execv()` function.

Note that the environment is not passed in a `tet_remexec()` call because it is not expected that there will be any correlation of the environment information on the remote machine to that of the calling process. Any data that is need by the remote process must be passed as an argument.

Return value

A successful call to `tet_remexec()` returns a positive value (the `remoteid`) which identifies the remote process within the context of the calling process. This value has no meaning outside the calling process. If the call to `tet_remexec()` fails, a value of –1 is returned and `tet_errno` is set to indicate the cause of the error.

In addition, an unsuccessful call to `tet_remexec()` may set `errno` to one of the following values:

`EINVAL` `sysname` does not refer to a known remote system.

`ENOEXEC` `file` cannot be executed on the remote system.

`ENOEXEC` Synchronisation with the remote process was not successful.

`EFAULT` The `file` or `argv` parameters are invalid.

`EIO` The connection with the remote system is broken.

Portability

This function is not provided in TETware-Lite or in any of the thread-safe APIs.

Application notes

This function is only provided for backward compatibility with existing test cases.

The preferred method of launching a process on a remote system is to arrange for the test case part executing on that system to perform the operation instead. If it is necessary for the operation to be controlled from another system, this can be achieved by appropriate calls to `tet_remsync()` from the test case parts that are running on each system.

8.14.3 `tet_remwait()`

Synopsis

```
int tet_remwait(int remoteid, int waittime, int *statloc);
```

Description

A call to `tet_remwait()` waits for the termination of a remote process initiated by `tet_remexec()`.

`remoteid` specifies the remote process identifier returned from a previous successful call to `tet_remexec()`.

`waittime` specifies the maximum number of seconds that the `tet_remwait()` call should wait before returning. If `waittime` is greater than zero, a call to `tet_remwait()` will be successful if the remote process exits within the specified time. If `waittime` is zero, a call to `tet_remwait()` will return immediately whether or not it is successful. If `waittime` is negative, a call to `tet_remwait()` will wait indefinitely for the remote process to exit or until the request fails for some reason.

A successful call to `tet_remwait()` returns the exit status of the remote process in the location pointed to by `statloc`. The exit status value returned indirectly through `*statloc` uses a standard encoding that is independent of the type of remote system on which the process is executed or the encoding used to return exit status values on that system.

The system used by TETware to encode the exit status of a remote process returned indirectly through `*statloc` by `tet_remwait()` is the one that traditionally has been used on many UNIX systems, as follows:

- If the remote process terminated normally, bits 0 through 7 contain zero and bits 8 through 15 contain the low order 8 bits of the argument that the remote process passed to `exit()` (but see under the Portability heading below).

- If the remote process terminated due a signal, bits 0 through 6 contain the number of the signal that caused the process to terminate and bits 8 through 15 contain zero. In addition, bit 7 is set if receipt of the signal caused a core image to be produced on the remote system.
- If the remote process is in a stopped state, bits 0 through 7 contain the value 0177 and bits 8 through 15 contain the number of the signal which caused the process to stop.

Return value

A successful call to `tet_remwait()` returns zero. If the call to `tet_remwait()` times out or is unsuccessful for some other reason, −1 is returned and `tet_errno` is set to indicate the cause of the error.

In addition, an unsuccessful call to `tet_remwait()` may set `errno` to one of the following values:

`EINVAL` `remoteid` does not refer to a process initiated from a call to `tet_remexec()`.

`ECHILD` `remoteid` refers to a process which is has already been waited for by a successful call to `tet_remwait()`.

`EAGAIN` The number of seconds specified by `timeout` expires before the remote process terminates.

`EINTR` The call to `tet_wait()` is interrupted.

`EIO` The connection to the remote system is broken.

Portability

This function is not provided in TETware-Lite or in any of the thread-safe APIs.

As indicated above, the process exit status returned by `tet_remwait()` uses a standard encoding which may not be the same as the one used by any particular operating system. For this reason, test suite authors are reminded that it is not appropriate to use the macros defined in `<sys/wait.h>` on a POSIX-conforming system to decode this value.

If a signal value is encoded in a process exit status, it is the value of the signal on the remote system. Test suite authors are reminded that this value may not refer to the the same (or any) signal on the system on which `tet_remwait()` is called.

The indication that a core image has been produced is not specified by POSIX. Therefore the setting of bit 7 to indicate that a core image has been produced as a result of a signal is implementation-dependent.

On a Win32 system the range of values which may usefully be passed to `exit()` is greater than the useful range on UNIX systems. Therefore it is possible for a process to exit with a non-zero status value whose low-order 8 bits are zero. In order to enable such a status to be identified as non-zero after a call to `tet_remwait()`, the API returns a status value of 1 in cases where the value of the low-order 8 bits of a non-zero exit status value from a process on a Win32 system is itself zero. Therefore, if the exit status of a process running on a Win32 system is to be returned unaltered by a call to `tet_remwait()`, its value should be in the range 0 to 127.

The C runtime support library on the Win32 system does not support the concept of a stopped process, encode the receipt of a signal in a process's exit status or generate a core image when a signal is raised in a process. Therefore, these indications are not available when a call to `tet_remwait()` returns an exit status from a remote process invoked on a Win32 system.

When a remote process running on a Win32 system is terminated by a call to `tet_remkill()`, a call to `tet_remwait()` returns a process exit status of 3. This is the same value as that generated by the C runtime support library when the default action is taken after a signal is generated by means of a call to `raise()`.

Application notes

This function is only provided for backward compatibility with existing test cases.

The preferred method of performing remote process control operations is described in the note which accompanies the description of the deprecated `tet_remexec()` function.

8.14.4 `tet_remkill()`

Synopsis

```
int tet_remkill(int remoteid);
```

Description

A call to `tet_remkill()` instructs the TCCD server which controls the remote process designated by `remoteid` to terminate the process. `remoteid` refers to a process started by a previous call to `tet_remexec()`.

On UNIX systems TCCD terminates the process by sending a `SIGTERM` signal; therefore the process is not terminated if this signal is being blocked or ignored.

A call to `tet_remkill()` returns immediately without awaiting confirmation that the remote process has terminated. (This information can be obtained from a subsequent call to `tet_remwait()` if required.)

Return value

A successful call to `tet_remkill()` returns zero. If the call to `tet_remkill()` is unsuccessful, –1 is returned and `tet_errno` is set to indicate the cause of the error.

In addition, an unsuccessful call to `tet_remkill()` may set `errno` to one of the following values:

`EINVAL` `remoteid` does not refer to a process initiated from a call to `tet_remexec()`.

`EIO` The connection to the remote system is broken.

Portability

This function is not provided in TETware-Lite or in any of the thread-safe APIs.

Application notes

This function is only provided for backward compatibility with existing test cases.

The preferred method of performing remote process control operations is described in the note which accompanies the description of the deprecated `tet_remexec()` function.

Test case authors are discouraged from using `tet_remkill()` to terminate a process which is running on a Win32 system. The reasons for this are discussed in the appendix entitled ‘‘Implementation notes for TETware on Win32 systems’’ in the TETware User Guide.

9. The C++ API

9.1 Introduction

This chapter describes the TETware C++ API. Different versions of the C++ API are supplied with TETware-Lite and Distributed TETware. The Distributed version of the C++ API may be used when writing both distributed and non-distributed test cases, whereas the Lite version of the C++ API may only be used when writing non-distributed test cases. The types of test case supported by each API version corresponds to the types of test case which may be processed by the TCC included with each TETware version.

On UNIX systems, this API has been designed to work with the USL C++ compiler release 3 or later, and with GNU `g++` release 2.4.5 or later.

On Win32 systems, this API is known to work with the Microsoft Visual C++ compiler.

9.2 C++ language binding

On UNIX systems, test cases written to this language binding attach themselves to it through the following files:

- *tet-root*`/lib/tet3/libapi.a` contains the support routines for test purposes. This is the same library as is provided with the C API.
- *tet-root*`/lib/tet3/Ctcm.o` contains the TCM. This file contains the routine `main()` and associated support routines for the sequencing and control of invocable components and test purposes.
- *tet-root*`/lib/tet3/Ctcmchild.o` contains the child process controller. This file contains a `main()` routine which can be used by test suites when building processes which test purposes will launch using the `tet_exec()` and `tet_spawn()` interfaces.
- *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. This file is the one that is provided with the C API; however, when this file is processed by a C++ compiler, its contents are made visible within an `extern "C"` block.

The names of these files are similar on Win32 systems; the differences are that object files (`.o` files) instead have a `.obj` suffix and library files (`.a` files) instead have a `.lib` suffix.

A test suite should access each of these files by means of its build tool, in a way which is appropriate given the available Software Generation System.

A thread-safe version of the C++ API is supplied in addition to the standard (that is: single-threaded) version. Distinct versions of the thread-safe C++ API are supplied with Distributed TETware and TETware-Lite. Differences between the standard and thread-safe APIs are described in the chapter entitled ‘‘The Thread-safe C and C++ APIs’’ elsewhere in this guide.

9.3 Using the C++ language binding

This API may be considered to be a ‘‘lightweight’’ binding, in that only a small part of it is built using a C++ compiler. This language binding uses the API library that is supplied with the C language binding. Details of all the functions and interfaces provided by this API library are presented in the chapter entitled ‘‘The C API’’ elsewhere in this guide.

In Distributed TETware the C++ API does not provide support for remote executed processes.

In the C and C++ language bindings, the TCM references variables that you must define in your test code. When you write a test case that uses the C++ language binding, you must enclose the definitions of these variables in an `extern "C"` code block, thus:

```
extern "C" {
        struct tet_testlist tet_testlist[] = {
                ...
        };

        void (*tet_startup)() = ...

        void (*tet_cleanup)() = ...
}
```

in order to enable the linker to resolve references made to these variables from the TCM code.

10. The Thread-safe C and C++ APIs

10.1 Introduction

TETware provides thread-safe versions of the C and C++ APIs in addition to the standard (single-threaded) API versions. The thread-safe APIs are provided in both TETware-Lite and Distributed TETware.

On UNIX systems each API can be built to support either POSIX threads or ‘‘UNIX International threads’’ (UI threads), but not both at the same time. On systems which support both types of threads, you should build TETware to support the type of threads that you wish to use in test cases.

When you use a thread-safe API on a UNIX system, you must compile all application source code that uses the API (including test suite libraries, for example) with either `–DTET_THREADS` when using UI threads, or `–DTET_POSIX_THREADS` when using POSIX threads. You must specify these options in addition to any other compiler options that may be required when compiling and/or linking multi-threaded programs.

On Win32 systems each API is built for use with the multi-threaded DLL version of the C runtime support library (`MSVCRT.LIB`). Note that the use of a thread-safe API in conjunction with the static version of the C runtime support library (`LIBCMT.LIB`) is not supported on Win32 systems.

When you use a thread-safe API on a Win32 system in conjunction with the defined build environment,[17] you must compile all application source code that uses the API with `–MD` and `–DTET_THREADS`. You must also use the `–MD` compiler option when linking a test case which uses a thread-safe API; when this option is used, the compiler instructs the linker to link with the correct version of the C runtime support library.

All of the standard interfaces are available in the thread-safe APIs with the exception of the deprecated `tet_remexec()`, `tet_remwait()` and `tet_remkill()` API functions.

10.2 C language binding

On UNIX systems, applications written to the thread-safe C language binding attach themselves to it through the following files:

- *tet-root*`/lib/tet3/libthrapi.a` is the thread-safe version of the API library.
- *tet-root*`/lib/tet3/thrtcm.o` is the thread-safe version of the TCM.
- *tet-root*`/lib/tet3/thrtcmchild.o` is the thread-safe equivalent of `tcmchild.o`.
- *tet-root*`/inc/tet3/tet_api.h` is the same file as used in the standard API.

The names of these files are similar on Win32 systems; the differences are that object files (`.o` files) instead have a `.obj` suffix and library files (`.a` files) instead have a `.lib` suffix.

17. That is: Microsoft Visual C++ and the MKS toolkit.

On UNIX systems, the extra threads-related contents of `tet_api.h` are made visible by compiling applications with `TET_THREADS` or `TET_POSIX_THREADS` defined. On Win32 systems, the extra threads-related contents of this file are made visible by compiling applications with `TET_THREADS` defined.

10.3 C++ language binding

On UNIX systems, applications written to the thread-safe C++ language binding attach themselves to it through the following files:

- *tet-root*`/lib/tet3/libthrapi.a` is the same library as for the thread-safe C language binding.
- *tet-root*`/lib/tet3/Cthrtcm.o` is the C++ version of the thread-safe TCM.
- *tet-root*`/lib/tet3/Cthrtcmchild.o` is the C++ equivalent of `thrtcmchild.o`.
- *tet-root*`/inc/tet3/tet_api.h` is the same file as used in the thread-safe C language binding.

Again, the names of these files are similar on Win32 systems; the differences are that object files (`.o` files) instead have a `.obj` suffix and library files (`.a` files) instead have a `.lib` suffix.

All of the declarations in `tet_api.h` are placed within an `extern "C"` block when the file is compiled with a C++ compiler.

10.4 Functions that are specific to the Thread-safe APIs

10.4.1 Introduction

The following sections describe functions which are only provided in the thread-safe versions of the C and C++ APIs.

10.4.2 `tet_thr_create()` and `tet_pthread_create()`

These functions create a new thread in a test purpose on a UNIX system. Applications which use UI threads should call `tet_thr_create()` and applications which use POSIX threads should call `tet_pthread_create()`. These functions are not implemented on Win32 systems.

When one of these functions is used to create a new thread, the API stores information about the newly-created thread in order to enable the TCM to perform appropriate actions when the test purpose returns control to the TCM, or when an unexpected signal occurs.

The syntax of these functions is as follows:

```
int tet_thr_create(void *stack_base, size_t stack_size,
        void *(*start_routine)(void *), void *arg,
        long flags, thread_t *new_thread, int waittime);

int tet_pthread_create(pthread_t *new_thread, pthread_attr_t *attr,
        void *(*start_routine)(void *), void *arg, int waittime);
```

The arguments and return value are the same as those for the `thr_create()` and `pthread_create()` functions respectively, except for the addition of the `waittime` argument.

When one of these functions is used to create a thread that is not detached, the `waittime` argument determines the action to be taken when the main thread returns control to the TCM. If a positive `waittime` is specified, the TCM waits at least `waittime` seconds for the newly-created thread to exit after the main thread returns. If the newly-created thread is still running at the end of that time, it is terminated by the TCM. If a zero or negative `waittime` is specified, the TCM does not wait for the newly-created thread to exit when the main thread returns. Instead, if the newly-created thread is still running at that time, it is terminated immediately by the TCM. The method used by the TCM to terminate a thread is described in the section entitled ''Clean-up of left-over threads on UNIX systems'' later in this chapter.

The purpose of this wait time is to allow other threads a grace period in which to exit in the event of an abnormal return from the main thread. Normally, all non-main threads should be waited for by calls to `thr_join()` or `pthread_join()` in the test case.

Unlike other API calls, `tet_thr_create()` and `tet_pthread_create()` do not set `tet_errno` if the call fails.

If either of these functions are used to create a detached thread, the API does not store any information about the new thread and the `waittime` argument is ignored. It is the responsibility of the test suite author to ensure that the detached thread either terminates before the main thread returns, or that it cannot interfere with the operation of later test purposes in the test case. Since an unexpected signal can cause the main thread to skip to the next test purpose, it is recommended that detached threads are only created in child processes (where unexpected signals are not caught by the TCM).

Unexpected results may occur if an application creates a new thread on a UNIX system other than by using these functions.

10.4.3 `tet_beginthreadex()`

This function creates a new thread in a test purpose on a Win32 system. It is not implemented on UNIX systems.

When this function is used to create a new thread, the API stores information about the newly-created thread in order to enable the TCM to perform appropriate actions when the test purpose returns control to the TCM.

The syntax of this function is as follows:

```
unsigned long tet_beginthreadex(void *security, unsigned stack_size,
        unsigned (*start_address)(void *), void *arglist,
        unsigned initflag, unsigned *thrdaddr, int waittime);
```

The arguments and return value are the same as those for the `_beginthreadex()` function in the C runtime support library, except for the addition of the `waittime` argument.

When this function is used to create a thread, `waittime` argument determines the action to be taken when the main thread returns control to the TCM. If a positive `waittime` is specified, the TCM waits at least `waittime` seconds for the newly-created thread to exit after the main thread returns. If the newly-created thread is still running at the end of that time, the TCM aborts the test case. If a zero `waittime` is specified, the TCM waits indefinitely for the newly-created thread to exit after the main thread returns. If a negative `waittime` is specified, the TCM does not wait for the newly-created thread to exit when the main thread returns. Instead, if the newly-created thread is still running at that time, the TCM aborts the test case.

The purpose of this wait time is to allow other threads a grace period in which to exit in the event of an abnormal return from the main thread.

Normally, all non-main threads should be waited for by calls to `WaitForSingleObject()` in the application.

Unlike other API calls, `tet_beginthreadex()` does not set `tet_errno` if the call fails.

Unexpected results may occur if an application creates a new thread on a Win32 system other than by using this function.

The following points should be noted when using the `tet_beginthreadex()` function:

1. The `start_address` argument must point to a function that uses the `__stdcall` calling convention. That is: the function must be defined using:

```
unsigned int __stdcall function-name(void *argument)
{
        …
}
```

2. A test case that uses one of the TETware C APIs must be linked with one of the C runtime support libraries. Therefore a thread that is created by a call to `tet_beginthreadex()` should exit either by returning from the start function or by calling `_endthreadex()`. A memory leak will occur in the C runtime support library if a thread exits by calling `ExitThread()` directly.
3. On Win32 systems the value returned by `tet_beginthreadex()` is actually a `HANDLE`, so it should be closed by a call to `CloseHandle()` when no longer required.

10.4.4 `tet_fork1()`

This function creates a child process containing only the thread of the calling process, but otherwise behaves in the same as does `tet_fork()`. As with `tet_fork()`, this function is only implemented on UNIX systems.

The syntax of this function is as follows:

```
int tet_fork1(void (*childproc)(void), void (*parentproc)(void),
        int waittime, int validresults);
```

The arguments and return value of this function are the same as those of `tet_fork()`.

Applications must safeguard calls to `tet_fork1()` in the same way as for calls to `fork1()`. For example, if the child needs to obtain resources such as mutexes, then the calling thread must obtain all the resources before making the call, in order to ensure they are not held by a non-existent thread in the child process. The `tet_fork1()` function does this for all such resources used internally by the API.

Unexpected results may occur if an application creates a new process other than by using `tet_fork()`, `tet_fork1()` or `tet_spawn()`.

10.5 Unavailable interfaces

The deprecated `tet_remexec()`, `tet_remwait()` and `tet_remkill()` functions are not available in the thread-safe APIs. In TETware these functions are provided for backward compatibility with dTET2 and should not be used in new test cases. When a distributed test case needs to execute a new process on a remote system, it is recommended that the new process should instead be started by the test case part that is executing on that system. If necessary, user-defined synchronisation points can be used to ensure that the new process is executed at the proper time.

10.6 API differences

10.6.1 Introduction

The following sections describe differences between the standard and thread-safe APIs, for the interfaces that are common to both versions.

10.6.2 Thread-specific data

The values of `tet_errno` and `tet_child` are thread-specific in the thread-safe API. They must be accessed by using the definitions provided in `tet_api.h`, and not simply by an `extern` declaration.

10.6.3 Block and sequence numbers

The thread-safe API has per-thread block and sequence numbers. When a new thread is created a new block number is assigned to both the new thread and to the calling thread. It is recommended that a call to `tet_setblock()` should be made after each thread is waited for.

The thread-safe version of `tet_setcontext()` does not reset the block and sequence numbers, because another thread might already have a current block number of 1. However, calls to `tet_fork1()` and `tet_spawn()` **do** reset the block and sequence numbers in the child, as when these functions are called there is only one thread in the new process.

The new block number set by `tet_setblock()` is one greater than the number set by the previous `tet_setblock()` call in **any** thread, not just the previous call in the current thread. The same applies to the `tet_setblock()` calls that are made internally by the API; for example: in the parent process in `tet_fork()` and in processes executed with `tet_exec()` on UNIX systems.

10.6.4 `tet_spawn()`

On a UNIX system, the thread-safe version of `tet_spawn()` uses `fork1()` to create the child process. Therefore the same considerations regarding resources such as mutexes apply as described for `tet_fork1()` above.

10.6.5 `tet_fork()`

The method used to time out child processes in the thread-safe version of `tet_fork()` on a UNIX system does not use a `SIGALRM` signal. This is done so as not to interfere with the use of `SIGALRM` in other threads.

10.7 TCM differences

10.7.1 Introduction

The following sections describe differences between the standard and thread-safe TCMs.

10.7.2 Clean-up of left-over threads on UNIX systems

Each time the main thread returns to the TCM, any other threads that remain are cleaned up before the TCM continues. Normally this is done after the following functions return:

- The `(*tet_startup)()` and `(*tet_cleanup)()` functions.
- Each test purpose function.
- The functions called from `tet_fork()` or `tet_fork1()` in the child process.
- `tet_main()`

It can also be done under abnormal conditions; for example: before skipping to the next test purpose on receipt of an unexpected signal.

The TCM terminates a thread which does not exit within the grace time specified in the `tet_thr_create()` or `tet_pthread_create()` call when the thread was created. The method used to terminate such threads is to arrange for the thread to execute a handler for the `SIGABRT` signal which then calls `thr_exit()` (or `pthread_exit()`). If the thread still does not terminate (for example: because it is blocking the `SIGABRT` signal), then the TCM aborts the test case.[18]

In order to prevent a thread being ‘‘cleaned up’’ while it holds a resource such as a mutex, applications should block the `SIGABRT` signal during the time these resources are held by a non-main thread. The grace time specified when a thread is created should be longer than any period for which the `SIGABRT` signal is to be blocked.

The special `SIGABRT` handler is only installed for long enough to send the signal to the target thread; however, this represents a small time window where the behaviour of other threads with respect to `SIGABRT` may not be as expected. The handler attempts to perform the expected action if this should occur (by calling the old handler function if there was one or by calling `abort()` if the old signal action was `SIG_DFL`).

When the TCM cleans up a thread after receiving a signal, it terminates the thread immediately instead of waiting for the grace time that was specified when the thread was created.

10.7.3 Dealing with left-over threads on Win32 systems

Each time the main thread returns to the TCM, it waits for any other threads that remain before continuing. This is done after the following functions return:

- The `(*tet_startup)()` and `(*tet_cleanup)()` functions.

18. Or the process, in the case of a child process.

- Each test purpose function.
- `tet_main()`

There is no safe way for the TCM forcibly to terminate a thread on a Win32 system. If a thread does not exit within the grace time specified in the `tet_beginthreadex()` call, the TCM reports a fatal error after it has finished waiting for all the threads, then aborts the test case.[19]

10.7.4 Signal handling

On UNIX systems, unexpected signals are managed in the thread-safe TCM in much the same way as in the standard TCM. Signal handlers are installed by the main thread before the start of each test purpose. The TCM does not make use of `sigwait()`, as this could interfere with the use of signals in the test purpose.

When an unexpected signal is caught by the main thread on a UNIX system, the signal handler cleans up any other threads as described in the previous section before taking the normal action as in the standard API. If an unexpected signal is caught by a non-main thread, the signal handler will forward the signal to the main thread and then cause the calling thread to exit.

The TCM does not attempt to manage unexpected signals on Win32 systems.

10.8 Synchronisation requests in multi-threaded test cases

In Distributed TETware it is possible for parts of a distributed test case to synchronise with each other at user-defined points during execution. Since synchronisation is defined in terms of **systems** and not processes, only one process on a particular system may represent that system in a particular synchronisation event.

When the thread-safe APIs are used, it is not possible for two threads in the same process to participate in any sync event at the same time. This restriction is enforced in the API by the use of mutexes (on UNIX systems) or critical section objects (on Win32 systems). If two threads in the same process call one of the synchronisation functions at the same time, one call will be blocked until the other call has completed. Therefore, if both of the calls refer to the same sync event by specifying the same system list and (non-zero) sync point number, one of the calls will block until after the event occurs. As a consequence, when the blocked call finally returns, it will probably fail with an `ER_DONE` error.

19. Or the process, in the case of a child process.

11. The Shell and Korn Shell APIs

11.1 Introduction

This chapter describes the Shell and Korn Shell APIs which are supplied with both TETware-Lite and Distributed TETware. These APIs may be used to write non-distributed test cases. There is no support for distributed test cases provided by these APIs.

The Shell API is provided for use by test cases written in the language that is defined for the `sh` command interpreter in the *X/Open Portability Guide Issue 3 Volume 1*. In addition, the interfaces described here are implemented in the TETware Korn Shell API. The Korn Shell API is provided for use by test cases written in the language that is used by the `ksh` command interpreter. Both of these APIs make use of commands which are available on POSIX-conforming systems. Except where noted, the descriptions that follow apply equally to each of these APIs.

When TETware is used on a Win32 system, these APIs are designed to be used with commands provided in the MKS Toolkit. The MKS Shell, `sh`, is based on the Korn Shell and can be used in conjunction with either of the shell APIs. However, it is recommended that for reasons of efficiency you only use the Korn Shell API when writing shell language test cases for use on a Win32 system. Note that the names of each Shell language and Korn Shell language test case file must have a `.ksh` suffix on a Win32 system.

See the chapter entitled ‘‘Writing a Shell language API-conforming test suite’’ elsewhere in this guide for an example of how to write a Shell language based test suite.

11.2 Shell language binding

Support for the Shell language binding is provided through Shell language source files as follows:

- *tet-root*`/lib/xpg3sh/tcm.sh` contains the support routines for the sequencing and control of invocable components and test purposes (the Shell TCM).
- *tet-root*`/lib/xpg3sh/tetapi.sh` contains the support routines for use by test purposes (the Shell API).

These files must be ‘‘sourced’’ into an executable shell script file by using the `.` (dot) shell built-in command. Sourcing the Shell TCM also automatically sources the Shell API.

11.3 Korn Shell language binding

Support for the Korn Shell language binding is provided through Korn Shell language source files as follows:

- *tet-root*`/lib/ksh/tcm.ksh` contains the support routines for the sequencing and control of invocable components and test purposes (the Korn Shell TCM).
- *tet-root*`/lib/ksh/tetapi.ksh` contains the support routines for use by test purposes (the Korn Shell API).

These files must be ‘‘sourced’’ into an executable shell script file by using the `.` (dot) shell built-in command. Sourcing the Korn Shell TCM also automatically sources the Korn Shell API.

11.4 TCC dependencies

Test cases built with theses APIs may either be executed stand-alone or under the control of either TCC version.

11.5 Test case structure and management

11.5.1 Introduction

These variables are used when test cases are initialised and cleaned up, and in selecting invocable components and test purposes to execute.

11.5.2 `iclist`, `ic`*n*, `tet_startup` and `tet_cleanup`

Synopsis

`iclist="`*blank-separated list of invocable component names*`"`

`ic1="`*blank-separated list of test purpose names*`"`
`ic2="`*blank-separated list of test purpose names*`"`
…

`tet_startup=`*startup-procedure*

`tet_cleanup=`*cleanup-procedure*

Description

The start up routine, clean up routine and each of the test purposes should be implemented by the test author as either shell functions or as separate executable shell scripts. These shell functions or scripts will be called by the shell TCM according to the requested set of invocable components. The `iclist` definition is provided by the test suite author, and contains a blank separated list of invocable component names. These invocable component names are formed by prefixing each invocable component number with the letters `ic`.

When an invocable component is requested by the TCC, the shell TCM executes each name in the associated list of test purposes. Each of the test purposes is executed in a subshell with the appropriate signal handling being applied to the subshell.

The TCM does not perform any explicit error checking on the contents of a list of test purposes. It is the responsibility of the test author to ensure that the names reference shell functions or executable shell scripts.

The shell variables `tet_startup` and `tet_cleanup` are set to refer to the shell function or script to be used for test case specific start up and clean up procedures, respectively. The start up procedure is executed before the first requested invocable component and the clean up procedure is executed on completion of the last requested invocable component. These routines are executed irrespective of which invocable components are requested. If no start up or clean up is required, the `tet_startup` and `tet_cleanup` variables may be left unset or set to an empty string.

The TCM and API are provided as shell scripts which must be sourced by the test suite author immediately after the `tet_startup`, `tet_cleanup` and `iclist` variables, each of the `ic`*n* variables, and any shell functions used by the test case have been defined. The shell script is sourced by use of the `.` (dot) shell built-in command. Note that if a test purpose is written as a separate shell script, that script must source the shell API in order to have access to API support

routines.

11.5.3 `tet_thistest`

Synopsis

```
$tet_thistest
```

Description

The `tet_thistest` shell variable contains the **name** of the currently executing test purpose, as specified in the `ic`*n* variable.

11.6 Insulating from the test environment

The following configuration variables are used by the shell TCM to help determine which events should be handled for the test case, and which should be passed through.

`TET_SIG_IGN` defines (by comma separated number) the set of signals that are to be ignored during test purpose execution. Any signal that is not set to be ignored or to be left with its current disposition (see `TET_SIG_LEAVE` below), will be caught when raised and the result of the test purpose will be set to `UNRESOLVED` because of the receipt of an unexpected signal. A test purpose may undertake its own signal handling as required for the execution of that test purpose; the disposition of signals will be reset after the test purpose has completed. The API needs to know how many signals the implementation supports in order to set up trap statements for these signals.

`TET_SIG_LEAVE` defines (by number) the set of signals that are to be left unchanged during test execution. In most cases this will mean that the signal takes its default action. However, the user can change the disposition of the signal (to ignore) before executing the TCC if this signal is to remain ignored during the execution of the test purposes.

The implementation does not allow a standard set of signals to be set to be ignored or left unchanged, as this may pervert test results.

11.7 Making journal entries

11.7.1 Introduction

These functions are provided for use by test cases when making entries in the execution results file.

11.7.2 `tet_setcontext` and `tet_setblock`

Synopsis

```
tet_setcontext
tet_setblock
```

Description

The `tet_setcontext` shell function changes the context of the calling process. When the current context is not equal to the value of `$$` (the shell builtin variable which contains the shell's process ID), a call to `tet_setcontext` sets the context to the value of `$$`. Otherwise, if the current context is already equal to `$$`, a call to `tet_setcontext` sets the current context to a new value. This behaviour enables a context to be established in a subshell with a different value to that established in a parent shell.[20]

The current context is stored in the environment variable `TET_CONTEXT` which is marked for export. This enables the context to be passed to subsequent processes by using this environment variable. The `tet_setcontext` function should be executed by any application which executes a background subshell and which wishes to write entries to the execution results file from both processes. The `tet_setcontext` function must be executed from the child process, not from the parent. Test suite authors should ensure that `tet_setcontext` is only called when it is necessary to change the context in a subshell. Gratuitous calls to this function should not be made.

The parent should call `tet_setblock` as appropriate to distinguish its output before, during and after execution of the child.

The `tet_setblock` shell function increments the current block ID. The value of the current block ID is reset to one at the start of every test purpose and after a call to `tet_setcontext` which altered the current context. The sequence ID of the next entry is set to one at the start of each new block. The current block ID is stored in the shell variable `TET_BLOCK` which is marked for export.

---

20. A subshell is a sequence of shell commands enclosed in parentheses, thus: `( ... )`.

    Note that the value of `$$` is the same in a subshell as it is in the parent shell. Thus it is not possible to use `$$` to determine the value of the process ID of a subshell.

11.7.3 `tet_infoline`

Synopsis

```
tet_infoline data ...
```

Description

The `tet_infoline` shell function outputs an information line to the execution results file. The sequence number is incremented by one after the line is output. If the current context and the current block ID have not been set, the call to `tet_infoline` causes the current context to be set using the value of the calling process ID and the current block ID to be set to one. Note that `tet_infoline` does not process backslash escapes like the shell `echo` built-in command. If more than one argument is passed to `tet_infoline`, each argument is separated from the next by a space character when the line is written to the execution results file.

11.7.4 `tet_result`

Synopsis

```
tet_result result
```

Description

The `tet_result` shell function sets the result code that will be output at the end of the test purpose. The `result` argument specifies the **name** of the result that is to be output. This result is output to the execution results file by the TCM upon test purpose completion. This ensures that all informational messages are written out before the test purpose result, and that there is one (and only one) result generated per test purpose.

If a test purpose does not call `tet_result`, the TCM generates a result of `NORESULT`. If more than one call to `tet_result` is made with different result codes, the TCM determines the final result code by use of precedence rules. The precedence order (highest first) is:

`FAIL`
`UNRESOLVED`, `UNINITIATED`
`NORESULT` (i.e., invalid result codes)
Test suite supplied codes
`UNSUPPORTED`, `UNTESTED`, `NOTINUSE`
`PASS`

Where two or more codes have the same precedence then all calls to `tet_result` with one of those codes are ignored except the first such call.

11.8 Canceling test purposes

11.8.1 Introduction

These functions are provided for use when cancelling test purposes.

11.8.2 `tet_delete`

Synopsis

```
tet_delete testname reason
```

Description

The shell function `tet_delete` marks the test purpose specified by `testname` as canceled. The TCM will output `reason` as the reason for cancellation on the information line that is generated whenever it attempts to execute this test purpose. The argument `testname` matches the name which is used to call this test purpose. If the requested `testname` does not match the name of a test purpose, no action is taken. If the requested `testname` is already marked as canceled the reason is changed to `reason` and the test purpose remains marked as canceled. If `reason` is an empty string then the requested `testname` is marked as active; this enables previously canceled test purposes to be re-activated.

11.8.3 `tet_reason`

Synopsis

```
tet_reason testname
```

Description

The shell function `tet_reason` prints the reason why the test purpose specified by `testname` has been canceled and returns a value of 0. The reason is printed on the standard output. If the test purpose specified by `testname` is not marked as canceled or does not match the name of a test purpose, no reason is printed and the function returns a value of 1.

11.9 Manipulating configuration variables

There is no explicit shell interface to support this functionality. The API ensures that the configuration information is available to the test purposes as shell variables marked `readonly`. Each of these shell variables can be accessed using the normal shell mechanisms.

11.10 Generation and execution of processes

There is no explicit shell interface to support this functionality. The API ensures that the configuration information and the `tet_thistest` shell variable are available as shell variables. The ability to use parentheses to generate a subshell environment enables these variables to be inherited when a subshell is generated. The only facilities that are not provided in the shell are the ability to timeout a subshell process and the examination of the exit code from the subshell. The shell provides facilities to accomplish these tasks in a relatively straightforward manner and this is considered to be an issue for the application programmer rather than for the API.

11.11 Executed process support

Shell scripts which are executed by a test case written to this API should source the shell API to include the necessary support routines using the **.** (dot) shell built-in command. Note that this will not provide TCM functions (like signal handling and test purpose sequencing). Executed processes which need this type of support should be test cases in their own right.

12. The Perl API

12.1 Introduction

This chapter describes the TETware Perl API. The Perl API requires the use of the `perl` utility and may be used on Win32 operating systems as well as on UNIX systems. On a Win32 system the name of a Perl test case must include a `.pl` suffix if it is to be recognised as such by the TETware TCC.

Non-distributed test cases written using this API may be run stand-alone or under the control of both the Distributed and Lite versions of the TETware TCC. The Perl API does not support distributed testing.

12.2 Description

In many respects the Perl language binding is similar to the Shell (`xpg3sh`) language binding. Test cases written to this language binding attach themselves to it through the following files:

- *tet-root*`/lib/perl/tcm.pl` contains the Test Case Manager.
- *tet-root*`/lib/perl/api.pl` contains the support routines for use by test purposes.

The Perl API is equivalent to the `posix_c` API provided in TET 1.10.

The following Perl calling conventions should be observed:

```
&tet'setcontext;
&tet'setblock;
&tet'infoline("text");
&tet'result("result-name");
&tet'delete("test-name"[, "reason"]);
deletion-reason = &tet'reason("test-name");
```

The default result code list is `PASS`, `FAIL`, `UNRESOLVED`, `NOTINUSE`, `UNSUPPORTED`, `UNTESTED`, `UNINITIATED` and `NORESULT`.

The usage of each call and variable is equivalent to the corresponding calls and variables in the Shell API.

Variable references should take the following forms:

```
@iclist=(ic1,ic2,...icn);
@ic1=("my_tp1");
@ic2=("my_tp2","my_tp3");
...

$tet'startup="my_startup_routine";
$tet'cleanup="my_cleanup_routine";

@tet'sig_leave_list=(...);
@tet'sig_ignore_list=(...);

$tet'thistest;
```

A Perl API-compliant program should adhere to the following structure:

```
set iclist, ICs, optional setup and cleanup routines
code for subroutines
require "$ENV{\"TET_ROOT\"}/lib/perl/tcm.pl";
```

Example test suites written in Perl that test the API are provided in the *tet-root*/contrib/suite and *tet-root*/contrib/api directories in the TETware distribution. In addition, a Perl demonstration test suite is provided in the *tet-root*/contrib/perldemo directory in the TETware distribution. Instructions for running the Perl demonstration test suite are presented in the section entitled ‘‘The Perl API demonstration’’ in the TETware User Guide.

13. Test reporting and journaling

13.1 Making journal entries

The TETware API journaling facility provides a mechanism for outputting data to the execution results file. The API ensures that each entry in this file is written atomically and that there is sequencing information applied to the entry (to allow `tcc` to reorder data that is produced from two or more concurrently executing processes started by a single test purpose). `tcc` ensures that simultaneous execution of test cases are isolated from one another.

In order to allow for the correct sequencing of information the following attributes are defined:

- System identifier.
- The current context.
- The current block.

The current block is a subdivision of the current context and provides a means of ensuring contiguity, after resequencing, of a block of data that needed to traverse several entries. The need to traverse several entries may be caused by the limitations on the atomicity imposed by the implementation, or may be purely a matter of convenience for the test suite author.

The current context is initialised during test case start-up and should be changed only after a new process is generated. This allows the author to choose whether a number of concurrently executing test purposes should have the same context or different ones.

The system identification is used to distinguish entries written from test case parts on the multiple systems participating in distributed test cases.

The current block is initialised to one by the start-up routines at the commencement of each test purpose. The test author can increment the current block at any point during the output of entries in order to distinguish one block of data from another. Each individual entry within a block will be sequenced starting at one. Use of the journaling support facilities allows data from concurrently executing test purposes to be ordered correctly by `tcc`.

13.1.1 Entries from the API

As mentioned above, the TCM handles the sequencing of test purposes as a part of executing invocable components. The sequencing mechanism outputs invocable component start and end information and test purpose start information to the execution results file. The test author is responsible for outputting test information and test results to the execution results file.

All of the data for an entry is transferred atomically to the execution results file. It is the responsibility of the test suite author to remain within the limitations imposed by the implementation for a single atomic write operation. TETware guarantees atomicity of writes up to 512 bytes.

If a test purpose executes another process that is built to the TETware API, and that executable is expected to generate journal messages, the test purpose must use the TETware API to communicate the current message context to the executed process.

13.1.2 Entries from test purposes

The API provides functionality for delivering informational messages and results from test purposes to the execution results file. These messages are in addition to those specified above, which are provided automatically by the TCM. The content of informational messages is limited only by the limit imposed upon the total length of a journal line. It is expected that test cases will use this mechanism to deliver special messages to the journal or for additional reporting sequences that can be analysed by test suite specific report treatment filters. Test purposes also deliver results to the execution results file. These results are checked by the API to ensure that they have been defined by TETware or by the test suite. In the event of an invalid result, the TCM delivers a message to the execution results file and sets the result to `NORESULT`. The result actions are also checked by the API. If a test purpose specifies a result for which the action is `Abort`, then the TCM will not process any more test purposes, call the user-supplied cleanup function (if one has been defined) and exit.

Note that if a test purpose neglects to generate a result via the API, the TCM will supply a result of `NORESULT` for that test purpose.

13.2 Journal files

13.2.1 Description

Result files are written by test cases, build tools and clean tools when run with output capture mode disabled. These result files are then transferred into the journal file by `tcc`. The format of lines in these files is identical.

13.2.2 Journal line parameters

The total length of a journal line must not exceed 512 bytes. Each journal line is made up of a message type, the parameters for that message, and a message area (the format of which is unconstrained). Each message may have zero or more parameters associated with it. These parameters (strings or integers represented by no more than ten decimal digits), are blank separated and contained between vertical bars. Possible parameters include:

- The TCC activity number (*activity*).

  This number is incremented each time an activity performed by `tcc`. Each build, execute or clean-up of a test case is considered an individual activity.

- The invocable component number (*ICnumber*).

- An invocable component count (*ICcount*).

  This is the number of invocable components executed in each test case (expected or actual as specified).

- The test purpose number (*TPnumber*).

  This number uniquely identifies the test purpose within a test case.

- The test purpose count (*TPcount*).

  This is the number of test purposes that make up an invocable component.

- The message context (*context*).

  This field represents the process that generated the journal line. It consists of a three

character system ID, followed by the process ID of the the process that initiated the entry.

- The message block number (*block*).

  This number is set to one at the start of each test purpose or new context, and is incremented each time the test purpose requests it. This number, along with the process identifier and message number (below) is used by `tcc` to order the data in an execution result file prior to transferring that file into the journal.

- The message sequence number (*sequence*).

  This number is set to one at the start of each block, and is incremented each time a message is written to the result file.

- The current time (*time*).

  Times are given using the notation *HH*`:`*MM*`:`*SS* with a 24 hour clock.

- The current date (*date*).

  Dates are given using the notation *YYYYMMDD*; for example, 19910610 for 10th June 1991.

- A test case name (*testcase*).

  This is the test case name as given in the scenario file.

- A path name (*pathname*).

  The full path name of a file.

- The `tcc` execution mode (*mode*); possible values for this parameter are as follows:

  `0` Build
  `1` Execute
  `2` Clean-up
  `3` Pseudo-mode value used when reporting distributed configuration variables

- A completion status (*status*).

  A non-negative value is the value returned to `tcc` by the operating system after execution of a test case or tool. Negative values are reserved for use by TETware. The following values may be used by `tcc` to indicate some problem when processing a test case:

Status value	Meaning
–1	The test case or tool could not be executed by `tcc`
–2	The test case or tool was timed out by `tcc`
–3	One or more locks could not be obtained by `tcc`
–4	`tcc` encountered some other error while processing the test case

13.2.3 Journal line descriptions

A description of each type of journal line that may be produced by TETware processes is presented in the appendix entitled ‘‘TETware journal lines’’ in the TETware User Guide.

13.3 Result file processing

13.3.1 Execution results from an API-conforming test case

When a test case using an API which does not support distributed testing[21] is executed (whether stand-alone or under the control of `tcc`), the API writes journal lines to an execution results file called `tet_xres` which resides in the test case execution directory.

When a test case using an API which supports distributed testing is executed by the Distributed version of `tcc`, the API sends journal lines to the Execution Results daemon (`tetxresd`) which writes them to an execution results file. `tetxresd` maintains a separate execution results file for use by each non-distributed test case and a single (or combined) execution results file for use by all parts of a distributed test case.

13.3.2 Processing results from a non API-conforming test case

When `tcc` executes a test case which does not use an API, it pretends that the test case consists of a single invocable component which contains a single test purpose. Before the test case is executed, `tcc` writes a TCM Start message, an IC Start message and a TP Start message to the journal. When the test case finishes execution, `tcc` writes a TP Result message and an IC End message to the journal. The result contained in the TP Result line is determined by the test case's exit status; zero status causes `PASS` to be reported and non-zero status causes `FAIL` to be reported.

Note that `tcc` does not perform automatic result generation when it executes a non API-conforming build or clean tool.

13.3.3 Processing results from a non-distributed API-conforming test case

When a test case run under the control of `tcc` finishes execution, `tcc` reads the execution results file (wherever the file is) and transfers its contents to the journal. When the Distributed version of `tcc` executes a non-distributed test case, it has to be aware of the possibility that the API might either write journal lines to the `tet_xres` file or send them to `tetxresd`. Therefore, when such a test case finishes execution, `tcc` first inspects the execution results file maintained by `tetxresd`. If the file contains at least one line, `tcc` uses this file. Otherwise, if the file is empty, `tcc` looks for a `tet_xres` file in the test case execution directory and uses that instead. When the Distributed version of `tcc` decides to use a `tet_xres` file produced by a non-distributed test case that has been executed on a remote system, it must first transfer the file to the local system before it can be used.

Once `tcc` has identified the location of an execution results file that has been generated by a non-distributed test case (by whatever means), it transfers lines from that file to the journal. When `tcc` performs this operation, it inspects the type of each line read from the execution results file and processes it as follows:

---

21. The C and C++ APIs in Distributed TETware support distributed testing. The other APIs in Distributed TETware and all the APIs in TETware-Lite do not support distributed testing.

1. While the line is not a TP Start line, it is simply copied to the journal.
2. When a TP Start line is found, it is transferred to the journal. Then subsequent lines up to a TP Result line[22] are ordered as follows:
   a. `tcc` inspects the type of the first un-transferred line in the range. If the line is not a Test Case Information Line, it is transferred to the journal and step (a) is repeated. If the line is a Test Case Information Line, it is transferred to the journal and the context and block numbers are remembered.
   b. `tcc` then inspects all the other un-transferred lines in the range and identifies lines with the same context and block numbers. These lines are transferred to the journal in order of ascending sequence number.
   c. When `tcc` reaches the end of the range, it returns to step (a). This process is repeated until all lines in the range are transferred.
3. Then `tcc` copies the TP Result line to the journal. If no TP Result line appears, `tcc` supplies one which contains a result of `NORESULT`.

These steps are repeated until the end of the execution results file is reached.

13.3.4 Processing results from a distributed API-conforming test case

When a distributed test case is run under the control of the Distributed version of `tcc`, parts of the test case which run on each participating system each send execution results lines to `tetxresd` for processing. The API ensures that only one part of the test case generates the TCM Start, IC Start, IC End and TP Start lines that must appear in the execution results file.

Each part of a distributed test purpose is expected to generate a Test Purpose Result. `tetxresd` arbitrates between all the partial results and generates a single consolidated result for each test purpose. If a test purpose part does not supply at least one result, `tetxresd` records a partial result of `NORESULT` on behalf of that system before performing the result arbitration.

When `tcc` copies the execution results file generated by a distributed test case to the journal, it does not reorder Test Purpose Information lines; instead, they are copied to the journal in the order in which they were received by `tetxresd`. Therefore, test case authors should ensure that, when two or more Test Purpose Information Lines from a particular process are required to appear in the journal without being separated by lines from another process, the lines are presented to the API using a function which instructs `tetxresd` to write all the lines to the execution results file in a single operation.

13.4 Support for user-supplied report writers

As indicated previously, TETware generates a journal file using a well-defined format. It is expected that test suite authors will provide a report writer which presents the information contained in the TETware journal in a format which is appropriate for the type of testing being undertaken.

---

22. Or another line type which indicates the end of the scope of the current test purpose, or end-of-file.

Each line in the journal file consists of three fields; each field is separated from the next by a `|` character. The value in the first field of each line indicates the type of the line. For convenience of test suite authors who wish to write a report writer using the C language, these values are defined in a header file which is supplied with the TETware distribution. The name of this file is `tet_jrnl.h` and it resides in the *tet-root*`/inc/tet3` directory.

14. Writing a C language API-conforming test suite

14.1 Introduction

This chapter describes a sample non-distributed test suite that conforms to TETware's C language binding of the API. The source code for the test suite can be found in the appendix entitled ''Example C language API test suite source files'' at the end of this guide. This test suite has been designed to run on a UNIX type of operating system.

This sample test suite is designed to illustrate how a non-distributed test suite can be structured under TETware, as well as how individual test cases and their test purposes relate to each other and to the API. The test suite has been deliberately kept simple and realistic. For example, one test purpose compares the returned error code against an expected error code of a failed system call, while another test purpose in the same test case checks the successful execution of the system call.

Small segments of code from the test suite appear in the following sections to help illustrate specific points. Refer to the appropriate section in the appendix entitled ''Example C language API test suite source files'' at the end of this guide to see the code in its entirety.

14.2 Defining a test suite

Test suites reside in subdirectories of *tet-root*. As explained in the chapter entitled ''Testing structure'' earlier in this guide, the name of the subdirectory and the test suite are the same.

The following figure shows the component files of the sample test suite, called `C-API`:

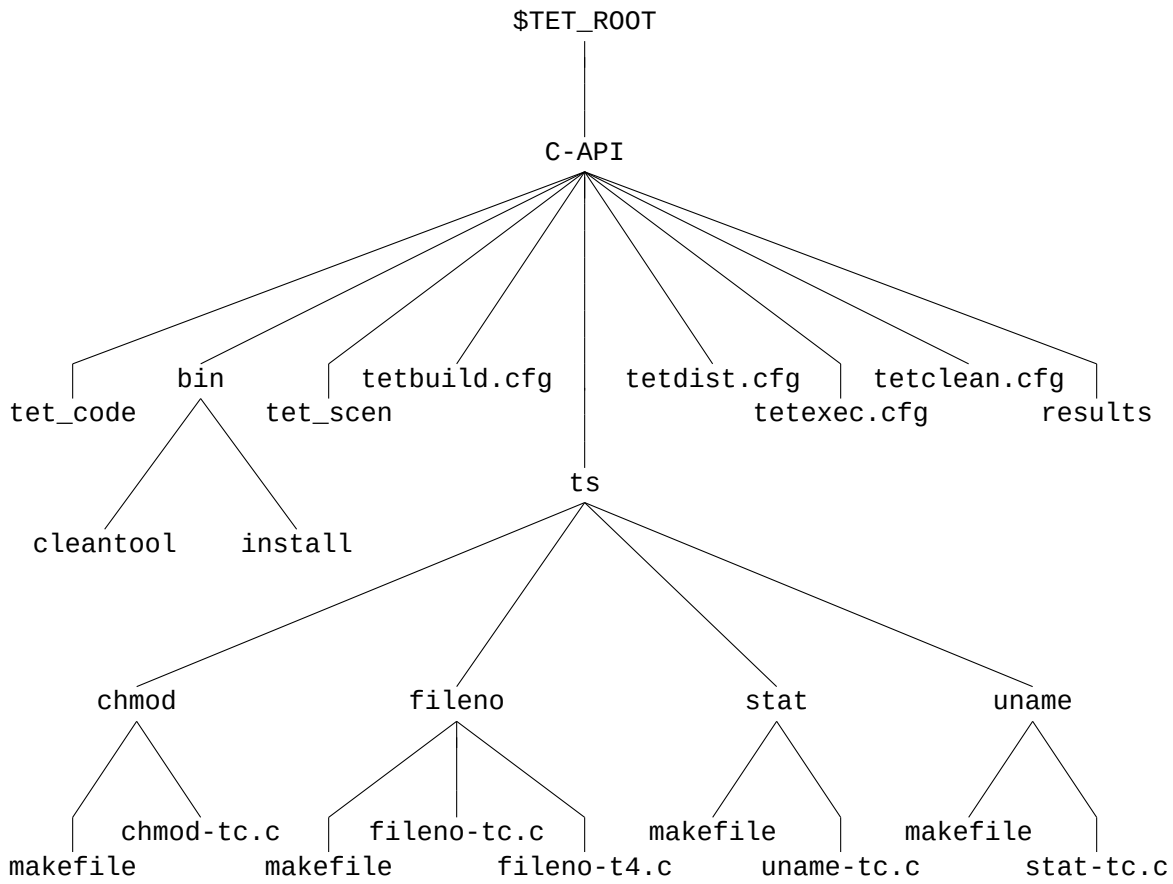

Figure 17. Directory structure for the example C language test suite

The make-up of this test suite is similar to the demonstration test suite as defined for the master system and contains the following files:

- An install script and clean tool in the `bin` directory.
- Configuration files for test build, execution, and cleanup.
- A control file, `tet_scen`.
- A result codes file, `tet_code`.
- Several test cases in a directory structure under the directory `ts`.
- A results directory.

If this test suite is run using TETware-Lite, the `tetdist.cfg` file is not required. If this test suite is run on the local system using Distributed TETware, a `systems` file is required. In addition, the `tetdist.cfg` file is required when this test suite is run on a remote system or when Distributed TETware is built to use XTI as the network transport.

The control file, `tet_scen`, lists the components of the test suite; and its contents determine the scenarios that can be used in running the test suite. The control file, `tet_scen`, for the `C-API` test suite contains the following lines:

```
#       chmod, fileno, stat, uname test suite.

all
        "Starting Full Test Suite"
        /ts/chmod/chmod-tc
        /ts/fileno/fileno-tc
        /ts/stat/stat-tc
        /ts/uname/uname-tc
        "Completed Full Test Suite"

chmod
        "Starting chmod Test Case"
        /ts/chmod/chmod-tc
        "Finished chmod Test Case"

fileno
        "Starting fileno Test Case"
        /ts/fileno/fileno-tc
        "Finished fileno Test Case"

stat
        "Starting stat Test Case"
        /ts/stat/stat-tc
        "Finished stat Test Case"

uname
        "Starting uname Test Case"
        /ts/uname/uname-tc
        "Finished uname Test Case"

# EOF
```

The control file lists five scenarios for the test suite: `all` (required), `chmod`, `fileno`, `stat` and `uname`. Since the test suite is composed of four test cases, one for the `chmod()` system call, one for the `fileno()` system call, one for the `stat()` system call, and one for the `uname()` system call, the control file has been written to allow each test case to be handled as a separate scenario, or for the whole test suite to be run at once with the `all` scenario.

The lines enclosed in double quotes (") are optional information lines that get passed into the journal file. The lines that begin with a slash or stroke character (/) name the executable test cases associated with each scenario. Note that, even though these lines begin with a slash character, the location of the test cases is interpreted relative to the local directory (the root directory for the test suite). In this instance, the test cases are in a subdirectory named `ts`.

The clean tool is used to remove unwanted files after the build of each test case. It is invoked in the source directory of the test case. In this case it is set to `exec make clean` to remove unwanted object files as defined in each `makefile`.

14.3 Defining common test case functions and variables

Since most test suites lend themselves to lots of code redundancy, making an effort to group together common functions and variables can greatly simplify the writing and debugging of a test suite. With the `C-API` test suite (which is very small), no common functions and variables other than the standard ones in `tetapi.h` were created.

One additional result code was invented, however, which would normally be defined in a test suite specific header file. But because it is only used within one test case in this very small test suite, it is instead defined within `uname-tc.c` as follows:

```
#undef TET_INSPECT      /* must undefine because TET_ is reserved prefix */
#define TET_INSPECT 33  /* this would normally be in a test suite header */
```

14.4 Initialising test cases

Every test case requires some minimum initialisation of functions and variables. The `fileno-tc` test case provides a good illustration of how this initialisation can be handled.

```
/* fileno-tc.c : test case for fileno() interface */

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

#include <tet_api.h>

extern char **environ;

static void cleanup();
static void tp1(), tp2(), tp3(), tp4(), ch4();

/* Initialise TCM data structures */
void (*tet_startup)() = NULL;
void (*tet_cleanup)() = cleanup;
struct tet_testlist tet_testlist[] = {
    { tp1, 1 },
    { tp2, 2 },
    { tp3, 3 },
    { tp4, 4 },
    { NULL, 0 }
};

/* Test Case Wide Declarations */
static char msg[256];                   /* buffer for info lines */
```

After the `#include` statements, several functions are declared. As described in the chapter entitled ‘‘The C API’’ earlier in this guide, TETware provides the option of naming both a startup and a cleanup function. The named startup function will be called before the first test purpose is executed; and the cleanup function will be called after all test purposes have been executed. In

this test case, only the cleanup function is named. The cleanup function `cleanup()` removes files created during the course of the test case.

The `stat-tc` test case includes a more substantial cleanup function, as well as a startup function. It requires that a file be created before the first test purpose, so this is handled by the startup function; this same file, as well as another file and a directory created during the tests, is then removed in the cleanup function. See the appendix entitled ‘‘Example C language API test suite source files’’ at the end of this guide for a complete code listing of the `stat-tc` test case.

The `fileno-tc` test case includes four test purposes, contained in the functions `tp1()`, `tp2()`, `tp3()` and `tp4()`. First the functions are declared (including an extra function which is a child process started by `tp4()`), as shown above. Then they are listed in the `tet_testlist` array with the invocable component to which they belong. In this case, each test purpose can be executed individually, so they are assigned to separate invocable components. If, say, `tp2()` depended on prior execution of `tp1()`, then they would be assigned the same IC number. After the array is set, any test case wide declarations are made. This commonly includes a buffer to use for constructing information lines to be output with `tet_infoline()`.

14.5 Controlling and recording test case execution results

Identifying and executing highly specific tests is central to any test case. Each test purpose in a test case typically targets one specific test that is loosely or strongly related to the other test purposes contained in the test case. The central purpose of each of these test purposes is to relay information about the execution of the test for the tester to examine later. This relaying of information can take the form of informational messages describing the test being executed, fatal or non-fatal errors that were encountered, and specific test execution results, such as `PASS` or `FAIL`.

The `chmod-tc` test case contains test purposes as follows:

`tp1` A successful `chmod` of a file, expecting a return code of 0.

`tp2` A failed `chmod` of a non-existent file, expecting a return code of –1 and `errno` set to `ENOENT`.

`tp3` A failed `chmod` of a file that contains a non-directory path component, expecting a return code of –1 and `errno` set to `ENOTDIR`.

Functions `tp1()` and `tp2()` are shown here and are described below.

```
static void
tp1()         /* successful chmod of file: return 0 */
{
    int ret, err;
    mode_t mode;

    tet_infoline("SUCCESSFUL CHMOD OF FILE");

    /* change mode of file created in startup function */

    errno = 0;
    if ((ret=chmod(tfile, (mode_t)0)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "chmod(\"%s\", 0) returned %d, expected 0",
```

```
                tfile, ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
            return;
        }

        /* check mode was changed correctly */

        if (stat(tfile, &buf) == -1)
        {
            (void) sprintf(msg,
                "stat(\"%s\", buf) failed - errno %d", tfile, errno);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }

        mode = buf.st_mode & O_ACCMODE;
        if (mode != 0)
        {
            (void) sprintf(msg, "chmod(\"%s\", 0) set mode to 0%lo, expected 0",
                tfile, (long)mode);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }

    static void
    tp2()       /* chmod of non-existent file: return -1, errno ENOENT */
    {
        int ret, err;

        tet_infoline("CHMOD OF NON-EXISTENT FILE");

        /* ensure file does not exist */

        if (stat("chmod.2", &buf) != -1 && unlink("chmod.2") == -1)
        {
            tet_infoline("could not unlink chmod.2");
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* check return value and errno set by call */

        errno = 0;
        ret = chmod("chmod.2", (mode_t)0);
```

```
        if (ret != -1 || errno != ENOENT)
        {
            err = errno;
            if (ret != -1)
            {
                (void) sprintf(msg,
                    "chmod(\"chmod.2\", 0) returned %d, expected -1", ret);
                tet_infoline(msg);
            }

            if (err != ENOENT)
            {
                (void) sprintf(msg,
                    "chmod(\"chmod.2\", 0) set errno to %d, expected %d (ENOENT)",
                    err, ENOENT);
                tet_infoline(msg);
            }

            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }
```

The comments for the code should clarify what is happening on each line. However, it is important to note that a lot of useful diagnostics have been written right into the tests. If any of the system calls fail, whether it is the one being specifically tested or one that the test relies on, that failure will be reported. Also, the tests begin the same, with a message about the test's purpose; and they end the same, with a pass/fail result being reported.

This sort of consistency yields two important benefits:

- Test purposes will be easier to write when they follow some sort of template.
- Test purposes will be easier to debug and evaluate when diagnostic information is built in from the very start.

14.6 Results that must be verified by the user

Some test cases may require user verification of information generated by a test case. An example of this can be found in the `uname-tc` test case when system specific information is being reported.

```
    static void
    tp1()         /* successful uname: return 0 */
    {
        int ret, err;
        struct utsname name;

        tet_infoline("UNAME OUTPUT FOR MANUAL CHECK");

        /* The test cannot determine automatically whether the information
           returned by uname() is correct.  It therefore outputs the
           information with an INSPECT result code for checking manually. */
```

```
        errno = 0;
        if ((ret=uname(&name)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "uname() returned %d, expected 0", ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else
        {
            (void) sprintf(msg, "System Name:  \"%s\"", name.sysname);
            tet_infoline(msg);
            (void) sprintf(msg, "Node Name:    \"%s\"", name.nodename);
            tet_infoline(msg);
            (void) sprintf(msg, "Release:      \"%s\"", name.release);
            tet_infoline(msg);
            (void) sprintf(msg, "Version:      \"%s\"", name.version);
            tet_infoline(msg);
            (void) sprintf(msg, "Machine Type: \"%s\"", name.machine);
            tet_infoline(msg);

            tet_result(TET_INSPECT);
        }
    }
```

Since the information from `uname()` will be different on every machine, the output needs to be reported and then verified. Here the information is simply being printed out for the tester to see and check, but no attempt has been made to interact with the tester to receive verification of the information and then use that verification to set the pass/fail result. Instead, a result code of `INSPECT` has been used.

14.7 Child processes and subprograms

Some test purposes require the creation of a child process or execution of a subprogram. TETware provides several interfaces to facilitate this, as follows:

`tet_fork()` an API function called by test purposes to create a child process and perform processing in parent and child concurrently.

`tet_exec()` an API function called by child processes to execute subprograms.

`tet_main()` a user-supplied function to be defined in subprograms executed by `tet_exec()`.

An example of their use can be found in test purpose `tp4` of the `fileno` test case:

```
static void
tp4()         /* on entry to main(), stream position of stdin, stdout and
                 stderr is same as fileno(stream) */
{
    tet_infoline("ON ENTRY TO MAIN, STREAM POSITION OF STDIN, \
        STDOUT AND STDERR");

    /* fork and execute subprogram, so that unique file positions can be
       set up on entry to main() in subprogram */

    (void) tet_fork(ch4, TET_NULLFP, 30, 0);
}

static void
ch4()
{
    int fd, ret;
    static char *args[] = { "./fileno-t4", NULL };

    /* set up file positions to be inherited by stdin/stdout/stderr
       in subprogram */

    for (fd = 0; fd < 3; fd++)
    {
        (void) close(fd);
        if ((ret=open("fileno.4", O_RDWR|O_CREAT, S_IRWXU)) != fd)
        {
            (void) sprintf(msg, "open() returned %d, expected %d", ret, fd);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }
        if (lseek(fd, (off_t)(123 + 45*fd), SEEK_SET) == -1)
        {
            (void) sprintf(msg, "lseek() failed - errno %d", errno);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }
    }

    /* execute subprogram to carry out remainder of test */

    (void) tet_exec(args[0], args, environ);

    (void) sprintf(msg, "tet_exec(\"%s\", args, env) failed - errno %d",
        args[0], errno);
    tet_infoline(msg);
    tet_result(TET_UNRESOLVED);
}
```

All the testing is done in the child, so the function `tp4()` simply calls `tet_fork()` and ignores the return value. If it needed to do any processing after the call to `tet_fork()`, it should check that the return value was one of the expected child exit codes before continuing.

The arguments to `tet_fork()` are as follows:

- A function to be executed in the child.
- A function to be executed in the parent. In this case no parent processing is required, so the null function pointer `TET_NULLFP` (defined in `tet_api.h`) is used.
- A timeout period in seconds.
- A bitwise OR of the valid child exit codes. In this case the only valid exit code is zero.

The file `fileno-t4.c` contains the definition of `tet_main()`, as follows:

```
int
tet_main(argc, argv)
int argc;
char **argv;
{
    long ret, pos;
    int fd, err, fail = 0;
    static FILE *streams[] =  {  stdin,   stdout,   stderr };
    static char *strnames[] = { "stdin", "stdout", "stderr" };

    /* check file positions of streams are same as set up in parent */

    for (fd = 0; fd < 3; fd++)
    {
        pos = 123 + 45*fd; /* must match lseek() in parent */
        errno = 0;
        if ((ret = ftell(streams[fd])) != pos)
        {
            err = errno;
            (void) sprintf(msg, "ftell(%s) returned %ld, expected %ld",
                strnames[fd], ret, pos);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            fail = 1;
        }
    }

    if (fail == 0)
        tet_result(TET_PASS);
    else
        tet_result(TET_FAIL);

    return 0;
}
```

The `tet_fork()` API function relies for its operation on the `fork()` system call which is provided by the UNIX operating system. Since `fork()` is not available on Win32 operating systems, the `tet_fork()` and `tet_exec()` API functions are not provided when TETware runs on Win32 systems.

In order to assist test suite authors in writing test cases which are portable to both UNIX and Win32 systems, TETware provides the `tet_spawn()` and `tet_wait()` API functions which may be used to facilitate subprogram execution. These functions are available on UNIX systems as well as on Win32 systems.

14.8 Cleaning up test cases

Since test cases often change and/or create data, it is important to cleanup this data before exiting the test case. As explained earlier, one way to do this is to specify a cleanup function with TETware's `tet_cleanup` utility. The cleanup function named in the `stat-tc` test case provides a good example.

```
static void
cleanup()
{
    /* remove file created by start-up */
    (void) unlink(tfile);

    /* remove files created by test purposes, in case they don't run
       to completion */
    (void) rmdir("stat.d");
    (void) unlink("stat.p");
}
```

The `cleanup` function is called when all the test purposes have finished executing. As shown, it simply removes the files and directory that were created during the test.

15. Writing a Shell language API-conforming test suite

15.1 Introduction

This chapter describes a sample non-distributed test suite that conforms to TETware's shell language binding of the API. The source code for the test suite can be found in the appendix entitled ''Example Shell API test suite source files'' at the end of this guide.

This test suite has been designed to run on a UNIX type of operating system. Some minor changes may be required in order to make this test suite function correctly on Win32 operating systems.

The test suite described in this chapter uses the Shell (`xpg3sh`) TCM and API. It can be adapted to use the Korn Shell (`ksh`) TCM and API by changing the single line in each test case which determines which TCM is to be used. This is possible because test cases in the test suite do not use syntax which is specific to either type of shell.

This sample test suite, like the one in the chapter entitled ''Writing a C language API-conforming test suite'', is designed to illustrate how a test suite can be structured under TETware, as well as how individual test cases and their test purposes relate to each other and to the API. Like the `C-API` test suite, this test suite has been deliberately kept simple and realistic. However, instead of system calls being tested, the equivalent user-level commands are tested. Sample tests include checking a returned error code and error message against an expected error code and expected error message and printing out system specific information for verification by the tester.

Note that no support for distributed test cases is provided by the Shell API in Distributed TETware. It is possible to execute test cases on a local system or on one or more remote systems, but no synchronisation between test parts on multiple systems is possible. When Distributed TETware is used it is necessary to supply a `systems` file. In addition, it is necessary to supply a `tetdist.cfg` file if the test suite is to be processed on remote systems or if Distributed TETware has been built to use the XTI network transport.

Small segments of code from the test suite appear in the following sections to help illustrate specific points. Refer to the appropriate section in the appendix entitled ''Example Shell API test suite source files'' at the end of this guide to see the code in its entirety.

15.2 Defining a test suite

As explained in the chapter entitled ''Writing a C language API-conforming test suite'', test suites reside in subdirectories of *tet-root*. The name of the subdirectory and the test suite are the same.

The following figure shows the component files of the sample test suite, called `SHELL-API`:

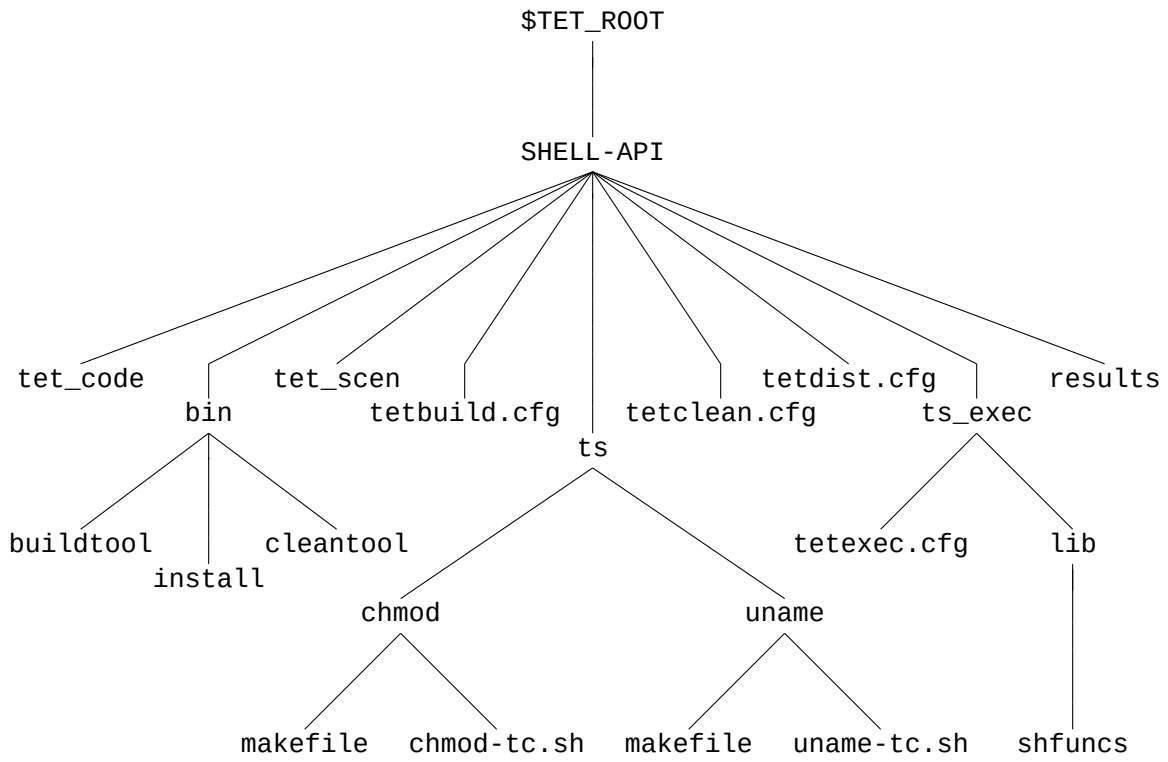

Figure 18. Directory structure for the example Shell language test suite

The make-up of this test suite is similar to the `C-API` test suite and contains the following files:

- An install script, build tool and clean tool in the `bin` directory.
- Some configuration files for test build, execution, and cleanup
- A control file, `tet_scen`.
- A result codes file, `tet_code`.
- Several test cases in a directory structure under the directory `ts`.
- An alternate execution directory `ts_exec`.
- A results directory.

The control file, `tet_scen`, is similar to the control file for the `C-API` test suite. See the chapter entitled ‘‘Writing a C language API-conforming test suite’’ for a description of the control file and how its structure relates to the scenarios that can be run.

The installation utility `install` creates the directory structure under the alternate execution directory to match the structure under the `ts` directory. For the purpose of this example the location of the alternate execution directory is fixed as `$TET_ROOT/SHELL-API/ts_exec` but in general it would be obtained from the user and could be located anywhere.

The build tool is used to build each test case. It is invoked in the source directory of the test case and installs the relevant files under the alternate execution directory. It does this by calling `make`, setting `TET_EXECUTE` to the correct value on the `make` command line in order to override the default value in each `makefile`.

The clean tool is used to remove the installed files from under the alternate execution directory. It works in the same way as the build tool except it executes a `make clean` instead of just `make`.

15.3 Defining common test case functions and variables

Just as with the `C-API` test suite, it makes good sense to minimise code redundancy by grouping together common functions and variables. In the process of writing the `SHELL-API` test suite, several common functions were created. This code was collected into a file named `shfuncs`, which is in the `lib` subdirectory of the `ts_exec` directory. The `shfuncs` file is sourced into each of the two test cases using the shell built-in `.` (dot) command. The required TETware Shell API file, `tcm.sh` (which in turn sources in the other required TETware Shell API file `tetapi.sh`) is also sourced into each of the two test cases. It is important to note the point at which these files are sourced in. Since the TETware API files read definitions and begin execution when they are sourced in, they must be sourced in as the very last part of each test case. Therefore, the last line of each of the test cases sources in `tcm.sh`.

`shfuncs` contains the following functions:

```
# shfuncs : test suite common shell functions

tpstart() # write test purpose banner and initialise variables
{
    tet_infoline "$*"
    FAIL=N
}

tpresult() # give test purpose result
{
    # $1 is result code to give if FAIL=N (default PASS)
    if [ $FAIL = N ]
    then
        tet_result ${1-PASS}
    else
        tet_result FAIL
    fi
}

check_exit() # execute command (saving output) and check exit code
{
    # $1 is command, $2 is expected exit code (0 or "N" for non-zero)
    eval "$1" > out.stdout 2> out.stderr
    CODE=$?
    if [ $2 = 0 -a $CODE -ne 0 ]
    then
        tet_infoline "Command ($1) gave exit code $CODE, expected 0"
        FAIL=Y
    elif [ $2 != 0 -a $CODE -eq 0 ]
    then
```

```
            tet_infoline "Command ($1) gave exit code $CODE, expected non-zero"
            FAIL=Y
        fi
    }

    check_nostdout() # check that nothing went to stdout
    {
        if [ -s out.stdout ]
        then
            tet_infoline "Unexpected output written to stdout, as shown below:"
            infofile out.stdout stdout:
            FAIL=Y
        fi
    }

    check_nostderr() # check that nothing went to stderr
    {
        if [ -s out.stderr ]
        then
            tet_infoline "Unexpected output written to stderr, as shown below:"
            infofile out.stderr stderr:
            FAIL=Y
        fi
    }

    check_stderr() # check that stderr matches expected error
    {
        # $1 is file containing regexp for expected error
        # if no argument supplied, just check out.stderr is not empty

        case $1 in
        "")
            if [ ! -s out.stderr ]
            then
                tet_infoline "Expected output to stderr, but none written"
                FAIL=Y
            fi
            ;;
        *)
            expfile="$1"
            OK=Y
            exec 4<&0 0< "$expfile" 3< out.stderr
            while read expline
            do
                if read line <&3
                then
                    if expr "$line" : "$expline" > /dev/null
                    then
                        :
                    else
                        OK=N
                        break
                    fi
                else
                    OK=N
```

```
            fi
        done
        exec 0<&4 3<&- 4<&-
        if [ $OK = N ]
        then
            tet_infoline "Incorrect output written to stderr, as shown below"
            infofile "$expfile" "expected stderr:"
            infofile out.stderr "received stderr:"
            FAIL=Y
        fi
        ;;
    esac
}

infofile() # write file to journal using tet_infoline
{
    # $1 is file name, $2 is prefix for tet_infoline

    prefix=$2
    while read line
    do
        tet_infoline "$prefix$line"
    done < $1
}
```

Since these functions perform commonly required tasks, they are better defined once rather than twice. Also, should they ever need to be changed, this means changing only one file.

Executing the test for each test purpose in some common, controlled way can make writing the tests and checking their results much easier. The function `check_exit` was written to:

- execute a command in a given argument (`$1`);
- capture both standard error and standard output in separate files (in case one or both need to be checked);
- record the exit code in a variable called `$CODE`;
- output a message to the journal if an unexpected exit code is found.

This function is coded as follows:

```
check_exit() # execute command (saving output) and check exit code
{
    # $1 is command, $2 is expected exit code (0 or "N" for non-zero)
    eval "$1" > out.stdout 2> out.stderr
    CODE=$?
    if [ $2 = 0 -a $CODE -ne 0 ]
    then
        tet_infoline "Command ($1) gave exit code $CODE, expected 0"
        FAIL=Y
    elif [ $2 != 0 -a $CODE -eq 0 ]
    then
        tet_infoline "Command ($1) gave exit code $CODE, expected non-zero"
        FAIL=Y
    fi
}
```

15.4 Initialising test cases

Every test case requires some minimum initialisation of functions and variables. The `uname-tc` test case provides a good illustration of how this initialisation can be handled.

```
:
# uname-tc.sh : test case for uname command

tet_startup=""                          # no startup function
tet_cleanup="cleanup"                   # cleanup function
iclist="ic1 ic2"                        # list invocable components
ic1="tp1"                               # functions for ic1
ic2="tp2"                               # functions for ic2
```

As described in the chapter entitled ‘‘The Shell and Korn Shell APIs’’ elsewhere in this guide, TETware provides the option of naming both a startup and cleanup function. The named startup function will be called before the first test purpose is executed; and the cleanup function will be called after all test purposes have been executed. Here, only a cleanup function is named, by setting `tet_cleanup` equal to the name of the function that will be used.

A `cleanup` function is used by both of the test cases.

```
cleanup() # clean-up function
{
     rm -f out.stdout out.stderr out.experr
}
```

It simply removes the files containing the actual standard output, actual standard error and expected standard error for the test.

The `iclist` variable must contain a space-separated list of the invocable components contained in the test case. This list must be in the form shown above, meaning: `ic1`, `ic2`, and so on. No other names can be used. The next lines define the correspondence between invocable components (`ic`*n*) and the test purpose(s) that they contain. In this test case, each test purpose can be executed individually, so they are assigned to separate invocable components. If, say, `tp2` depended on the prior execution of `tp1`, then the definitions would be:

```
iclist=ic1                              # list invocable components
ic1="tp1 tp2"                           # functions for ic1
```

15.5 Controlling and recording test case execution results

As shown above, a lot of effort has been taken to report on the processing of each test case, and even on the individual test purposes. The `chmod-tc` test case, presented below, shows how information about the processing of a test case can be handled.

The `chmod-tc` test case contains test purposes as follows:

`tp1` successful `chmod` of a file with an expected exit code of 0.

`tp2` failed `chmod` of a non-existent file with an expected exit code of non-zero

`tp3` failed `chmod` due to invalid syntax with an expected exit code of non-zero.

Function `tp1` is shown here and is described below.

```
tp1() # simple chmod of file - successful: exit 0
{
    tpstart "SIMPLE CHMOD OF FILE: EXIT 0"

    echo x > chmod.1 2> out.stderr      # create file
    if [ ! -f chmod.1 ]
    then
        tet_infoline "Could not create test file: chmod.1"
        tet_infoline ‘cat out.stderr‘
        tet_result UNRESOLVED
        return
    fi

    check_exit "chmod 777 chmod.1" 0    # check exit value

    MODE=‘ls -l chmod.1 |cut -d" " -f1‘ # get and check mode of file
    if [ X"$MODE" != X"-rwxrwxrwx" ]
    then
        tet_infoline "chmod 777 set mode to $MODE, expected -rwxrwxrwx"
        FAIL=Y
    fi

    check_nostdout                      # should be no stdout
    check_nostderr                      # should be no stderr

    tpresult                            # set result code
}
```

The comments for the code should clarify what is happening on each line. Like the C-API test cases, this test purpose begins with reporting information about what the test has been designed to check and ends with setting the test result. In this case, this is done by the function tpresult, where the test status variable $FAIL is tested and reported on. If the file that is needed for the test cannot be created, the test outputs diagnostics to the journal and returns a result of UNRESOLVED. Note that in addition to checking the exit code, the file itself is checked to make sure that the mode set by chmod was actually set. Also, since a successful execution of this command means that nothing is written to standard error or standard output, functions contained in shfuncs are used to make sure that no data was output by the command.

In the test purpose, tp2, much of the same function calls are used, even though in this test chmod is expected to fail.

```
tp2() # chmod of non-existent file : exit non-zero
{
    tpstart "CHMOD OF NON-EXISTENT FILE: EXIT NON-ZERO"

    # ensure test file does not exist
    rm -f chmod.2 2> out.stderr
    if [ -f chmod.2 ]
    then
        tet_infoline "Could not remove test file: chmod.2"
        tet_infoline ‘cat out.stderr‘
        tet_result UNRESOLVED
        return
    fi
```

```
        check_exit "chmod 777 chmod.2" N    # check exit value

        check_nostdout                      # should be no stdout
        check_stderr                        # check error message

        tpresult                            # set result code
    }
```

Again, the description of the test is reported, the exit code is checked against that expected by `check_exit`, `tp_result` is called to report the test result, and `check_nostdout` is used to make sure no data was sent to standard output. In this case, however, an error message should be produced; so the expected message is captured in a file in order that it can be later compared with the error message received. This is done through the function `check_stderr` which is defined in `shfuncs` and is shown here.

```
check_stderr() # check that stderr matches expected error
{
    # $1 is file containing regexp for expected error
    # if no argument supplied, just check out.stderr is not empty

    case $1 in
    "")
        if [ ! -s out.stderr ]
        then
            tet_infoline "Expected output to stderr, but none written"
            FAIL=Y
        fi
        ;;
    *)
        expfile="$1"
        OK=Y
        exec 4<&0 0< "$expfile" 3< out.stderr
        while read expline
        do
            if read line <&3
            then
                if expr "$line" : "$expline" > /dev/null
                then
                    :
                else
                    OK=N
                    break
                fi
            else
                OK=N
            fi
        done
        exec 0<&4 3<&- 4<&-
        if [ $OK = N ]
        then
            tet_infoline "Incorrect output written to stderr, as shown below"
            infofile "$expfile" "expected stderr:"
            infofile out.stderr "received stderr:"
            FAIL=Y
        fi
```

```
        ;;
    esac
}
```

The two files, one containing the expected error output in regular expression form, and the other containing the received error output, are compared line-by-line and, if they are identical, nothing is done. However, if they differ, it is important to know how they differ; therefore, both files are printed for the tester to evaluate later and the status of the test purpose is set to show a failed result.

As shown here, a lot of useful diagnostics have been written right into the tests. If any of the commands fail, whether it is the one being specifically tested or one that the test relies on, that failure will be reported. Also, each test case and test purpose begins with information reported in a consistent format; and they end the same, with a pass/fail (or other) result being reported.

As with the `C-API` test suite, this sort of consistency yields two important benefits:

- Test purposes will be easier to write when they follow some sort of template.
- Test purposes will be easier to debug and evaluate when diagnostic information is built in from the very start.

15.6 Results that must be verified by the user

Some test cases may require user verification of information generated by a test case. An example of this can be found in the `uname-tc` test case when system specific information is being reported.

```
tp1() # simple uname of file - successful: exit 0
{
    tpstart "UNAME OUTPUT FOR MANUAL CHECK"

    check_exit "uname -a" 0             # check exit value

    infofile out.stdout                 # send output to journal

    check_nostderr                      # should be no stderr

    tpresult INSPECT                    # set result code
}
```

Since the output from `uname` will be different on every machine, this information needs to be reported and then verified. Here the information is being printed out for the tester to see and check; the test purpose result is `INSPECT` to indicate that the tester must inspect the output in the journal.

15.7 Cleaning up test cases

Since test cases often change and/or create data, it is important to cleanup this data before exiting the test case. As explained earlier, one way to do this is to specify a cleanup function with TETware's `tet_cleanup` utility. The `cleanup` function is the most practical place to specify the removal of temporary files.

16. The distributed demonstration test suite

16.1 Introduction

This chapter describes the official TETware demonstration test suite. The demonstration consists of simple distributed test cases which use the C API. Each test case is designed to execute on the local (or **master**) system and a remote (or **slave**) system. This test suite is useful in that it helps to delineate the basic components of a distributed test suite in its simplest form.

The distributed demonstration test suite has been designed to run on a pair of UNIX systems, a pair of Windows NT systems, or on one UNIX and one Windows NT system. When the demonstration is configured to run between a UNIX and a Windows NT system, you may configure either type of system to act as either master or slave.

Since this is a distributed test suite, it must be processed using Distributed TETware. It cannot be used with TETware-Lite.

Source files for this test suite is included below the directory *tet-root*`/src/tet3/demo` in the TETware distribution. Instructions for building, installing and running the demonstration are presented in the chapter entitled ‘‘Running the TETware demonstrations’’ in the TETware User Guide. An example of the journal file produced when the test case is build, executed and cleaned by Distributed TETware is presented in the appendix entitled ‘‘TETware demonstration journal file’’, also in the TETware User Guide.

Examples of non-distributed test cases, are presented in the chapters entitled ‘‘Writing a C language API-conforming test suite’’ and ‘‘Writing a Shell language API-conforming test suite’’ elsewhere in this guide.

16.2 Test suite files

The following figure shows the file structure of the distributed demonstration test suite on the master system. The same structure is replicated on the slave system except that the `tetdist.cfg`, `tet_code` and `tet_scen` files are not present. It is not necessary for the value of *tet-root* to be the same on each system because configuration variables are available to define it separately for each system.

Each file in the test suite is described in the sections that follow. For ease of reference, listings of all the files in this test suite are presented in the appendix entitled ‘‘Example distributed test case source files’’ at the end of this guide. You should refer to these listings when reading the following sections.

The following figure shows the component files of the example distributed test suite:

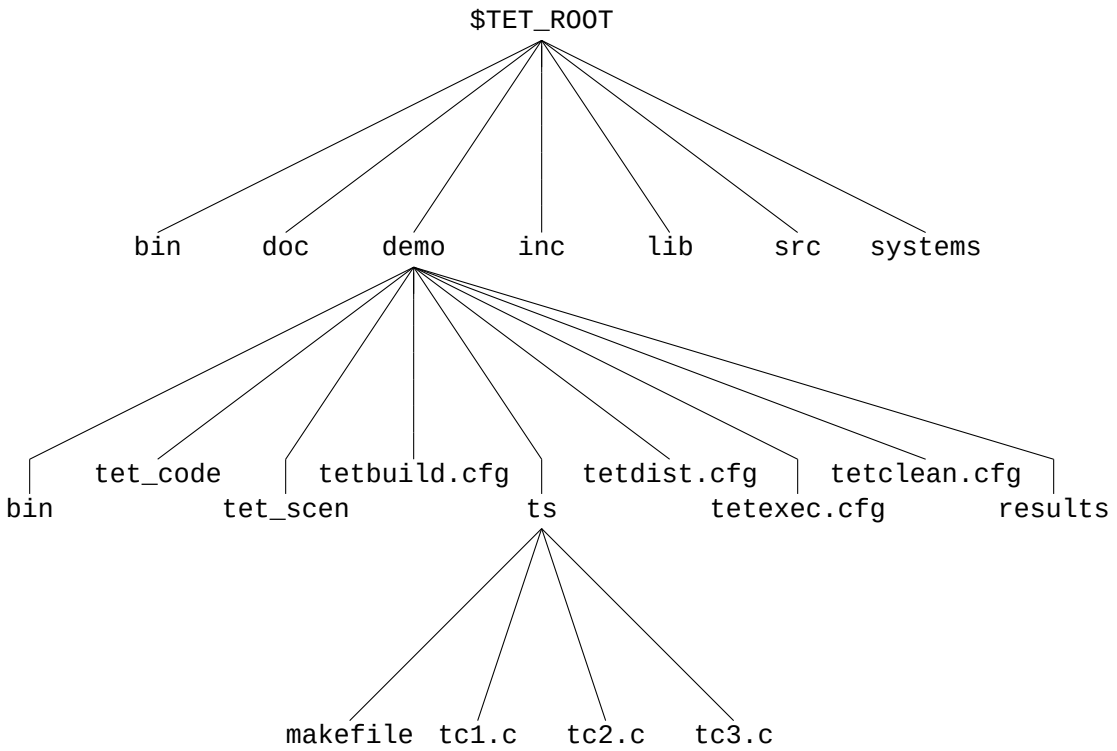

Figure 19. Directory structure for the distributed demonstration test suite

16.2.1 The `systems` file

This file contains the mappings that assign system identifiers to host names. The file must be located in the *tet-root* directory on each system participating in the test.

In the distribution it contains the following lines:

```
#       Example system file for demonstration
000     master
001     slave
```

You must edit this file to contain values that are appropriate for your installation.

If you are using a version of TETware that uses the socket network interface, you only need to replace the names `master` and `slave` with host names suitable for your installation. In addition, you should ensure that these host names are in the hosts databases on both systems.

If you are using a version of TETware that uses XTI as the network transport interface you will need to add a third field to each entry in this file. The extra field should contain the address of the Test Case Controller daemon (`tccd`) on each system. The format of this address is described in the section entitled ‘‘System definitions’’ elsewhere in this guide.

16.2.2 The `tet_code` file

The `tet_code` file is located in the test suite root directory on the master system and contains result code definitions for the test suite.

This file contains the following lines:

```
#       tet_code file for the TETware demonstration
#
# TET reserved codes
0 "PASS"
1 "FAIL"
2 "UNRESOLVED"
3 "NOTINUSE"
4 "UNSUPPORTED"
5 "UNTESTED"
6 "UNINITIATED"
7 "NORESULT"

# Test suite additional codes
101 "FATAL"     Abort
102 "INSPECT"
```

The first group of lines define the standard result codes that are specified in IEEE Std 1003.3-1991. The second group of lines define some extra result codes for use with this particular test suite. Note that when the action indicator field (the third field) is not present, a default action of `Continue` is assumed.

16.2.3 The `tet_scen` file

The `tet_scen` file is located in the test suite root directory on the master system and contains the test suite's scenario, or control, definitions.

This file contains the following lines:

```
#       scenario file for the TETware demonstration
#
all
        "starting scenario"
        :remote,000,001:
        /ts/tc1
        /ts/tc2
        "next is the last test case"
        /ts/tc3
        :endremote:
        "done"
```

This file controls the execution sequence of the test suite. The first non-comment line (`all`) defines the name of the scenario. Subsequent lines contain directives, scenario information lines and test case names. The lines in double quotation marks are scenario information lines that are printed into the journal file. Test case lines list names of test cases to be processed. Although each test case name looks like an absolute path name, it is interpreted relative to the test suite root directory.

The `:remote,000,001:` and `:endremote:` directives tell TETware to process the test cases specified between them on the systems designated `000` and `001` in the `systems` file on the local system.[23] The fact that system `000` is specified with the `remote` directive tells TETware to process the test cases as **distributed** test cases. The fact that system `000` appears first in the system list tells TETware to treat system `000` (the local system) as the master system.

16.2.4 The `tetbuild.cfg` file

The `tetbuild.cfg` file contains variable definitions which determine the way in which TETware processes each test case in build mode. One of these files is provided on each system.

In the distribution the following variables are defined in this file on the master system:

```
TET_BUILD_TOOL=make
TET_BUILD_FILE=-f makefile
TET_OUTPUT_CAPTURE=True
```

The meanings of these variables are as follows:

`TET_BUILD_TOOL` specifies the command to use for building the test cases.

`TET_BUILD_FILE` specifies arguments to pass to the build tool before the test case name.

`TET_OUTPUT_CAPTURE` is used here to specify that all build tool standard output and standard error should be captured and recorded in the journal file, rather than being sent to the default place.

The values specified for these variables in the build configuration file instruct TETware to invoke the following command in the test case source directory when it builds each test case:

`make -f makefile` *test-case*

Setting the value of `TET_OUTPUT_CAPTURE` to `True` provides default values of `False` for `TET_API_COMPLIANT` and `True` for `TET_PASS_TC_NAME`. The values of these two variables tell TETware that the build tool does not use the API and that the test case name should be passed as an argument to the build tool after the argument specified by `TET_BUILD_FILE`.

In the distribution no values are defined in this file on the slave system; therefore the values defined on the master system are used.

The values defined in each file are correct when both of the systems are UNIX systems. The comments in the file on each system show how these values may be changed to support other combinations of system types. Note the way in which the default value of each variable in the slave system is taken from the corresponding value defined on the master system. Also, note the way in which the precedence of variable definitions is used to provide the correct values when either system is a Windows NT system, with the minimum of reconfiguration.

When the test suite is built on a Windows NT system, the file `ntbuild.ksh` is used as the build tool. This file is a shell script which ensures that MKS Make uses the correct configuration

23. That is: the system on which `tcc` is invoked.

file, then it appends a `.exe` suffix to its last argument (the test case name). Finally it invokes `make` with all its arguments. This method of providing portability between UNIX and Windows NT systems enables the number of changes that must be made during the porting operation to be kept to a minimum.

16.2.5 The `tetclean.cfg` file

The `tetclean.cfg` file contains parameters which determine the way in which TETware processes each test case in clean mode. One of these files is provided on each system.

In the distribution the following variables are defined in this file on the master system:

```
TET_CLEAN_TOOL=rm
TET_CLEAN_FILE=-f
TET_OUTPUT_CAPTURE=True
```

The meanings of these variables are the same as those described in the previous section.

In the distribution no values are defined in this file on the slave system; therefore the values defined on the master system are used.

When TETware processes each test case in clean mode, the following command will be executed in the test case source directory:

`rm -f` *test-case*

Again, the default values in each file are correct when both of the systems are UNIX systems. The comments in the file on each system show how these values may be changed to support other combinations of system types.

When the test suite is cleaned on a Windows NT system, the file `ntclean.ksh` is used as the clean tool. This file is a shell script which simply appends a `.exe` suffix to its last argument, then invokes `rm` with all its arguments.

16.2.6 The `tetexec.cfg` file

The `tetexec.cfg` file contains variable definitions which determine the way in which TETware processes each test case in execute mode. One of these files is provided on each system.

In the distribution the following variables are defined in this file on the master system:

```
TET_OUTPUT_CAPTURE=False
TET_EXEC_IN_PLACE=True
```

The meanings of these variables is as follows:

`TET_OUTPUT_CAPTURE` Setting this variable to `False` tells TETware not to record test case output in the journal file.

`TET_EXEC_IN_PLACE` Setting this variable to `False` tells TETware to copy all the files in the test case source directory to a location below the temporary execution directory before executing the test case. This location then becomes the test case execution directory.

Since no value has been specified for `TET_EXEC_TOOL`, TETware executes each test case

directly. Test case execution takes place in the test case execution directory. Setting the value of TET_OUTPUT_CAPTURE to False provides a default value of True for TET_API_COMPLIANT. The value of this variable tells TETware that test cases use the API.

In the distribution no values are defined in this file on the slave system; therefore the values defined on the master system are used. The values defined in these files in the distribution are correct for both UNIX systems and Windows NT systems.

16.2.7 The tetdist.cfg file

This file is only provided on the master system. It contains variable assignments that specify parameters for slave systems that are equivalent to those parameters on the master system that tcc obtains from environment variables or deduces from the current working directory. It may also be used to define network-related parameters when TETware is built to use the XTI network interface.

In the distribution the following variables are defined in this file:

```
TET_REM001_TET_ROOT=/home/tet
TET_REM001_TET_TSROOT=/home/tet/demo
TET_XTI_TPI=/dev/tcp
TET_XTI_MODE=tcp
TET_LOCALHOST=01.02.03.04
```

When you install the demonstration you must change the values of all these variables to values that are correct for your system.

TET_REM001_ is a variable name prefix used to define a variable's value for a particular slave system (in this case, a slave with system designation 001). The name of the variable being defined is the part of the name after this prefix.

The variables defined in this file enable TETware to locate the test suite on the remote system and (when XTI is used) to obtain information to be used by the network transport interface, as follows:

TET_REM001_TET_ROOT	The location of the **tet root** directory on the slave system.
TET_REM001_TET_TSROOT	The location of the **test suite root** directory on the slave system.
TET_XTI_TPI	When the XTI network transport is used, the name of the transport provider interface. This variable is not required when the socket network interface is used.
TET_XTI_MODE	When the XTI network transport is used, the type of transport provider to use. This variable is not required when the socket network interface is used.
TET_LOCALHOST	When the XTI network transport is used and the transport provider is TCP/IP, the master system's external IP address in dotted-decimal notation. This variable is not required when the socket network interface is used.

16.2.8 The `makefile` file

The `makefile` is used by the build tool (`make`) when building each test case.

This file is provided in the test case source directory on each system and contains the following lines:

```
# include file and library locations - don't change
LIBDIR   = ../../lib/tet3
INCDIR   = ../../inc/tet3

# SGS definitions - customise as required for your system
# name of the C compiler
CC       = cc
# the following is appropriate when using the defined build environment
# on a Windows NT system
# CC = cl -nologo

# flags for the C compiler
CFLAGS   = -I$(INCDIR)

# system libraries:
# the socket version on SVR4 and Solaris usually needs -lsocket -lnsl
# the XTI version usually needs -lxti
# the Windows NT version needs wsock32.lib
SYSLIBS =

# suffixes - customise as required for your system
# object file suffix - .o on UNIX, .obj on Windows NT
O = .o
# archive library suffix - .a on UNIX, .lib on windows NT
A = .a
# executable file suffix - blank on UNIX, .exe on Windows NT
E =

all:     tc1$E tc2$E tc3$E

tc1$E:   tc1.c $(INCDIR)/tet_api.h
         $(CC) $(CFLAGS) -o tc1$E tc1.c $(LIBDIR)/tcm$O $(LIBDIR)/libapi$A \
                  $(SYSLIBS)

tc2$E:   tc2.c $(INCDIR)/tet_api.h
         $(CC) $(CFLAGS) -o tc2$E tc2.c $(LIBDIR)/tcm$O $(LIBDIR)/libapi$A \
                  $(SYSLIBS)

tc3$E:   tc3.c $(INCDIR)/tet_api.h
         $(CC) $(CFLAGS) -o tc3$E tc3.c $(LIBDIR)/tcm$O $(LIBDIR)/libapi$A \
                  $(SYSLIBS)
```

This is a typical makefile which contains dependencies and rules for building each individual test case. Note the use of variables to specify the different libraries and file name suffixes that are used on different types of system.

The default values are correct when TETware is built to use the socket network interface on an arbitrary UNIX system. You will probably need to customise each makefile for use on any particular type of system.

16.2.9 The `tc1.c` file

On each system the file `tc1.c` is the source file for the first test case in the test suite.

This test case contains a single test purpose. The master part of the test purpose prints a single test case information line to the journal file by calling `tet_infoline()` and records a `PASS` result by calling `tet_result()`. The slave part of the test purpose does the same. Therefore the consolidated result of the test purpose is `PASS`.

16.2.10 The `tc2.c` file

On each system the file `tc2.c` is the source file for the second test case in the test suite.

This test case contains a single test purpose. The master part of the test purpose prints a number of information lines to the journal file by calling `tet_minfoline()` and records a `PASS` result by calling `tet_result()`. The slave part of the test purpose prints a single test case information line to the journal file by calling `tet_infoline()` and records a `FAIL` result by calling `tet_result()`. Therefore the consolidated result of the test purpose is `FAIL`.

This test purpose illustrates how `tet_minfoline()` may be used to print several lines to the journal as a single operation. If the lines had been printed by calling `tet_infoline()` a number of times from the master test purpose part, it is likely that the line printed by the slave test purpose part would have appeared somewhere in between the lines printed by the master part.

16.2.11 The `tc3.c` file

On each system the file `tc3.c` is the source file for the third test case in the test suite. This test case contains two test purposes. Recall that TETware performs automatic synchronisation between each part of a distributed test case at the start and end of each test purpose. Each test purpose in this test case demonstrate how API functions can be used to perform synchronisation at user-defined points during test purpose execution.

The master and slave parts of the first test purpose each print a message to the journal. Then they synchronise with each other by calling the `tet_remsync()` API function using sync point 101, a sync vote of `YES` and a timeout value of 10 seconds. If the synchronisation request is successful, each test purpose part reports a `PASS` result. Otherwise, diagnostics are printed to the journal and each test purpose part reports an `UNRESOLVED` result.

The master and slave parts of the second test purpose each print a message to the journal. Then they synchronise with each other by calling the `tet_remsync()` API function using sync point 201, a sync vote of `YES` and a timeout value of 10 seconds. In addition, the master test purpose part sends message data with the request and the slave test purpose part expects to receive message data when the call returns. This is done by initialising members of a `tet_synmsg` structure and passing a pointer to this structure as one of the arguments to `tet_remsync()`. If the synchronisation request is successful, each test purpose part reports a `PASS` result. Otherwise, diagnostics are printed to the journal and each test purpose part reports an `UNRESOLVED` result. In addition, the master test purpose part prints the message data before it calls `tet_remsync()` and the slave test purpose part prints the message data received after the successful return of the `tet_remsync()` call.

A common function — `error()` — is used in both parts of the test case to print a diagnostic when an API call is unsuccessful. The first parameter to this function is the value of the global `tet_errno` variable which is set by the API whenever an API function call is unsuccessful. The `error()` function uses this value to index the `tet_errlist[]` array, provided by the

API, which contains short message strings describing each API error that can occur. The API provides a global variable `tet_nerr` which contains the number of entries in the `tet_errlist[]` array. Note that the `error()` function uses this value to check that the value obtained from `tet_errno` refers to an entry which is within the bounds of the `tet_errlist[]` array.

For more information on how TETware synchronisation works, see the chapter entitled ‘‘Test case synchronisation’’ in the TETware User Guide.

APPENDICES

A. The TETware end-user licence

++++++++++++++++++++++++++ TET END USER LICENCE ++++++++++++++++++++++++++

BY OPENING THE PACKAGE, 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 AND RETURN IT TO THE PLACE OF PURCHASE FOR A FULL REFUND.

TETWARE RELEASE 3.2 END USER LICENCE
REDISTRIBUTION NOT PERMITTED

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

A. Free binary evaluation copies – valid for 90 days, full functionality – no warranty.

B. Free binary restricted versions – no warranty, limited functionality.

C. Licenced versions – full functionality, warranty fitness as described in documentation, includes source, binary and annual support.

PART I (A & B above) – TERMS APPLICABLE WHEN LICENCE FEES NOT (YET) PAID (LIMITED TO EVALUATION, EDUCATIONAL AND NON-PROFIT USE).

GRANT.

X/Open grants you a non-exclusive licence to use the Software free of charge if

a. you are a student, faculty member or staff member of an educational institution (K-12, junior college, college or library) or an employee of an organisation which meets X/Open's criteria for a charitable non-profit organisation; or

b. your use of the Software is for the purpose of evaluating whether to purchase an ongoing licence to the Software.

The evaluation period for use by or on behalf of a commercial entity is limited to 90 days; evaluation use by others is not subject to this 90 day limit. Government agencies (other than public libraries) are not considered educational or charitable non-profit organisations for purposes of this Agreement. 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.

X/OPEN DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL X/OPEN 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 (C above) – TERMS APPLICABLE WHEN LICENCE FEES PAID.

GRANT.

Subject to payment of applicable licence fees, X/Open grants to you a non-exclusive licence to use the Software and accompanying documentation (‘‘Documentation’’) as described below.

Copyright © 1996,1997 X/Open Company Ltd.

LIMITED WARRANTY.

X/Open 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. X/Open 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.

SCOPE OF GRANT.

Permission to use for any purpose is hereby granted. Modification of the source is permitted. Redistribution of the source code is not permitted without express written permission of X/Open. Distribution of sources containing adaptations is expressly prohibited.

Redistribution of binaries or binary products containing TETware code is permitted subject to the following conditions:

— this copyright notice is included unchanged with any binary distribution;

— the company distributing binary versions notifies X/Open;

— the company distributing binary versions holds an annual TET support agreement in effect with X/Open for the period the product is being sold, or a one off binary distribution fee equal to four years annual support is paid.

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

Portions of this work contain code and documentation derived from other versions of the Test Environment Toolkit, which contain the following copyright notices:

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 restricts the usage of that code as per this licence.

++

B. Example C language API test suite source files

B.1 Introduction

This appendix contains listings for the files that comprise the example C language test suite presented in the chapter entitled ‘‘Writing a C language API-conforming test suite’’.

This test suite has been designed to run on a UNIX type of operating system.

B.2 `tet_code`

```
# TET reserved codes
0 "PASS"
1 "FAIL"
2 "UNRESOLVED"
3 "NOTINUSE"
4 "UNSUPPORTED"
5 "UNTESTED"
6 "UNINITIATED"
7 "NORESULT"

# Test suite additional codes
33 "INSPECT"
```

B.3 `install`

```
echo This is the C-API test suite install tool.
```

B.4 `cleantool`

```
exec make clean
```

B.5 `tet_scen`

```
#       chmod, fileno, stat, uname test suite.

all
        "Starting Full Test Suite"
        /ts/chmod/chmod-tc
        /ts/fileno/fileno-tc
        /ts/stat/stat-tc
        /ts/uname/uname-tc
        "Completed Full Test Suite"

chmod
        "Starting chmod Test Case"
        /ts/chmod/chmod-tc
        "Finished chmod Test Case"

fileno
        "Starting fileno Test Case"
        /ts/fileno/fileno-tc
        "Finished fileno Test Case"

stat
```

```
        "Starting stat Test Case"
        /ts/stat/stat-tc
        "Finished stat Test Case"

uname
        "Starting uname Test Case"
        /ts/uname/uname-tc
        "Finished uname Test Case"

# EOF
```

B.6 tetbuild.cfg

```
TET_OUTPUT_CAPTURE=True
TET_BUILD_TOOL=make
TET_BUILD_FILE=-f makefile
```

B.7 tetexec.cfg

```
TET_OUTPUT_CAPTURE=False

# The name of a character device file (or "unsup" if not supported)
CHARDEV=/dev/null

# The name of a block device file (or "unsup" if not supported)
BLOCKDEV=unsup
```

B.8 tetclean.cfg

```
TET_OUTPUT_CAPTURE=True
TET_CLEAN_TOOL=cleantool
TET_CLEAN_FILE=
```

B.9 Makefile for chmod-tc.c

```
TET_ROOT = ../../..
LIBDIR   = $(TET_ROOT)/lib/tet3
INCDIR   = $(TET_ROOT)/inc/tet3
CC       = cc
CFLAGS   = -I$(INCDIR) -D_POSIX_SOURCE

chmod-tc:       chmod-tc.c $(INCDIR)/tet_api.h
                $(CC) $(CFLAGS) -o chmod-tc chmod-tc.c $(LIBDIR)/tcm.o \
                        $(LIBDIR)/libapi.a
                -rm -f chmod-tc.o

clean:
                rm -f chmod-tc chmod-tc.o

lint:
                lint $(CFLAGS) chmod-tc.c -ltcm
```

B.10 chmod-tc.c

```
/* chmod-tc.c : test case for chmod() interface */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <tet_api.h>

static void tp1(), tp2(), tp3();
static void startup(), cleanup();

/* Initialize TCM data structures */
void (*tet_startup)() = startup;
void (*tet_cleanup)() = cleanup;
struct tet_testlist tet_testlist[] = {
    { tp1, 1 },
    { tp2, 2 },
    { tp3, 3 },
    { NULL, 0 }
};

/* Test Case Wide Declarations */
static char *tfile = "chmod.1";         /* test file name */
static char *tndir = "chmod.1/chmod.1"; /* path with non-directory in prefix */
static struct stat buf;                 /* buffer for stat(ing) file */
static char msg[256];                   /* buffer for info lines */

static void
startup()
{
    int fd;
    static char *reason = "Failed to create test file in startup";

    if ((fd=creat(tfile, S_IRWXU)) < 0)
    {
        (void) sprintf(msg,
            "creat(\"%s\", S_IRWXU) failed in startup - errno %d",
            tfile, errno);
        tet_infoline(msg);

        /* Prevent tests which use this file from executing */
        tet_delete(1, reason);
        tet_delete(3, reason);
    }
    else
        (void) close(fd);
}

static void
cleanup()
{
    /* remove file created by start-up */
    (void) unlink(tfile);
}
```

```
static void
tp1()         /* successful chmod of file: return 0 */
{
    int ret, err;
    mode_t mode;

    tet_infoline("SUCCESSFUL CHMOD OF FILE");

    /* change mode of file created in startup function */

    errno = 0;
    if ((ret=chmod(tfile, (mode_t)0)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "chmod(\"%s\", 0) returned %d, expected 0",
            tfile, ret);
        tet_infoline(msg);
        if (err != 0)
        {
            (void) sprintf(msg, "errno was set to %d", err);
            tet_infoline(msg);
        }
        tet_result(TET_FAIL);
        return;
    }

    /* check mode was changed correctly */

    if (stat(tfile, &buf) == -1)
    {
        (void) sprintf(msg,
            "stat(\"%s\", buf) failed - errno %d", tfile, errno);
        tet_infoline(msg);
        tet_result(TET_UNRESOLVED);
        return;
    }

    mode = buf.st_mode & O_ACCMODE;
    if (mode != 0)
    {
        (void) sprintf(msg, "chmod(\"%s\", 0) set mode to 0%lo, expected 0",
            tfile, (long)mode);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else
        tet_result(TET_PASS);
}

static void
tp2()       /* chmod of non-existent file: return -1, errno ENOENT */
{
    int ret, err;

    tet_infoline("CHMOD OF NON-EXISTENT FILE");

    /* ensure file does not exist */

    if (stat("chmod.2", &buf) != -1 && unlink("chmod.2") == -1)
```

```
        {
            tet_infoline("could not unlink chmod.2");
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* check return value and errno set by call */

        errno = 0;
        ret = chmod("chmod.2", (mode_t)0);

        if (ret != -1 || errno != ENOENT)
        {
            err = errno;
            if (ret != -1)
            {
                (void) sprintf(msg,
                    "chmod(\"chmod.2\", 0) returned %d, expected -1", ret);
                tet_infoline(msg);
            }

            if (err != ENOENT)
            {
                (void) sprintf(msg,
                    "chmod(\"chmod.2\", 0) set errno to %d, \
                            expected %d (ENOENT)", err, ENOENT);
                tet_infoline(msg);
            }

            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }

    static void
    tp3()       /* non-directory path component: return -1, errno ENOTDIR */
    {
        int ret, err;

        tet_infoline("CHMOD OF NON-DIRECTORY PATH PREFIX COMPONENT");

        /* tndir is a pathname containing a plain file (created by the
           startup function) in the prefix */

        errno = 0;
        ret = chmod(tndir, (mode_t)0);

        /* check return value and errno set by call */

        if (ret != -1 || errno != ENOTDIR)
        {
            err = errno;
            if (ret != -1)
            {
                (void) sprintf(msg,
                    "chmod(\"%s\", 0) returned %d, expected -1", tndir, ret);
                tet_infoline(msg);
            }
```

```
            if (err != ENOTDIR)
            {
                (void) sprintf(msg,
                    "chmod(\"%s\", 0) set errno to %d, expected %d (ENOTDIR)",
                    tndir, err, ENOTDIR);
                tet_infoline(msg);
            }

            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }
```

B.11 Makefile for `fileno-tc.c`

```
TET_ROOT = ../../..
LIBDIR   = $(TET_ROOT)/lib/tet3
INCDIR   = $(TET_ROOT)/inc/tet3
CC       = cc
CFLAGS   = -I$(INCDIR) -D_POSIX_SOURCE

fileno-tc:      fileno-t4 fileno-tc.c $(INCDIR)/tet_api.h
                $(CC) $(CFLAGS) -o fileno-tc fileno-tc.c $(LIBDIR)/tcm.o \
                        $(LIBDIR)/libapi.a
                -rm -f fileno-tc.o

fileno-t4:      fileno-t4.c $(INCDIR)/tet_api.h
                $(CC) $(CFLAGS) -o fileno-t4 fileno-t4.c \
                        $(LIBDIR)/tcmchild.o $(LIBDIR)/libapi.a
                -rm -f fileno-t4.o

clean:
                rm -f fileno-tc fileno-tc.o fileno-t4 fileno-t4.o

lint:
                lint $(CFLAGS) fileno-tc.c -ltcm
                lint $(CFLAGS) fileno-t4.c -ltcmc
```

B.12 `fileno-tc.c`

```
/* fileno-tc.c : test case for fileno() interface */

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

#include <tet_api.h>

extern char **environ;

static void cleanup();
static void tp1(), tp2(), tp3(), tp4(), ch4();

/* Initialize TCM data structures */
```

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

/* Test Case Wide Declarations */
static char msg[256];                   /* buffer for info lines */

static void
cleanup()
{
    (void) unlink("fileno.1");
    (void) unlink("fileno.4");
}

static void
tp1()         /* successful fileno: return fd associated with stream */
{
    FILE *fp;
    struct stat buf1, buf2;

    tet_infoline("FD RETURNED BY FILENO REFERS TO FILE OPEN ON STREAM");

    /* open stream to test file */

    if ((fp=fopen("fileno.1", "w")) == NULL)
    {
        (void) sprintf(msg, "fopen(\"fileno.1\", \"w\") failed - errno %d",
            errno);
        tet_infoline(msg);
        tet_result(TET_UNRESOLVED);
        return;
    }

    /* check device and inode numbers from file descriptor associated
       with the stream match those from the file itself */

    if (stat("fileno.1", &buf1) == -1)
    {
        (void) sprintf(msg, "stat(\"fileno.1\", buf1) failed - errno %d",
            errno);
        tet_infoline(msg);
        tet_result(TET_UNRESOLVED);
        return;
    }

    if (fstat(fileno(fp), &buf2) == -1)
    {
        (void) sprintf(msg, "fstat(fileno(fp), buf2) failed - errno %d",
            errno);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else if (buf1.st_ino != buf2.st_ino || buf1.st_dev != buf2.st_dev)
```

```
        {
            tet_infoline("fileno(fp) does not refer to same file as fp");
            (void) sprintf(msg, "st_dev, st_ino of file: 0x%lx, %ld",
                (long)buf1.st_dev, (long)buf1.st_ino);
            tet_infoline(msg);
            (void) sprintf(msg, "st_dev, st_ino of fileno(fp): 0x%lx, %ld",
                (long)buf2.st_dev, (long)buf2.st_ino);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);

        (void) fclose(fp);
    }

    static void
    tp2()         /* fileno on stdin/stdout/stderr: return 0/1/2 */
    {
        int fd, fail = 0;

        tet_infoline("FILENO ON STDIN/STDOUT/STDERR");

        /* check return value of fileno() for stdin/stdout/stderr */
        /* this code relies on the fact that the TCM does not interfere
           with these streams */

        if ((fd = fileno(stdin)) != 0)
        {
            (void) sprintf(msg, "fileno(stdin) returned %d, expected 0", fd);
            tet_infoline(msg);
            tet_result(TET_FAIL);
            fail = 1;
        }

        if ((fd = fileno(stdout)) != 1)
        {
            (void) sprintf(msg, "fileno(stdout) returned %d, expected 1", fd);
            tet_infoline(msg);
            tet_result(TET_FAIL);
            fail = 1;
        }

        if ((fd = fileno(stderr)) != 2)
        {
            (void) sprintf(msg, "fileno(stderr) returned %d, expected 2", fd);
            tet_infoline(msg);
            tet_result(TET_FAIL);
            fail = 1;
        }

        if (fail == 0)
            tet_result(TET_PASS);
    }

    static void
    tp3()         /* on entry to main(), stdin is readable, stdout and stderr
                     are writable */
    {
```

```
int flags, fail = 0;

tet_infoline("ON ENTRY TO MAIN, STDIN IS READABLE, STDOUT AND STDERR \
    ARE WRITABLE");
/* this code relies on the fact that the TCM does not interfere
   with these streams */

/* check file descriptor associated with stdin is readable */

if ((flags = fcntl(fileno(stdin), F_GETFL)) == -1)
{
    (void) sprintf(msg, "fcntl(fileno(stdin), F_GETFL) failed - errno %d",
        errno);
    tet_infoline(msg);
    tet_result(TET_UNRESOLVED);
    return;
}

flags &= O_ACCMODE;
if (flags != O_RDONLY && flags != O_RDWR)
{
    tet_infoline("stdin is not readable");
    fail = 1;
}

/* check file descriptor associated with stdout is writable */

if ((flags = fcntl(fileno(stdout), F_GETFL)) == -1)
{
    (void) sprintf(msg, "fcntl(fileno(stdout), F_GETFL) failed - errno %d",
        errno);
    tet_infoline(msg);
    tet_result(TET_UNRESOLVED);
    return;
}

flags &= O_ACCMODE;
if (flags != O_WRONLY && flags != O_RDWR)
{
    tet_infoline("stdout is not writable");
    fail = 1;
}

/* check file descriptor associated with stderr is writable */

if ((flags = fcntl(fileno(stderr), F_GETFL)) == -1)
{
    (void) sprintf(msg, "fcntl(fileno(stderr), F_GETFL) failed - errno %d",
        errno);
    tet_infoline(msg);
    tet_result(TET_UNRESOLVED);
    return;
}

flags &= O_ACCMODE;
if (flags != O_WRONLY && flags != O_RDWR)
{
    tet_infoline("stderr is not writable");
    fail = 1;
```

```
    }

    if (fail == 0)
        tet_result(TET_PASS);
    else
        tet_result(TET_FAIL);
}

static void
tp4()         /* on entry to main(), stream position of stdin, stdout and
                 stderr is same as fileno(stream) */
{
    tet_infoline("ON ENTRY TO MAIN, STREAM POSITION OF STDIN, STDOUT \
        AND STDERR");

    /* fork and execute subprogram, so that unique file positions can be
       set up on entry to main() in subprogram */

    (void) tet_fork(ch4, TET_NULLFP, 30, 0);
}

static void
ch4()
{
    int fd, ret;
    static char *args[] = { "./fileno-t4", NULL };

    /* set up file positions to be inherited by stdin/stdout/stderr
       in subprogram */

    for (fd = 0; fd < 3; fd++)
    {
        (void) close(fd);
        if ((ret=open("fileno.4", O_RDWR|O_CREAT, S_IRWXU)) != fd)
        {
            (void) sprintf(msg, "open() returned %d, expected %d", ret, fd);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }
        if (lseek(fd, (off_t)(123 + 45*fd), SEEK_SET) == -1)
        {
            (void) sprintf(msg, "lseek() failed - errno %d", errno);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }
    }

    /* execute subprogram to carry out remainder of test */

    (void) tet_exec(args[0], args, environ);

    (void) sprintf(msg, "tet_exec(\"%s\", args, env) failed - errno %d",
        args[0], errno);
    tet_infoline(msg);
    tet_result(TET_UNRESOLVED);
}
```

B.13 fileno-t4.c

```
/* fileno-t4.c : child program of test purpose 4 for fileno() */

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#include <tet_api.h>

static char msg[256];                   /* buffer for info lines */

/* ARGSUSED */

int
tet_main(argc, argv)
int argc;
char **argv;
{
    long ret, pos;
    int fd, err, fail = 0;
    static FILE *streams[] =  {  stdin,   stdout,   stderr };
    static char *strnames[] = { "stdin", "stdout", "stderr" };

    /* check file positions of streams are same as set up in parent */

    for (fd = 0; fd < 3; fd++)
    {
        pos = 123 + 45*fd; /* must match lseek() in parent */
        errno = 0;
        if ((ret = ftell(streams[fd])) != pos)
        {
            err = errno;
            (void) sprintf(msg, "ftell(%s) returned %ld, expected %ld",
                strnames[fd], ret, pos);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            fail = 1;
        }
    }

    if (fail == 0)
        tet_result(TET_PASS);
    else
        tet_result(TET_FAIL);

    return 0;
}
```

B.14 Makefile for `stat-tc.c`

```
TET_ROOT = ../../..
LIBDIR   = $(TET_ROOT)/lib/tet3
INCDIR   = $(TET_ROOT)/inc/tet3
CC       = cc
CFLAGS   = -I$(INCDIR) -D_POSIX_SOURCE

stat-tc:        stat-tc.c $(INCDIR)/tet_api.h
                $(CC) $(CFLAGS) -o stat-tc stat-tc.c $(LIBDIR)/tcm.o \
                        $(LIBDIR)/libapi.a
                -rm -f stat-tc.o

clean:
                rm -f stat-tc stat-tc.o

lint:
                lint $(CFLAGS) stat-tc.c -ltcm
```

B.15 `stat-tc.c`

```
/* stat-tc.c : test case for stat() interface */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <tet_api.h>

static void tp1(), tp2(), tp3(), tp4(), tp5(), tp6(), tp7();
static void startup(), cleanup();

/* Initialize TCM data structures */
void (*tet_startup)() = startup;
void (*tet_cleanup)() = cleanup;
struct tet_testlist tet_testlist[] = {
    { tp1, 1 },
    { tp2, 2 },
    { tp3, 3 },
    { tp4, 4 },
    { tp5, 5 },
    { tp6, 6 },
    { tp7, 7 },
    { NULL, 0 }
};

/* Test Case Wide Declarations */
static char *tfile = "stat.1";          /* test file name */
static char *tndir = "stat.1/stat.1";   /* path with non-directory in prefix */
static struct stat buf;                 /* buffer for stat(ing) file */
static char msg[256];                   /* buffer for info lines */

static void
startup()
{
    int fd;
    static char *reason = "Failed to create test file in startup";
```

```
        if ((fd=creat(tfile, S_IRWXU)) < 0)
        {
            (void) sprintf(msg,
                "creat(\"%s\", S_IRWXU) failed in startup - errno %d",
                tfile, errno);
            tet_infoline(msg);

            /* Prevent tests which use this file from executing */
            tet_delete(1, reason);
            tet_delete(7, reason);
        }
        else
            (void) close(fd);
    }

    static void
    cleanup()
    {
        /* remove file created by start-up */
        (void) unlink(tfile);

        /* remove files created by test purposes, in case they don't run
           to completion */
        (void) rmdir("stat.d");
        (void) unlink("stat.p");
    }

    static void
    tp1()         /* successful stat of plain file: return 0 */
    {
        int ret, err;

        tet_infoline("SUCCESSFUL STAT OF PLAIN FILE");

        /* stat file created in startup function and check mode indicates
           a plain file */

        errno = 0;
        if ((ret=stat(tfile, &buf)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
                tfile, ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else if (!S_ISREG(buf.st_mode))
        {
            tet_infoline("S_ISREG(st_mode) was not true for plain file");
            (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
```

```
        else
            tet_result(TET_PASS);
    }

    static void
    tp2()         /* successful stat of directory: return 0 */
    {
        int ret, err;
        char *tdir = "stat.d";

        tet_infoline("SUCCESSFUL STAT OF DIRECTORY");

        /* create a test directory */

        if (mkdir(tdir, S_IRWXU) == -1)
        {
            (void) sprintf(msg,
                "mkdir(\"%s\", S_IRWXU) failed in startup - errno %d",
                tdir, errno);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* stat the directory and check mode indicates a directory */

        errno = 0;
        if ((ret=stat(tdir, &buf)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
                tdir, ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else if (!S_ISDIR(buf.st_mode))
        {
            tet_infoline("S_ISDIR(st_mode) was not true for directory");
            (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);

        (void) rmdir(tdir);
    }

    static void
    tp3()         /* successful stat of FIFO file: return 0 */
    {
        int ret, err;
        char *tfifo = "stat.p";
```

```
        tet_infoline("SUCCESSFUL STAT OF FIFO");

        /* create a test FIFO */

        if (mkfifo(tfifo, S_IRWXU) == -1)
        {
            (void) sprintf(msg,
                "mkfifo(\"%s\", S_IRWXU) failed in startup - errno %d",
                tfifo, errno);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* stat the FIFO and check mode indicates a FIFO */

        errno = 0;
        if ((ret=stat(tfifo, &buf)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
                tfifo, ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else if (!S_ISFIFO(buf.st_mode))
        {
            tet_infoline("S_ISFIFO(st_mode) was not true for FIFO file");
            (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);

        (void) unlink(tfifo);
    }

    static void
    tp4()        /* successful stat of character device file: return 0 */
    {
        int ret, err;
        char *chardev;

        tet_infoline("SUCCESSFUL STAT OF CHARACTER DEVICE FILE");

        /* obtain device name from execution configuration parameter */

        chardev = tet_getvar("CHARDEV");
        if (chardev == NULL || *chardev == '\0')
        {
            tet_infoline("parameter CHARDEV is not set");
            tet_result(TET_UNRESOLVED);
            return;
```

```
        }

        /* check if parameter indicates character devices are not supported */

        if (strcmp(chardev, "unsup") == 0)
        {
            tet_infoline("parameter CHARDEV is set to \"unsup\"");
            tet_result(TET_UNSUPPORTED);
            return;
        }

        /* stat the device and check mode indicates a character device */

        errno = 0;
        if ((ret=stat(chardev, &buf)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
                chardev, ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else if (!S_ISCHR(buf.st_mode))
        {
            (void) sprintf(msg, "S_ISCHR(st_mode) was not true for \"%s\"",
                chardev);
            tet_infoline(msg);
            (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }

    static void
    tp5()         /* successful stat of block device file: return 0 */
    {
        int ret, err;
        char *blockdev;

        tet_infoline("SUCCESSFUL STAT OF BLOCK DEVICE FILE");

        /* obtain device name from execution configuration parameter */

        blockdev = tet_getvar("BLOCKDEV");
        if (blockdev == NULL || *blockdev == '\0')
        {
            tet_infoline("parameter BLOCKDEV is not set");
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* check if parameter indicates block devices are not supported */
```

```
        if (strcmp(blockdev, "unsup") == 0)
        {
            tet_infoline("parameter BLOCKDEV is set to \"unsup\"");
            tet_result(TET_UNSUPPORTED);
            return;
        }

        /* stat the device and check mode indicates a block device */

        errno = 0;
        if ((ret=stat(blockdev, &buf)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
                blockdev, ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else if (!S_ISBLK(buf.st_mode))
        {
            (void) sprintf(msg, "S_ISBLK(st_mode) was not true for \"%s\"",
                blockdev);
            tet_infoline(msg);
            (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }

    static void
    tp6()       /* stat of non-existent file: return -1, errno ENOENT */
    {
        int ret, err;

        tet_infoline("STAT OF NON-EXISTENT FILE");

        /* ensure file does not exist */

        if (stat("stat.6", &buf) != -1 && unlink("stat.6") == -1)
        {
            tet_infoline("could not unlink stat.6");
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* check return value and errno set by call */

        errno = 0;
        ret = stat("stat.6", &buf);

        if (ret != -1 || errno != ENOENT)
        {
```

```
            err = errno;
            if (ret != -1)
            {
                (void) sprintf(msg,
                    "stat(\"stat.6\", 0) returned %d, expected -1", ret);
                tet_infoline(msg);
            }

            if (err != ENOENT)
            {
                (void) sprintf(msg,
                    "stat(\"stat.6\", 0) set errno to %d, expected %d (ENOENT)",
                    err, ENOENT);
                tet_infoline(msg);
            }

            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }

    static void
    tp7()       /* non-directory path component: return -1, errno ENOTDIR */
    {
        int ret, err;

        tet_infoline("STAT OF NON-DIRECTORY PATH PREFIX COMPONENT");

        /* tndir is a pathname containing a plain file (created by the
           startup function) in the prefix */

        errno = 0;
        ret = stat(tndir, &buf);

        /* check return value and errno set by call */

        if (ret != -1 || errno != ENOTDIR)
        {
            err = errno;
            if (ret != -1)
            {
                (void) sprintf(msg,
                    "stat(\"%s\", 0) returned %d, expected -1", tndir, ret);
                tet_infoline(msg);
            }

            if (err != ENOTDIR)
            {
                (void) sprintf(msg,
                    "stat(\"%s\", 0) set errno to %d, expected %d (ENOTDIR)",
                    tndir, err, ENOTDIR);
                tet_infoline(msg);
            }

            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
```

```
}
```

B.16 Makefile for `uname-tc.c`

```
TET_ROOT = ../../..
LIBDIR   = $(TET_ROOT)/lib/tet3
INCDIR   = $(TET_ROOT)/inc/tet3
CC       = cc
CFLAGS   = -I$(INCDIR) -D_POSIX_SOURCE

uname-tc:       uname-tc.c $(INCDIR)/tet_api.h
                $(CC) $(CFLAGS) -o uname-tc uname-tc.c $(LIBDIR)/tcm.o \
                        $(LIBDIR)/libapi.a
                -rm -f uname-tc.o

clean:
                rm -f uname-tc uname-tc.o

lint:
                lint $(CFLAGS) uname-tc.c -ltcm
```

B.17 `uname-tc.c`

```
/* uname-tc.c : test case for uname() interface */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/utsname.h>

#include <tet_api.h>

#undef TET_INSPECT      /* must undefine because TET_ is reserved prefix */
#define TET_INSPECT 33  /* this would normally be in a test suite header */

static void tp1();

/* Initialize TCM data structures */
void (*tet_startup)() = NULL;   /* no start-up function */
void (*tet_cleanup)() = NULL;   /* no clean-up function */
struct tet_testlist tet_testlist[] = {
    { tp1, 1 },
    { NULL, 0 }
};

/* Test Case Wide Declarations */
static char msg[256];                   /* buffer for info lines */

static void
tp1()         /* successful uname: return 0 */
{
    int ret, err;
    struct utsname name;

    tet_infoline("UNAME OUTPUT FOR MANUAL CHECK");

    /* The test cannot determine automatically whether the information
       returned by uname() is correct.  It therefore outputs the
       information with an INSPECT result code for checking manually. */
```

```
        errno = 0;
        if ((ret=uname(&name)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "uname() returned %d, expected 0", ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else
        {
            (void) sprintf(msg, "System Name:  \"%s\"", name.sysname);
            tet_infoline(msg);
            (void) sprintf(msg, "Node Name:    \"%s\"", name.nodename);
            tet_infoline(msg);
            (void) sprintf(msg, "Release:      \"%s\"", name.release);
            tet_infoline(msg);
            (void) sprintf(msg, "Version:      \"%s\"", name.version);
            tet_infoline(msg);
            (void) sprintf(msg, "Machine Type: \"%s\"", name.machine);
            tet_infoline(msg);

            tet_result(TET_INSPECT);
        }
    }
```

C. Example Shell API test suite source files

C.1 Introduction

This appendix contains listings for the files that comprise the example Shell test suite presented in the chapter entitled ‘‘Writing a Shell language API-conforming test suite’’.

This test suite has been designed to run on a UNIX type of operating system. Changes to some of the support files may be required in order to make this test suite function correctly on Win32 operating systems using utilities provided in the MKS Toolkit.

The changes required include at least the following:

- The names of the build tool, clean tool and `install` scripts need a `.ksh` suffix in order to make them executable. The tool definitions in `tetbuild.cfg` and `tetclean.cfg` must be updated to reflect this change.
- The name of each test case must include a `.ksh` suffix. The install target in each test case’s makefile, must be updated to reflect this change, as should each test case name listed in the `tet_scen` file.
- The rule in each test case’s makefile which installs the test case must be modified so that the name of the test case after installation has a `.ksh` suffix. The `chmod` command is not required in the install rule and may be removed.
- It is recommended that each test case should be modified to use the Korn Shell API.

When testing command output, some of the test purposes in this test suite make assumptions about the format of the output which are not correct for Win32 systems. Therefore some of the test purposes which report a PASS result when run on a UNIX system can be expected to report a FAIL result when the test suite is run on a Win32 system.

C.2 `tet_code`

```
# TET reserved codes
0 "PASS"
1 "FAIL"
2 "UNRESOLVED"
3 "NOTINUSE"
4 "UNSUPPORTED"
5 "UNTESTED"
6 "UNINITIATED"
7 "NORESULT"

# Test suite additional codes
33 "INSPECT"
```

C.3 install

```
echo Installing SHELL-API test suite

cd SHELL-API || exit 1

# create alternate execution directory hierarchy
find ts -type d -print |
while read d
do
    if test ! -d ts_exec/"$d"
    then mkdir ts_exec/"$d"
    fi
done
```

C.4 buildtool

```
:
# Check TET_EXECUTE is set
if [ -z "$TET_EXECUTE" ]
then
        echo >&2 "No alternate execution directory supplied to buildtool"
        exit 1
fi

# Set TET_EXECUTE on command line to override default value in makefile
exec make TET_EXECUTE="$TET_EXECUTE"
```

C.5 cleantool

```
:
# Check TET_EXECUTE is set
if [ -z "$TET_EXECUTE" ]
then
        echo >&2 "No alternate execution directory supplied to cleantool"
        exit 1
fi

# Set TET_EXECUTE on command line to override default value in makefile
exec make TET_EXECUTE="$TET_EXECUTE" clean
```

C.6 tet_scen

```
#       chmod, uname test suite.
all
        "Starting Full Test Suite"
        /ts/chmod/chmod-tc
        /ts/uname/uname-tc
        "Completed Full Test Suite"
chmod
        "Starting chmod Test Case"
        /ts/chmod/chmod-tc
        "Finished chmod Test Case"
uname
        "Starting uname Test Case"
```

```
        /ts/uname/uname-tc
        "Finished uname Test Case"
# EOF
```

C.7 tetbuild.cfg

```
TET_OUTPUT_CAPTURE=True
TET_BUILD_TOOL=buildtool
```

C.8 tetexec.cfg

```
TET_OUTPUT_CAPTURE=False
TET_EXEC_IN_PLACE=True
```

C.9 tetclean.cfg

```
TET_OUTPUT_CAPTURE=True
TET_CLEAN_TOOL=cleantool
```

C.10 shfuncs — common functions used in the Shell API test suite

```
# shfuncs : test suite common shell functions

tpstart() # write test purpose banner and initialize variables
{
    tet_infoline "$*"
    FAIL=N
}

tpresult() # give test purpose result
{
    # $1 is result code to give if FAIL=N (default PASS)
    if [ $FAIL = N ]
    then
        tet_result ${1-PASS}
    else
        tet_result FAIL
    fi
}

check_exit() # execute command (saving output) and check exit code
{
    # $1 is command, $2 is expected exit code (0 or "N" for non-zero)
    eval "$1" > out.stdout 2> out.stderr
    CODE=$?
    if [ $2 = 0 -a $CODE -ne 0 ]
    then
        tet_infoline "Command ($1) gave exit code $CODE, expected 0"
        FAIL=Y
    elif [ $2 != 0 -a $CODE -eq 0 ]
    then
        tet_infoline "Command ($1) gave exit code $CODE, expected non-zero"
        FAIL=Y
```

```
    fi
}

check_nostdout() # check that nothing went to stdout
{
    if [ -s out.stdout ]
    then
        tet_infoline "Unexpected output written to stdout, as shown below:"
        infofile out.stdout stdout:
        FAIL=Y
    fi
}

check_nostderr() # check that nothing went to stderr
{
    if [ -s out.stderr ]
    then
        tet_infoline "Unexpected output written to stderr, as shown below:"
        infofile out.stderr stderr:
        FAIL=Y
    fi
}

check_stderr() # check that stderr matches expected error
{
    # $1 is file containing regexp for expected error
    # if no argument supplied, just check out.stderr is not empty

    case $1 in
    "")
        if [ ! -s out.stderr ]
        then
            tet_infoline "Expected output to stderr, but none written"
            FAIL=Y
        fi
        ;;
    *)
        expfile="$1"
        OK=Y
        exec 4<&0 0< "$expfile" 3< out.stderr
        while read expline
        do
            if read line <&3
            then
                if expr "$line" : "$expline" > /dev/null
                then
                    :
                else
                    OK=N
                    break
                fi
            else
                OK=N
            fi
        done
        exec 0<&4 3<&- 4<&-
```

```
        if [ $OK = N ]
        then
            tet_infoline "Incorrect output written to stderr, as shown below"
            infofile "$expfile" "expected stderr:"
            infofile out.stderr "received stderr:"
            FAIL=Y
        fi
        ;;
    esac
}

infofile() # write file to journal using tet_infoline
{
    # $1 is file name, $2 is prefix for tet_infoline

    prefix=$2
    while read line
    do
        tet_infoline "$prefix$line"
    done < $1
}
```

C.11 Makefile for `chmod-tc.sh`

```
TET_EXECUTE = ../../ts_exec
INSTALL_DIR = $(TET_EXECUTE)/ts/chmod

$(INSTALL_DIR)/chmod-tc: chmod-tc.sh clean
        cp chmod-tc.sh $@
        chmod 755 $@

clean:
        rm -f $(INSTALL_DIR)/chmod-tc
```

C.12 `chmod-tc.sh`

```
:
# chmod-tc.sh : test case for chmod command

tet_startup=""                          # no startup function
tet_cleanup="cleanup"                   # cleanup function
iclist="ic1 ic2 ic3"                    # list invocable components
ic1="tp1"                               # functions for ic1
ic2="tp2"                               # functions for ic2
ic3="tp3"                               # functions for ic3

tp1() # simple chmod of file - successful: exit 0
{
    tpstart "SIMPLE CHMOD OF FILE: EXIT 0"

    echo x > chmod.1 2> out.stderr      # create file
    if [ ! -f chmod.1 ]
    then
        tet_infoline "Could not create test file: chmod.1"
        tet_infoline ‘cat out.stderr‘
        tet_result UNRESOLVED
        return
```

```
        fi

        check_exit "chmod 777 chmod.1" 0    # check exit value

        MODE=‘ls -l chmod.1 |cut -d" " -f1‘ # get and check mode of file
        if [ X"$MODE" != X"-rwxrwxrwx" ]
        then
            tet_infoline "chmod 777 set mode to $MODE, expected -rwxrwxrwx"
            FAIL=Y
        fi

        check_nostdout                      # should be no stdout
        check_nostderr                      # should be no stderr

        tpresult                            # set result code
    }

    tp2() # chmod of non-existent file : exit non-zero
    {
        tpstart "CHMOD OF NON-EXISTENT FILE: EXIT NON-ZERO"

        # ensure test file does not exist
        rm -f chmod.2 2> out.stderr
        if [ -f chmod.2 ]
        then
            tet_infoline "Could not remove test file: chmod.2"
            tet_infoline ‘cat out.stderr‘
            tet_result UNRESOLVED
            return
        fi

        check_exit "chmod 777 chmod.2" N    # check exit value

        check_nostdout                      # should be no stdout
        check_stderr                        # check error message

        tpresult                            # set result code
    }

    tp3() # chmod with invalid syntax: exit non-zero
    {
        tpstart "CHMOD WITH INVALID SYNTAX: EXIT NON-ZERO"

        # expected error message
        echo "chmod: illegal option -- :\n.*" > out.experr

        check_exit "chmod -:" N             # check exit value

        check_nostdout                      # should be no stdout
        check_stderr out.experr             # check error message

        tpresult                            # set result code
    }

    cleanup() # clean-up function
    {
         rm -f out.stdout out.stderr out.experr
         rm -f chmod.1
    }

    # source common shell functions
    . $TET_EXECUTE/lib/shfuncs
```

```
# execute shell test case manager - must be last line
. $TET_ROOT/lib/xpg3sh/tcm.sh
```

This test case can be converted to use the Korn Shell API simply by changing the last line in this file to:

```
. $TET_ROOT/lib/ksh/tcm.ksh
```

C.13 Makefile for `uname-tc.sh`

```
TET_EXECUTE = ../../ts_exec
INSTALL_DIR = $(TET_EXECUTE)/ts/uname

$(INSTALL_DIR)/uname-tc: uname-tc.sh clean
        cp uname-tc.sh $@
        chmod 755 $@

clean:
        rm -f $(INSTALL_DIR)/uname-tc
```

C.14 `uname-tc.sh`

```
:
# uname-tc.sh : test case for uname command

tet_startup=""                          # no startup function
tet_cleanup="cleanup"                   # cleanup function
iclist="ic1 ic2"                        # list invocable components
ic1="tp1"                               # functions for ic1
ic2="tp2"                               # functions for ic2

tp1() # simple uname of file - successful: exit 0
{
    tpstart "UNAME OUTPUT FOR MANUAL CHECK"

    check_exit "uname -a" 0             # check exit value

    infofile out.stdout                 # send output to journal

    check_nostderr                      # should be no stderr

    tpresult INSPECT                    # set result code
}

tp2() # uname with invalid syntax: exit non-zero
{
    tpstart "UNAME WITH INVALID SYNTAX: EXIT NON-ZERO"

    # expected error message
    echo "uname: illegal option -- :\n.*" > out.experr

    check_exit "uname -:" N             # check exit value

    check_nostdout                      # should be no stdout
    check_stderr out.experr             # check error message

    tpresult                            # set result code
}

cleanup() # clean-up function
{
```

```
        rm -f out.stdout out.stderr out.experr
}

# source common shell functions
. $TET_EXECUTE/lib/shfuncs

# execute shell test case manager - must be last line
. $TET_ROOT/lib/xpg3sh/tcm.sh
```

This test case can be converted to use the Korn Shell API simply by changing the last line in this file to:

```
. $TET_ROOT/lib/ksh/tcm.ksh
```

D. Example distributed test case source files

D.1 Introduction

This appendix contains listings for the files that comprise the distributed demonstration test suite presented in the chapter entitled ‘‘The distributed demonstration test suite’’.

This test suite has been designed to run on a pair of UNIX systems, a pair of Windows NT systems, or on one UNIX and one Windows NT system. When the demonstration is configured to run between a UNIX and a Windows NT system, you may configure either type of system to act as either master or slave.

As distributed these files contain values which are appropriate when you run the demonstration on two UNIX systems. You must edit some of these these files if you run either part of the demonstration on a Windows NT system. Details of the changes that you must make are presented in comments contained in each file.

D.2 Files supplied on the master system

D.2.1 `tet_code`

```
#       tet_code file for the TETware demonstration
#
# TET reserved codes
0 "PASS"
1 "FAIL"
2 "UNRESOLVED"
3 "NOTINUSE"
4 "UNSUPPORTED"
5 "UNTESTED"
6 "UNINITIATED"
7 "NORESULT"

# Test suite additional codes
101 "FATAL"     Abort
102 "INSPECT"
```

D.2.2 `tet_scen`

```
all
        "starting scenario"
        :remote,000,001:
        /ts/tc1
        /ts/tc2
        "next is the last test case"
        /ts/tc3
        :endremote:
        "done"
```

D.2.3 tetbuild.cfg

```
#       master system build mode configuration file for the TETware
#       demonstration
#

#
#       the build tool:
#
# if both master and slave are UNIX-like systems,
# set TET_BUILD_TOOL to "make" in this file
#
# if both master and slave are Windows NT systems,
# set TET_BUILD_TOOL to "./ntbuild.ksh" in this file
#
# if the master is a UNIX-like system and the slave is a Windows NT system,
# set TET_BUILD_TOOL to "make" in this file and
# set TET_BUILD_TOOL in tetbuild.cfg on the slave system to "./ntbuild.ksh"
#
# if the master is a Windows NT system and the slave is a UNIX-like system,
# set TET_BUILD_TOOL to "make" in this file and
# set TET_REM000_TET_BUILD_TOOL in this file to "./ntbuild.ksh"

TET_BUILD_TOOL=make

# TET_BUILD_TOOL=./ntbuild.ksh
# TET_REM000_BUILD_TOOL=./ntbuild.ksh

# don't change
TET_BUILD_FILE=-f makefile
TET_OUTPUT_CAPTURE=True
```

D.2.4 tetclean.cfg

```
#       master system clean mode configuration file for the TETware
#       demonstration
#

#
#       the clean tool:
#
# if both master and slave are UNIX-like systems,
# set TET_CLEAN_TOOL to "rm" in this file
#
# if both master and slave are Windows NT systems,
# set TET_CLEAN_TOOL to "./ntclean.ksh" in this file
#
# if the master is a UNIX-like system and the slave is a Windows NT system,
# set TET_CLEAN_TOOL to "rm" in this file and
# set TET_CLEAN_TOOL in tetclean.cfg on the slave system to "./ntclean.ksh"
#
# if the master is a Windows NT system and the slave is a UNIX-like system,
# set TET_CLEAN_TOOL to "rm" in this file and
# set TET_REM000_TET_CLEAN_TOOL in this file to "./ntclean.ksh"

TET_CLEAN_TOOL=rm
```

```
# TET_CLEAN_TOOL=./ntclean.ksh
# TET_REM000_CLEAN_TOOL=./ntclean.ksh

# don't change
TET_CLEAN_FILE=-f
TET_OUTPUT_CAPTURE=True
```

D.2.5 tetdist.cfg

```
#       example distributed configuration file for the TETware demonstration
#
#       Please refer to the chapter entitled "Running the TETware
#       demonstration" in the TETware User Guide for
#       instructions on how to customise this file for your installation
#
TET_REM001_TET_ROOT=/home/tet
TET_REM001_TET_TSROOT=/home/tet/demo

# The following variables are referenced only by XTI-based versions of
# TETware

TET_XTI_TPI=/dev/tcp
TET_XTI_MODE=tcp
TET_LOCALHOST=01.02.03.04
```

D.2.6 tetexec.cfg

```
#       master system exec mode configuration file for the TETware
#       demonstration
#
TET_OUTPUT_CAPTURE=False
TET_EXEC_IN_PLACE=False
```

D.2.7 ts/makefile

```
# include file and library locations - don't change
LIBDIR  = ../../lib/tet3
INCDIR  = ../../inc/tet3

# SGS definitions - customise as required for your system
# name of the C compiler
CC      = cc
# the following is appropriate when using the defined build environment
# on a Windows NT system
# CC = cl -nologo

# flags for the C compiler
CFLAGS  = -I$(INCDIR)

# system libraries:
# the socket version on SVR4 and Solaris usually needs -lsocket -lnsl
# the XTI version usually needs -lxti
# the Windows NT version needs wsock32.lib
SYSLIBS =

# suffixes - customise as required for your system
# object file suffix - .o on UNIX, .obj on Windows NT
```

```
O = .o
# archive library suffix - .a on UNIX, .lib on windows NT
A = .a
# executable file suffix - blank on UNIX, .exe on Windows NT
E =

all:    tc1$E tc2$E tc3$E

tc1$E:  tc1.c $(INCDIR)/tet_api.h
        $(CC) $(CFLAGS) -o tc1$E tc1.c $(LIBDIR)/tcm$O $(LIBDIR)/libapi$A \
                $(SYSLIBS)

tc2$E:  tc2.c $(INCDIR)/tet_api.h
        $(CC) $(CFLAGS) -o tc2$E tc2.c $(LIBDIR)/tcm$O $(LIBDIR)/libapi$A \
                $(SYSLIBS)

tc3$E:  tc3.c $(INCDIR)/tet_api.h
        $(CC) $(CFLAGS) -o tc3$E tc3.c $(LIBDIR)/tcm$O $(LIBDIR)/libapi$A \
                $(SYSLIBS)
```

D.2.8 ts/tc1.c

```
#include <stdlib.h>
#include <tet_api.h>

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;
void tp1();

struct tet_testlist tet_testlist[] = { {tp1,1}, {NULL,0} };

void tp1()
{
        tet_infoline("This is the first test case (tc1)");
        tet_result(TET_PASS);
}
```

D.2.9 ts/tc2.c

```
#include <stdlib.h>
#include <tet_api.h>

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;
void tp1();

struct tet_testlist tet_testlist[] = { {tp1,1}, {NULL,0} };

void tp1()
{
        static char *lines[] = {
                "This is the second test case (tc2, master).",
                "",
                "The master part of this test purpose reports PASS",
                "but the slave part of this test purpose reports FAIL",
                "so the consolidated result of the test purpose is FAIL.",
                "",
                "The lines in this block of text are printed by a single",
                "call to tet_minfoline() in the master part of the test",
                "purpose so output from the slave part of the test purpose",
                "won't be mixed up with these lines."
```

```
        };
        static int Nlines = sizeof lines / sizeof lines[0];

        tet_minfoline(lines, Nlines);
        tet_result(TET_PASS);
}
```

D.2.10 ts/tc3.c

```
#include <stdlib.h>
#include <stdio.h>
#include <tet_api.h>

#define TIMEOUT 10      /* sync time out */

int sys1[] = { 1 };     /* system IDs to sync with */

static void error(err, rptstr)
int err;        /* tet_errno value, or zero if N/A */
char *rptstr;   /* failure to report */
{
        char *errstr, *colonstr = ": ";
        char errbuf[20];

        if (err == 0)
                errstr = colonstr = "";
        else if (err > 0 && err < tet_nerr)
                errstr = tet_errlist[err];
        else {
                (void) sprintf(errbuf, "unknown tet_errno value %d", tet_errno);
                errstr = errbuf;
        }

        if (tet_printf("%s%s%s", rptstr, colonstr, errstr) < 0) {
                (void) fprintf(stderr, "tet_printf() failed: tet_errno %d\n",
                        tet_errno);
                exit(EXIT_FAILURE);
        }
}

static void tp1()
{
        tet_infoline("This is tp1 in the third test case (tc3, master)");

        (void) tet_printf("sync with slave (sysid: %d)", *sys1);

        if (tet_remsync(101L, sys1, 1, TIMEOUT, TET_SV_YES,
                                (struct tet_synmsg *)0) != 0) {
                error(tet_errno, "tet_remsync() failed on master");
                tet_result(TET_UNRESOLVED);
        }
        else
                tet_result(TET_PASS);
}

static void tp2()
{
        int rescode = TET_UNRESOLVED;
        struct tet_synmsg msg;
```

```
        static char tdata[] = "test data";

        tet_infoline("This is tp2 in the third test case (tc3, master)");

        (void) tet_printf("send message
                        tdata, *sys1);

        msg.tsm_flags = TET_SMSNDMSG;
        msg.tsm_dlen = sizeof(tdata);
        msg.tsm_data = tdata;

        if (tet_remsync(201L, sys1, 1, TIMEOUT, TET_SV_YES, &msg) != 0)
                error(tet_errno, "tet_remsync() failed on master");
        else if ((msg.tsm_flags & TET_SMSNDMSG) == 0)
                error(0, "tet_remsync() cleared TET_SMSNDMSG flag on master");
        else if (msg.tsm_flags & TET_SMTRUNC)
                error(0, "tet_remsync() set TET_SMTRUNC flag on master");
        else
                rescode = TET_PASS;

        tet_result(rescode);
}

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;

struct tet_testlist tet_testlist[] = { {tp1,1}, {tp2,2}, {NULL,0} };
```

D.3 Files supplied on the slave system

D.3.1 `tetbuild.cfg`

```
#       slave system build mode configuration file for the TETware
#       demonstration
#
#       most of the configuration variables are inherited from the
#       master system
#
#       only variables that are specific to the slave system appear here
#

#
#       the build tool:
#
# when the slave is a UNIX-like system or both master and slave systems
# are of the same type, the value of TET_BUILD_TOOL to use is the one
# inherited from the master system
#
# when the master is a UNIX-like system and the slave is a Windows NT system,
# set TET_BUILD_TOOL to "./ntbuild.ksh" in this file, thus:
#
# TET_BUILD_TOOL=./ntbuild.ksh
```

D.3.2 tetclean.cfg

```
#       slave system clean mode configuration file for the TETware
#       demonstration
#
#       most of the configuration variables are inherited from the
#       master system
#
#       only variables that are specific to the slave system appear here
#

#
#       the clean tool:
#
# when the slave is a UNIX-like system or both master and slave systems
# are of the same type, the value of TET_CLEAN_TOOL to use is the one
# inherited from the master system
#
# when the master is a UNIX-like system and the slave is a Windows NT system,
# set TET_CLEAN_TOOL to "./ntclean.ksh" in this file, thus:
#
# TET_CLEAN_TOOL=./ntclean.ksh
```

D.3.3 tetexec.cfg

```
#       slave system exec mode configuration file for the TETware
#       demonstration
#
#       most of the configuration variables are inherited from the
#       master system
#
#       only variables that are specific to the slave system appear here
#
```

D.3.4 ts/makefile

```
# include file and library locations - don't change
LIBDIR  = ../../lib/tet3
INCDIR  = ../../inc/tet3

# SGS definitions - customise as required for your system
# name of the C compiler
CC      = cc
# the following is appropriate when using the defined build environment
# on a Windows NT system
# CC = cl -nologo

# flags for the C compiler
CFLAGS  = -I$(INCDIR)

# system libraries:
# the socket version on SVR4 and Solaris usually needs -lsocket -lnsl
# the XTI version usually needs -lxti
# the Windows NT version needs wsock32.lib
SYSLIBS =
```

```
# suffixes - customise as required for your system
# object file suffix - .o on UNIX, .obj on Windows NT
O = .o
# archive library suffix - .a on UNIX, .lib on windows NT
A = .a
# executable file suffix - blank on UNIX, .exe on Windows NT
E =

all:    tc1$E tc2$E tc3$E

tc1$E:  tc1.c $(INCDIR)/tet_api.h
        $(CC) $(CFLAGS) -o tc1$E tc1.c $(LIBDIR)/tcm$O $(LIBDIR)/libapi$A \
                $(SYSLIBS)

tc2$E:  tc2.c $(INCDIR)/tet_api.h
        $(CC) $(CFLAGS) -o tc2$E tc2.c $(LIBDIR)/tcm$O $(LIBDIR)/libapi$A \
                $(SYSLIBS)

tc3$E:  tc3.c $(INCDIR)/tet_api.h
        $(CC) $(CFLAGS) -o tc3$E tc3.c $(LIBDIR)/tcm$O $(LIBDIR)/libapi$A \
                $(SYSLIBS)
```

D.3.5 `ts/tc1.c`

```
#include <stdlib.h>
#include <tet_api.h>

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;
void tp1();

struct tet_testlist tet_testlist[] = { {tp1,1}, {NULL,0} };

void tp1()
{
        tet_infoline("This is the first test case (tc1)");
        tet_result(TET_PASS);
}
```

D.3.6 `ts/tc2.c`

```
#include <stdlib.h>
#include <tet_api.h>

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;
void tp1();

struct tet_testlist tet_testlist[] = { {tp1,1}, {NULL,0} };

void tp1()
{
        tet_infoline("This is the second test case (tc2, slave)");

        tet_result(TET_FAIL);
}
```

D.3.7 ts/tc3.c

```
#include <stdlib.h>
#include <stdio.h>
#include <tet_api.h>

#define TIMEOUT 10      /* sync time out */

int sys0[] = { 0 };     /* system IDs to sync with */

static void error(err, rptstr)
int err;        /* tet_errno value, or zero if N/A */
char *rptstr;   /* failure to report */
{
        char *errstr, *colonstr = ": ";
        char errbuf[20];

        if (err == 0)
                errstr = colonstr = "";
        else if (err > 0 && err < tet_nerr)
                errstr = tet_errlist[err];
        else {
                (void) sprintf(errbuf, "unknown tet_errno value %d", tet_errno);
                errstr = errbuf;
        }

        if (tet_printf("%s%s%s", rptstr, colonstr, errstr) < 0) {
                (void) fprintf(stderr, "tet_printf() failed: tet_errno %d\n",
                        tet_errno);
                exit(EXIT_FAILURE);
        }
}

static void tp1()
{
        tet_infoline("This is tp1 in the third test case (tc3, slave)");

        (void) tet_printf("sync with master (sysid: %d)", *sys0);

        if (tet_remsync(101L, sys0, 1, TIMEOUT, TET_SV_YES,
                                (struct tet_synmsg *)0) != 0) {
                error(tet_errno, "tet_remsync() failed on slave");
                tet_result(TET_UNRESOLVED);
        }
        else
                tet_result(TET_PASS);
}

static void tp2()
{
        int rescode = TET_UNRESOLVED;
        struct tet_synmsg msg;
        char rcvbuf[TET_SMMSGMAX];

        tet_infoline("This is tp2 in the third test case (tc3, slave)");

        (void) tet_printf("sync with master (sysid: %d) and receive data",
                        *sys0);
```

```
        msg.tsm_flags = TET_SMRCVMSG;
        msg.tsm_dlen = sizeof(rcvbuf);
        msg.tsm_data = rcvbuf;

        if (tet_remsync(201L, sys0, 1, TIMEOUT, TET_SV_YES, &msg) != 0)
                error(tet_errno, "tet_remsync() failed on slave");
        else if (msg.tsm_sysid == -1)
                error(0, "tet_remsync() set tsm_sysid to -1 on slave");
        else if (msg.tsm_flags & TET_SMTRUNC)
                error(0, "tet_remsync() set TET_SMTRUNC flag on slave");
        else if (msg.tsm_dlen <= 0)
                error(0, "tet_remsync() set tsm_dlen <= 0 on slave");
        else
        {
                (void) tet_printf("received message
                        msg.tsm_dlen, rcvbuf);
                rescode = TET_PASS;
        }

        tet_result(rescode);
}

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;

struct tet_testlist tet_testlist[] = { {tp1,1}, {tp2,2}, {NULL,0} };
```

D.4 Files supplied both systems

D.4.1 `systems`

```
#       Example system file for the TETware demonstration
#
#       Please refer to the chapter entitled "Running the TETware
#       demonstration" in the TETware User Guide for
#       instructions on how to customise this file for your installation
#
000     master
001     slave
```

D.4.2 `ts/ntbuild.ksh`

```
#       build tool for use when the distributed demo suite is to be built
#       on a Windows NT system using MKS Make

MAKESTARTUP=${ROOTDIR:-c:}/etc/msc.mk
export MAKESTARTUP

args=

while test $# -gt 1
do
        args="$args $1"
        shift
done

exec make $args $1.exe
```

D.4.3 ts/ntclean.ksh

```
#       clean tool for use when the distributed demo suite is to be cleaned
#       on a Windows NT system

args=

while test $# -gt 1
do
        args="$args $1"
        shift
done

exec rm $args $1.exe
```

E. Scenario language syntax summary

This appendix contains a brief summary of the syntax of the language that is used in a scenario file. A more complete description is presented in the chapter entitled ‘‘The scenario file’’ elsewhere in this guide.

In these descriptions, a language element enclosed in square brackets ([]) is optional and an ellipsis (. . .) indicates that the previous element(s) may be repeated.

Scenario elements

A scenario consists of a **scenario name**, followed by zero or more **scenario elements**. A scenario element may be a **simple element**, a **directive** or a **directive group**.

Elements are separated from each other by white space. A directive or directive group may have an **attached element** associated with it. An attached element is a simple element that has no white space between it and its directive or directive group.

Form of input

A scenario starts at the start of a line, and may be continued on one or more **continuation lines**. A continuation line is a line which starts with white space. A comment is introduced by # and ends at the end of the line. Blank lines and comments are ignored.

The general form of a scenario is:

scenario-name element . . .

or:

scenario-name
    element
    . . .

or some combination of the two.

Simple elements

The general form of a simple scenario element is:

simple-element

The simple elements are:

"*scenario information line*"
/*test-case-name*
/*file-name*
^*scenario-name*

A "*scenario information line*" always appears by itself.

A /*test-case-name* may appear by itself or may be attached to a directive. When a /*test-case-name* is attached to a directive, it is preceded by a @ character, thus:

:*directive*:@/*test-case-name*

A /*file-name* is always attached to a directive.

A ^*scenario-name* may appear by itself or may be attached to a directive.

A test case name may have a list of invocable components attached to it, thus:

/test-case-name{*ic-list*}

An *ic-list* consists of one or more numbers or number ranges. Each number or number range is separated from the next by a , character. A number range consists of a pair of numbers separated by a – character.

Directives

The general form of a directive is:

:*directive*[,*parameter*...]:*attached-element*

or:

:*directive*[,*parameter*...]:
element
...
:end*directive*:

An *attached-element* may be one of:

@/*test-case-name*
/*file-name*
^*scenario-name*

The directives that are supported in both TETware-Lite and Distributed TETware are:

```
:include:
:parallel[,count]:
:repeat[,count]:
:timed_loop,seconds:
:random:
```

In addition, `group` is accepted as a synonym for `parallel`.

The `include` directive must always have a */file-name* attached to it. The other directive syntax formats may not be used with this directive.

The directives that are supported only in Distributed TETware are:

```
:remote,nnn[,...]:
:distributed,nnn[,...]:
```

The **end** directives that are in both TETware-Lite and Distributed TETware are:

```
:endparallel:
:endrepeat:
:endtimed_loop:
:endrandom:
```

In addition, `endgroup` is accepted as a synonym for `endparallel`.

The **end** directives that are only in Distributed TETware are:

```
:endremote:
:enddistributed:
```

Directive groups

The general form of a directive group is:

:*directive*$_1$;*directive*$_2$...:*attached-element*

or:

:*directive*$_1$;*directive*$_2$...:
element
...
:...end*directive*$_2$;end*directive*$_1$:

Include files

A file specified by a /*file-name* is an **include file**. Each of the (non-blank, non-comment) lines in an include file contains a single simple scenario element.

The following simple scenario elements may appear in an include file:

"*scenario information line*"
/*test-case-name*

Directives and other simple elements may not appear in an include file. Leading white space on a line is permitted but ignored. A comment is introduced by a # character and ends at the end of the line. Blank lines and comments are ignored.

F. Conceptual models used by TETware

F.1 Introduction

This appendix contains diagrams which represent the conceptual models used by TETware. The diagrams presented here are based on similar diagrams which appear in the TET and dTET2 specifications.

F.2 TETware-Lite conceptual model

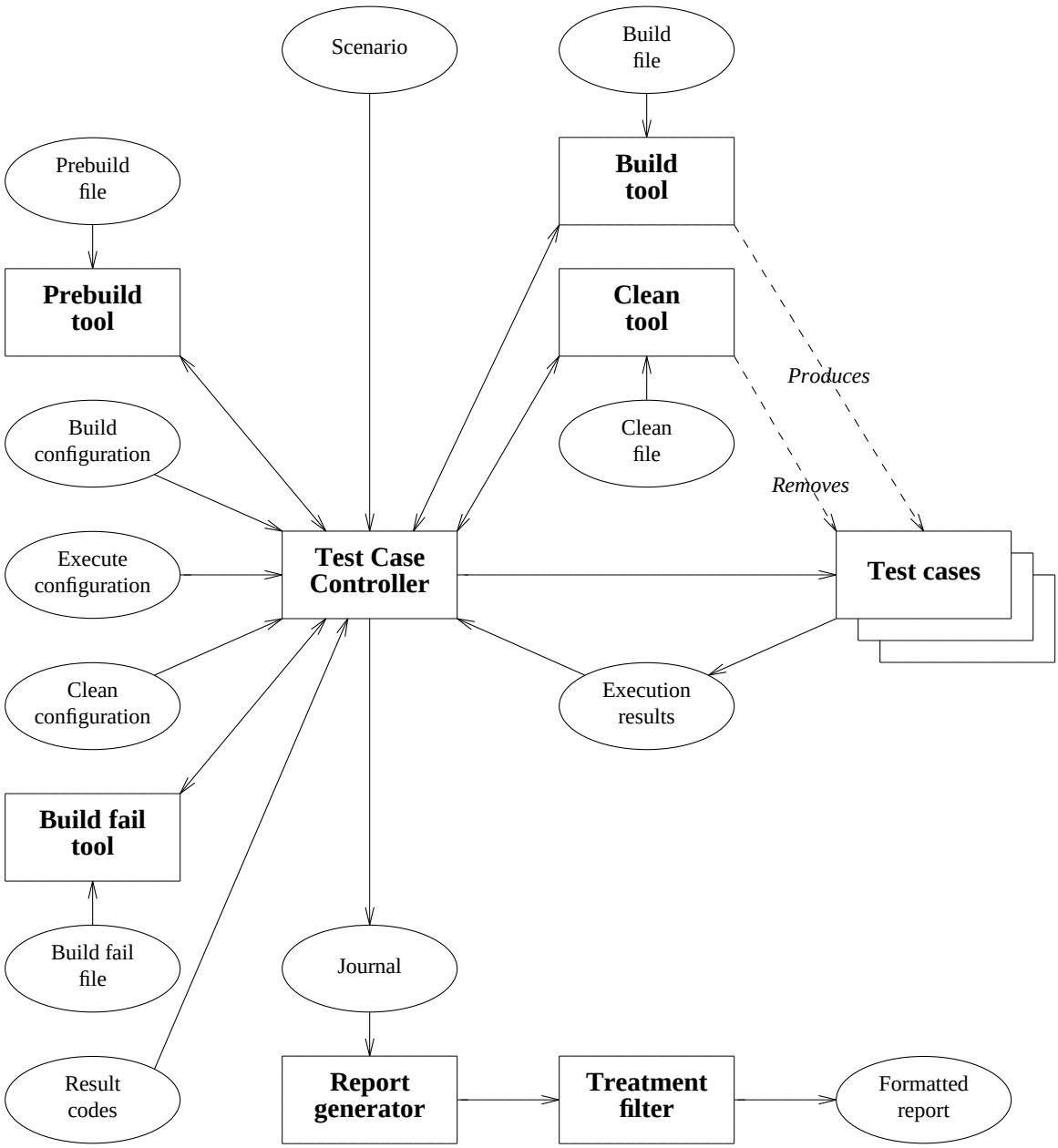

Figure 20. TETware-Lite conceptual model

F.3 Distributed TETware conceptual model – local system with test cases

The system illustrated here has test cases running on it and may also control the processing of test cases on remote systems.

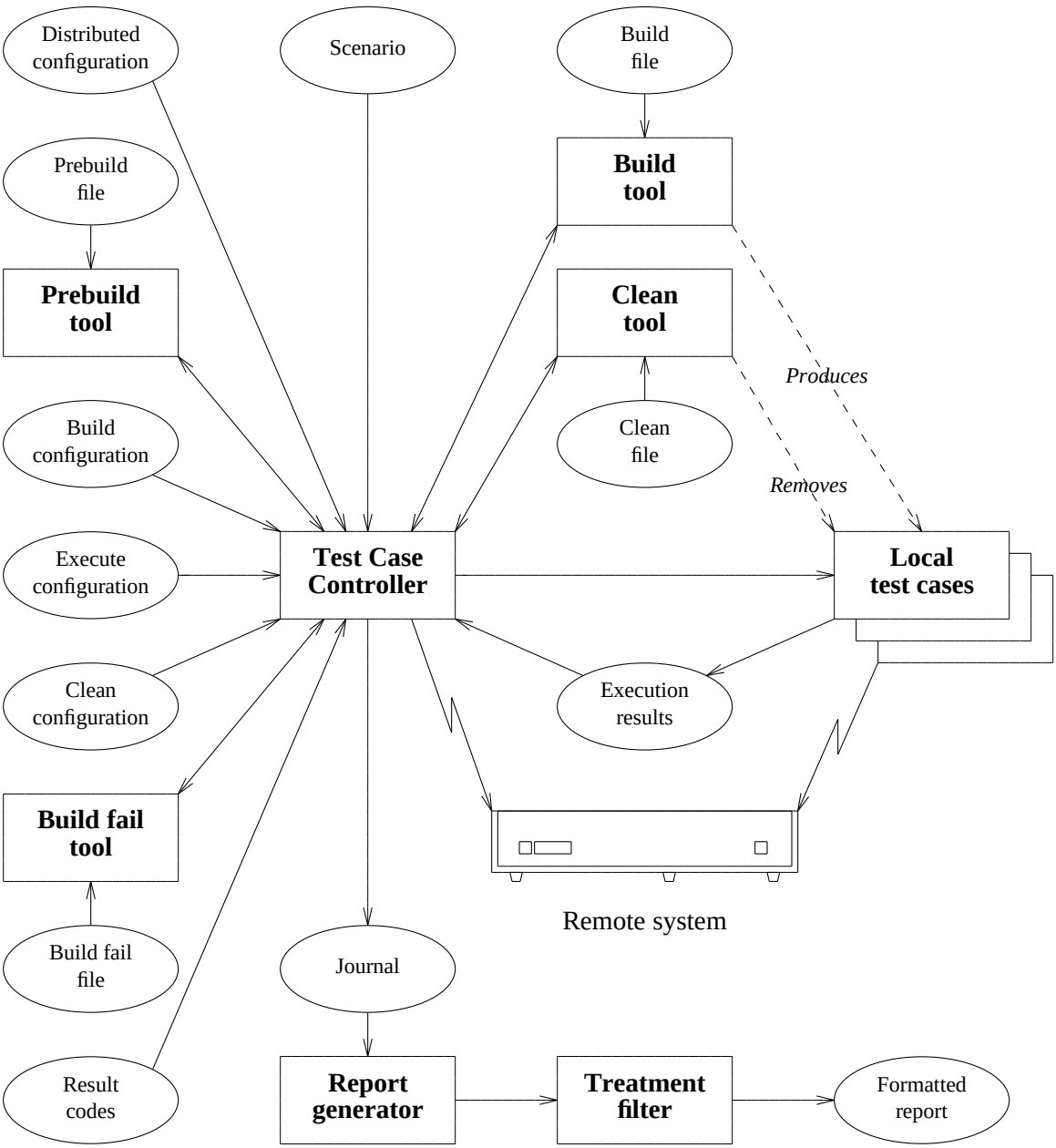

Figure 21. Distributed TETware conceptual model – local system with test cases

F.4 Distributed TETware conceptual model – local system without test cases

The system illustrated here does not have test cases running on it but controls the processing of test cases on remote systems.

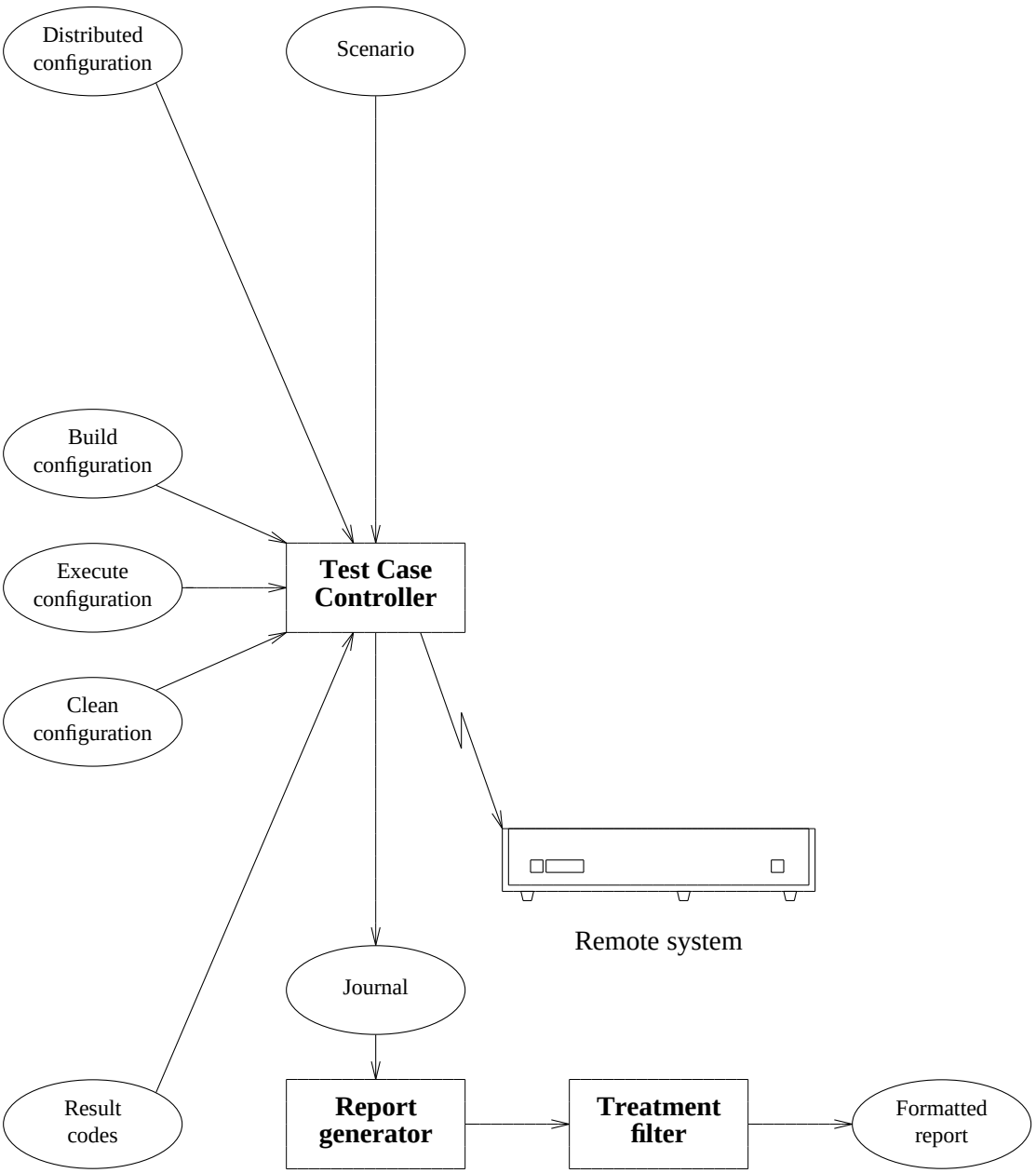

Figure 22. Distributed TETware conceptual model – local system without test cases

F.5 Distributed TETware conceptual model – remote system as master

The system illustrated here may be running remote (non-distributed) test cases or may be running the master parts of distributed test cases.

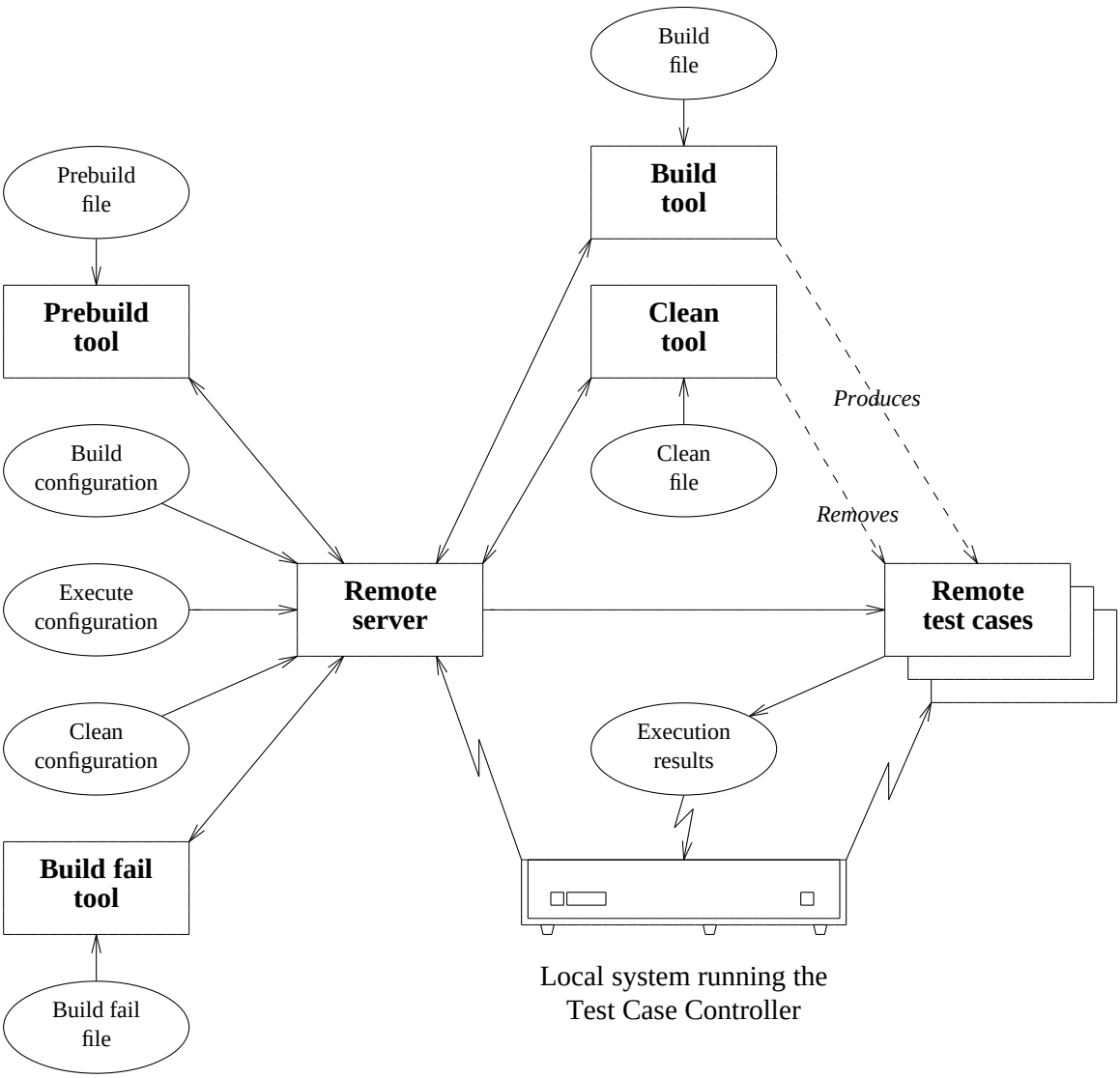

Figure 23. Distributed TETware conceptual model – remote system as master

F.6 Distributed TETware conceptual model – remote system as slave

The system illustrated here is running the slave parts of distributed test cases.

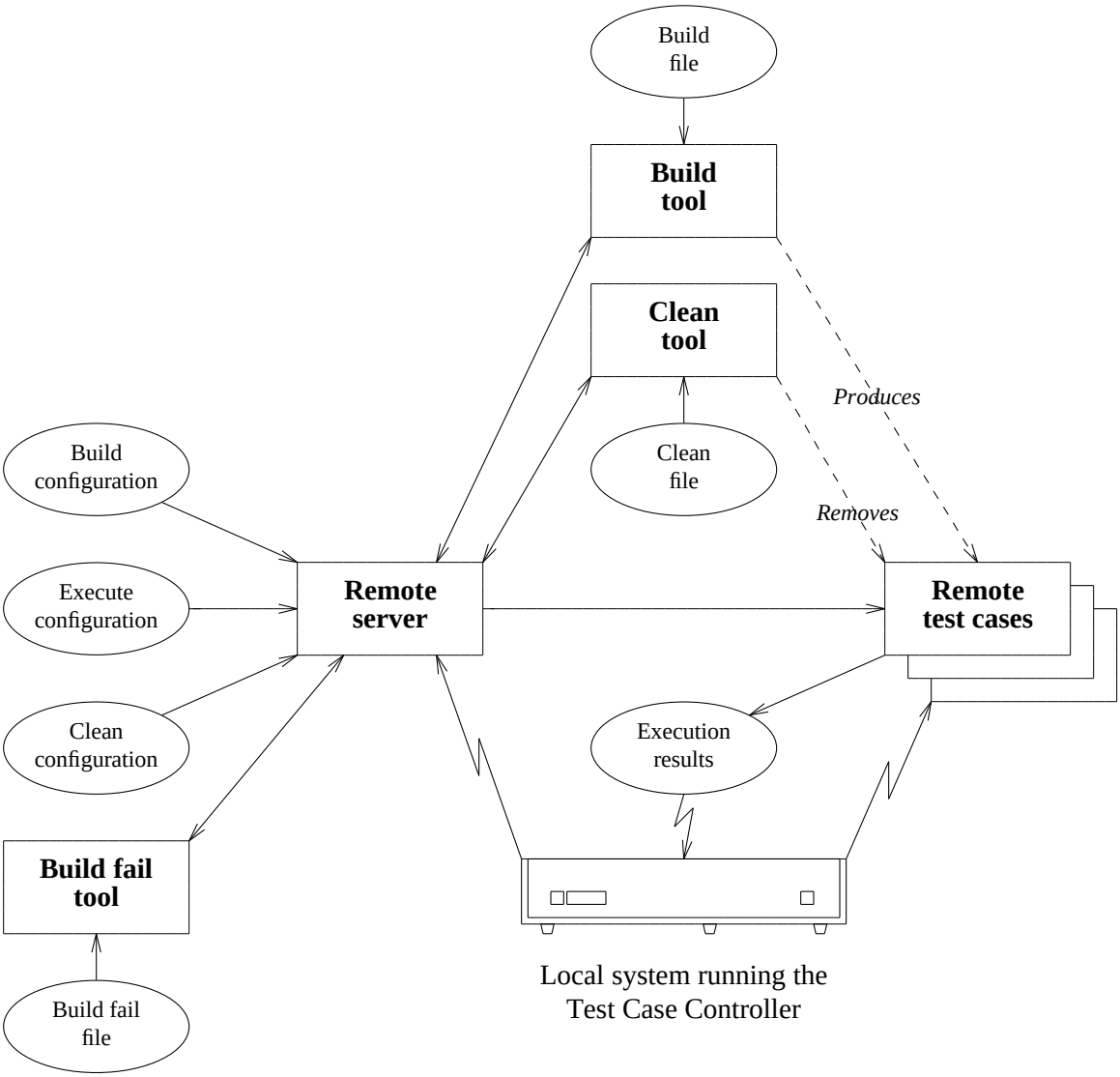

Figure 24. Distributed TETware conceptual model – remote system as slave

G. Background and goals

G.1 Introduction

The goal behind creating TETware and its predecessors is to produce a test driver that accommodates current and future testing needs of the test development community. To achieve this goal, input from a wide sample of the test development community has been used for the specification and development of TETware's functionality and interfaces.

G.2 Previous TET implementations

G.2.1 The Test Environment Toolkit

The TET project started in September of 1989, when the Open Software Foundation, UNIX International, and X/Open entered into an announced agreement to produce a specification for a test environment. These organisations agreed to develop and make freely available an implementation written to that specification; additionally, the three organisations committed to producing test suites for execution within that environment.

In the process of developing a specification, the project invited interested members of the test software development community to discuss their requirements for a test driver. It was the belief of the project that, through careful study of these requirements, a reasonably comprehensive and effective test driver could be specified. Having achieved this, the project expected that a substantial portion of the test development community would begin using TET for the development of conformance testing software.

G.2.2 The Distributed Test Environment Toolkit

The first set of major extensions made to the TET by X/Open was the Distributed Test Environment Toolkit (DTET) project that started in October 1991. The objective of the project was to extend the functionality of the TET to support the execution of **distributed test cases** and be backwards compatible with the TET. The DTET defined a distributed test case as a test case executing partly on a **master** system and partly on one or more **slave** systems. In such a test case, synchronisation between the test case controlling software on the multiple systems is required.

Initially, the DTET was designed for use as the underlying test harness for the development of a number of network testing requirements, including the X.400 Application Programming Interface (API), the OSF Distributed Computing Environment (DCE) and the X/Open Network File System (XNFS) test suites. Following this, the DTET was installed at other sites and has proved to be portable across a wide range of different systems.

The DTET was also able to execute **non-distributed** test cases (on either the master system or on a single or multiple remote systems). However, to do this the test cases had to be linked with the TET API library (`libapi.a`) and not the DTET library (`libdapi.a`). Depending on whether you were writing distributed or non-distributed test cases, you had to be aware of which library to use when linking your test case. However, many users found the ability of the DTET to execute TET test cases an advantage because they did not have to recompile or relink their test suites.

G.2.3 The Extended Test Environment Toolkit

In parallel with X/Open's development of the DTET, another extension to the base TET emerged. This TET version is known as the Extended Test Environment Toolkit and provides a number of enhancements to the base TET which have proved popular with members of the testing community. The latest version of this toolkit variant is ETET release 1.10.3 which appeared in 1994. This ETET release is based on TET release 1.10 and the enhancements contained therein were provided by SunSoft Inc., UNIX Systems Laboratories Inc., and others. The Korn Shell bindings included with ETET were provided by Hewlett-Packard Co.

Features provided in ETET over and above those in the base TET include additional directives to enable complex scenarios to be specified and additional configuration variables to enable more precise control to be exercised over the way in which the Test Case Controller processes test cases.

In addition to the Korn Shell binding mentioned above, the ETET distribution includes a C++ language binding, a Perl API and a quantity of user-contributed demonstration test suites and other software.

G.2.4 The Distributed Test Environment Toolkit Version 2

X/Open then wished to enhance the DTET by incorporating all the features of the TET to produce a common toolkit called dTET2. dTET2 was produced during 1993 and 1994. The dTET2 toolkit rationalised the differences in the TET and DTET toolkits by providing:

- A single toolkit for writing distributed and non-distributed tests, using only a single API.
- New Users' and Programmers' Guides.
- Support for the X/Open Transport Interface (XTI) in addition to Berkeley Sockets in the transport-specific parts of the toolkit.
- Fixes to problems inherited from the DTET and the TET.

G.3 TETware

Lately, X/Open has produced TETware with the objective of combining all the functionality of TET, dTET2 and ETET. In addition, X/Open wished to make TETware available on platforms running the Windows NT and Windows 95 operating systems as well as on UNIX systems and in other POSIX-conforming environments.

TETware is available in two major versions; namely, TETware-Lite and Distributed TETware. Distributed TETware provides all the functionality required to process both non-distributed and distributed test cases on numbers of systems at one time, whereas TETware-Lite is able to process non-distributed test cases on a single system. On POSIX-conforming platforms, TETware-Lite may be built to use only those features specified in POSIX.1.

Unlike previous TET implementations, TETware is provided to users under the terms of a software licence. X/Open intend to make demonstration versions of TETware with restricted functionality available for evaluation purposes.

G.4 Relationship between TETware and its predecessors

TETware includes all of the functionality provided by previous TET implementations, in addition to a number of new features. This is illustrated in the following diagram. Note that this diagram is not to scale.

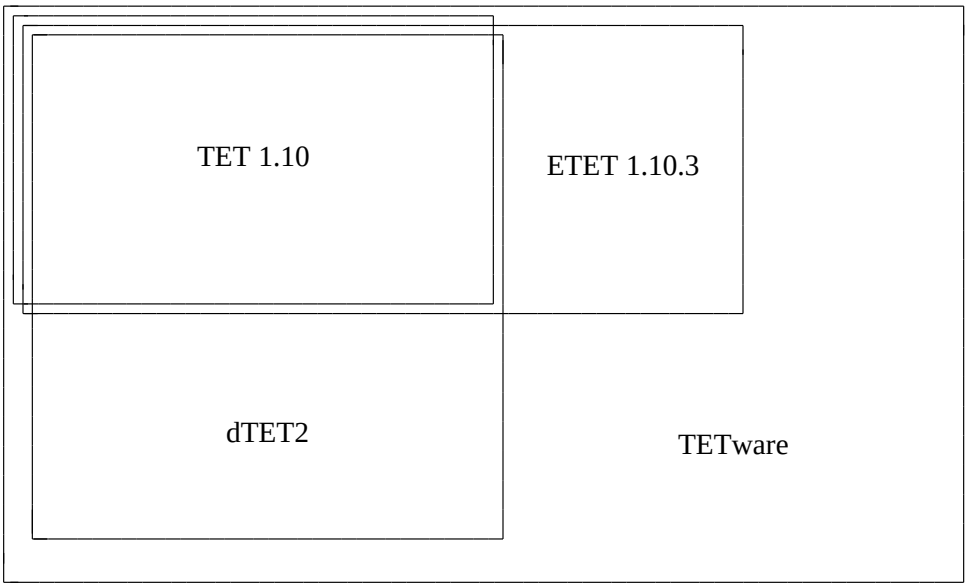

Figure 25. Relationship between TETware and its predecessors

H. Terminology

H.1 Test case types

Certain terms described here are used throughout the TETware documents to describe the different types of test case that may be executed by TETware.

A **local test case** is one that executes on the local system; that is, the system on which the Test Case Controller `tcc` is executed.

A **remote test case** is one that executes on a system other than the one on which `tcc` is executed. `tcc` collects the test case's execution results file output from the remote system and includes it in the journal file on the local system. Although it is possible for several remote test cases to execute concurrently on different remote systems, the test harness does not provide for interaction between remote test cases.

A **distributed test case** is one that has several parts; these parts execute concurrently on different systems. When a distributed test case is being executed, the test harness ensures that each test purpose part starts at the same time on each system. Thus each part of a particular distributed test case must always contain identical number of invocable components and test purposes, even if this means that some of the test purpose parts do nothing. It is likely that parts of a distributed test purpose will interact with each other in some way during the course of their execution. In particular, the test harness provides a means by which the different parts of a test purpose may synchronise with each other. Each test purpose part submits a result which indicates the success or failure of that part of the test purpose. The test harness arbitrates between the results submitted by the parts of the test purpose that are executing on each system and enters a single consolidated result in the journal file.

Distributed TETware can process all of these types of test case. TETware-Lite can only process local test cases.

H.2 Glossary

The following terms are used throughout this document.

Alternate execution directory

A directory specified by the user below which test case execution is to occur. When such a directory is specified, it is the responsibility of the build tool to copy test case files from the test case source directory to their location below this directory.

Application programming interface (API)

An application programming interface is the set of software interfaces between an application and the system. In the case of TETware, the API libraries offer specific facilities for use by test cases.

API-conforming test case

A test case that uses one of the TETware APIs. In particular, the test case uses the API to report test results.

Build file

A build file is a set of instructions passed to the build tool. The provision of a build file is optional.

Build tool

The build tool controls how test cases are prepared for execution, such as in creating binaries from source files. If configuration variables are needed by the build tool, it must use the TETware API. A test suite must define a build tool in order to enable TETware to process it in build mode.

Build fail file

A build fail file is a set of instructions passed to the build fail tool. The provision of a build fail file is optional.

Build fail tool

A test suite defined utility which is executed by the test case controller when the prebuild tool or build tool cannot be executed or returns non-zero exit status. The provision of a build fail tool is optional.

Clean file

A clean file is a set of instructions passed to the clean tool. The provision of a clean file is optional.

Clean tool

A clean tool controls how files or conditions created for or during execution of the test cases are removed, such as removing binary versions of test cases and any object files that were built when the binaries were created. If configuration variables are needed by the clean tool, it must use the TETware API. A test suite must define a clean tool in order to enable TETware to process it in clean mode.

Communication variable

Communication variables are environment variables that are used by the Test Case Controller to provide information to the build tool, clean tool, and test cases during execution. The names of communication variables all start with the prefix `TET_`.

Configuration variable

Configuration variables are used to change the execution behaviour of the TCC and the tools that it executes. Configuration variables are set via configuration variable files and via the TCC user interface. The names of variables used by TETware all begin with the prefix `TET_`.

Configuration variables may also be used to pass parameters to API-conforming test cases and tools. Test suite authors are cautioned to use obvious and consistent naming conventions to avoid potential conflicts with other configuration variables.

Distributed configuration variable

In Distributed TETware, distributed configuration variables are used to inform TCC of the location of test suite files and directories on remote systems. In addition, when Distributed TETware is built to use XTI as the network transport, distributed configuration variables are used to specify certain parameters needed by the transport-specific code.

Distributed test case

Refer to the description presented in the previous section.

Distributed testing

Within the context of TETware, this term refers to the processing of distributed test cases. It does not refer to the processing of non-distributed test cases on remote systems.

Exec file

An exec file is a set of test suite defined instructions for use in executing test cases under control of an exec tool.

Exec tool

A tool used for executing test cases under special control; for example: a debugger or command interpreter. Normally no exec tool is specified, which means that test cases are executed directly.

Execution results daemon

In Distributed TETware, the server used by the API to manage execution results files on behalf of test cases. The name of this server is `tetxresd`.

Execution results file

An API-conforming test case or tool places results and other journal information into the execution results file. A non-distributed test case each has its own execution results file. When Distributed TETware executes a distributed test case, all parts of the test case share a single execution results file. The TCC transfers the contents of the execution results file to the journal when processing of each test case finished.

Invocable component

An invocable component is the smallest unit that the TCM can execute individually. Invocable components are made up of one or more test purposes.

Journal

A journal is the file into which test results and tracking data are deposited by the TCC. This file may be processed by a report generator and/or test suite supplied treatment filter to create formatted reports of test results.

Local system

The system from which the building, execution and cleaning of the tests is controlled. This system contains the test scenario for a particular TCC invocation and (when test cases are to be processed on remote systems) transmits information to each of the remote systems in order that they undertake the necessary tasks as specified in the scenario file. Each Distributed TETware invocation has exactly one local system and zero or more remote systems. A TETware-Lite invocation only has a local system and no remote systems.

Local test case

Refer to the description presented in the previous section.

Master configuration

The master configuration for a particular mode of operation is constructed by reading configuration variables from a user-supplied file on the local system and adding variables defined on the TCC command line. In TETware-Lite this is the only configuration for a particular mode of operation. In Distributed TETware, the master configuration is used in conjunction with variables defined in configuration files on each remote system to generate each of the per-system configurations for a particular mode of operation.

Master system

In Distributed TETware, each test case is processed on one or more systems specified in a system list. This list may be specified by certain scenario directives. If no such list is specified, it defaults to a list containing a single entry for the local system. The master system is the first (or only) system in the list. Note that in TETware the meaning of this term is different from that defined in previous TET implementations.

Mode of operation

When the TCC processes test cases, it does so in one or more modes of operation. These modes are: build mode, execute mode and clean mode. The selected mode(s) of operation are specified for each TETware invocation by options on the `tcc` command line. At least one mode of operation must be selected for each TETware invocation.

Non API-conforming test case

A test case that does not use one of the TETware APIs. TETware deduces the result of this type of test case from the test case's exit status.

Output capture mode

When this mode is enabled, the TCC executes each test case or tool with standard output and standard error directed to a temporary file. TETware copies the contents of this file to the journal when the test case or tool finishes execution.

Per-system configuration

In Distributed TETware, the per-system configuration contains variables which are specific to that system for a particular mode of operation.

Prebuild file

A prebuild file is a set of test suite defined instructions to the prebuild tool for use in preparing for the building of executable versions of test cases.

Prebuild tool

When a prebuild tool is defined, the TCC uses it to undertake the preparation for the build operation. When Distributed TETware processes remote or distributed test cases on more than one system, the prebuild phase is only performed on the master system.

Remote system

In Distributed TETware, a system on which test cases are processed other than the system on which the TCC is invoked. A Distributed TETware invocation may control test cases on one or more remote systems.

Remote test case

Refer to the description presented in the previous section.

Result code

A result code is the determination made by a test purpose as to the status of the test it performed. TETware supports the result codes defined by IEEE Std 1003.3–1991, as well as additional, user-defined result codes. The status of test cases, represented by result codes, is recorded by the API in the journal and can be analysed by the report generator and appropriate treatment filters.

Runtime directory

When a runtime directory is specified, the TCC copies the directory hierarchy below the test suite root directory to a location below the runtime directory before processing the test suite. This location then becomes the new test suite root directory for that particular TCC invocation.

Scenario file

A scenario file is a file containing test scenario definitions.

SGS Software generation system.

Slave system

In Distributed TETware, each test case is processed on one or more systems specified in a system list. This list may be specified by certain scenario directives. If no such list is specified, it defaults to a list containing a single entry for the local system. When the system list contains more than one entry, the slave systems are defined by the second and subsequent entries in the list. Note that in TETware the meaning of this term is different from that defined in previous TET implementations.

Software generation system

The set of tools and other files that are used to compile programs on a particular system. This set includes (at least) the compiler and linker, archive maintainer, header and library files.

SYNCD

The Synchronisation daemon `tetsyncd`.

Synchronisation

In Distributed TETware, the process of ensuring that each part of a distributed test case has reached an agreed point in its execution. Certain synchronisation points are negotiated automatically by the TCMs (for example: at test purpose start) while others are defined by the test suite author and occur during test purpose execution.

Synchronisation daemon

In Distributed TETware, the server used by the API to manage the synchronisation process. The name of this server is `tetsyncd`.

System ID

Distributed TETware systems are identified by a three-digit system identification. The system IDs are mapped to some information (such as a host name) in the `systems` file which may be used to establish a connection with the TCCD on that system. The exact format of this mapping is transport dependent.

The local system always has system ID `000`. Other system IDs refer to remote systems.

TCC The Test Case Controller `tcc`.

TCCD

The Test Case Controller daemon `tccd` or `in.tccd`.

TCM The Test Case Manager.

Test case

A test case is the software which conducts tests. The scope of the term ‘‘test’’ is broad. It may range from a single test purpose for a single function being tested, all the way to a complete suite of conformance tests for a specification. The TCC builds test cases when invoked in build mode, executes the invocable components within test cases when in execute mode, and cleans up any unwanted files when in clean mode.

Test case controller (TCC)

The TCC is the tool that provides structure and control for test cases. The tool handles such functions as sequencing of invocable component execution, unexpected event handling, cleanup, parameter passing and transferring of test case execution results into the journal.

Test case controller daemon (TCCD)

When the Distributed version of the TCC wants to perform some action while processing a test case, it does not perform the action itself but instead instructs a TCCD to perform the action on a particular system. This separation of the control logic from processing actions enables Distributed TETware to control test case processing on an arbitrary number of systems from a single TCC invocation.

The name of this server is `tccd`. On UNIX systems where this server is run under control of `inetd`, the name of this server is `in.tccd`.

Test case execution directory

The directory in which a test case is executed. When an alternate execution directory is specified, the test case execution directory is below the alternate execution directory; otherwise, the test case execution directory is the same as the test case source directory.

Test case manager (TCM)

The TCM is a component of each TETware API. This component acts as a ‘‘wrapper’’ for test cases, providing interpretation of the command line, selection of invocable components, and support for the automatic sequencing of test purposes and invocable components, as well as insulation from spurious signals.

Test case source directory

The directory which contains the source files for a particular test case. It is usual to have a separate source directory for each test case in all but the smallest of test suites.

Test case processing

The action performed by the TCC on a test case which depends on TCC’s selected mode(s) of operation. That is: the test case is built when build mode is selected, executed when execute mode is selected and cleaned when clean mode is selected.

Test purpose

A test purpose is the software that represents the smallest level of granularity of a test specification. A test purpose always leads to a single result. In the case of an IEEE Std 1003.3–1991 conforming test suite, for example, test purposes would correspond to assertions.

Test scenario

A test scenario is a sequence of one or more invocable components associated with a single user-exposed name. When the TCC is invoked with a scenario name, all invocable

components associated with it are built, executed, and/or cleaned depending on the TCC mode selected.

Test suite

A test suite is a set of test case files and other required and optional files that are used by TETware when processing test cases. A test suite must contain at least one test case.

Test suite installer

The test suite installer is used to execute an installation tool supplied with the test suite. TETware does not provide this tool; instead test suite authors are responsible for providing and documenting the installation procedures.

Test suite root directory

The top of the directory subtree which contains the test suite. Usually, this directory resides immediately below the **tet root** directory.

Tet root directory

The top of the directory subtree in which TETware resides.

Thread-safe API

An API which is designed for use in a multi-threaded environment.

Win32 system

A computer system on which the WIN32 API is implemented, such as the Windows NT and Windows 95 operating systems.

XRESD

The Execution Results daemon `tetxresd`.