Test Environment Toolkit

TETware User Guide
Revision 1.1
TET3-UG-1.1

Released: 31st July 1997

X/Open Company Limited

This document is produced by UniSoft Ltd. at:

150 Minories
LONDON
EC3N 1LS
United Kingdom

CONTENTS

1. Introduction

1.1 Preface

This document is the TETware User 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 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 Audience

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

Test suite authors should refer to the TETware Programmers Guide for information about how to use the TETware Application Program Interface.

1.3 Conventions used in this guide

The following typographic conventions are used throughout this guide:

- `Courier font` is used for function and program names, literals and file names. Examples and computer-generated output are also presented in this font.
- The names of variables are presented in *italic font*. You should substitute the variable's value when typing a command that contains a word in this font.
- **Bold font** is used for headings and for emphasis.

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.

1.4 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 Programmers 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.

1.5 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. TETware overview

2.1 Introduction

The purpose of TETware is to provide a uniform framework, or test scaffold, into which both non-distributed and distributed test suites can be incorporated. By providing such a scaffold, test suites from different vendors can share a common interface allowing for, among other things, ease of portability.

In this context, and throughout this guide, the term **non-distributed test** means a test which executes on a single computer system. A non-distributed test can execute on the local system (that is: the system on which the TETware Test Case Controller is executing) or on a remote system (that is: a system other than the one on which the TETware Test Case Controller is executing).

Likewise, the term **distributed test** means a test which consists of several parts; each test part executes concurrently on a different computer system and contributes to the overall test result. Typically, distributed tests are used to verify some kind of interaction between two or more systems.

2.2 Test suite structure

A **test suite** is the largest grouping of tests that can be processed by the TETware Test Case Controller.

A test suite is made up of one or more **test cases**. A test case is the smallest test program unit that can be built or cleaned up by the Test Case Controller.

A test case consists of one or more **invocable components**. An invocable component is the smallest test program unit that can be executed by the Test Case Controller.

An invocable component consists of one or more **test purposes**. A test purpose typically tests an individual element of system operation for conformance to some statement of required behaviour, and yields a **result** indicating whether or not the element passed the test. Often, an invocable component contains a single test purpose.

A **distributed test purpose** is a test purpose which consists of several parts and each part executes on a different computer system. Each part of a distributed test purpose submits a **partial result** which indicates the success or failure of that part of the test purpose. These partial results are gathered from the various parts of the test purpose and, after arbitration between the partial results, a **consolidated result** is generated for the test purpose which appears in the test case journal file. A description of the way in which this arbitration is performed is presented in the section entitled ‘‘Making journal entries’’ in the TETware Programmers Guide.

A **test scenario** is a list of one or more invocable components from a test suite that are processed by a particular Test Case Controller invocation. A test suite often has a scenario named **all** associated with it; this scenario simply lists all the invocable components in the test suite.

2.3 TETware versions

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 local 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.

2.4 Components

TETware-Lite includes the following components:

- A Test Case Controller providing support for the building, execution, and clean-up of test scenarios. The name of this component is `tcc`[1].
- Test Case Managers and Application Programming Interface libraries which can be used to build test cases written in C, C++, Shell, Korn Shell and Perl. These components are not executable programs but are instead linked or otherwise included in each user-supplied test case.

  The names of these components are as follows[2]:

`tcm.o`	C Test Case Manager
`tcmchild.o`	C API child process interface module
`libapi.a`	C API function library
`thrtcm.o`	Thread-safe C Test Case Manager[3]
`thrtcmchild.o`	Thread-safe C API child process interface module
`libthrapi.a`	Thread-safe C API function library
`Ctcm.o`	C Test Case Manager
`Ctcmchild.o`	C++ API child process interface module
`Cthrtcm.o`	Thread-safe C++ Test Case Manager
`Cthrtcmchild.o`	Thread-safe C++ API child process interface module
`tcm.sh`	Shell Test Case Manager
`tetapi.sh`	Shell API function library
`tcm.ksh`	Korn Shell Test Case Manager
`tetapi.ksh`	Korn Shell API function library
`tcm.pl`	Perl Test Case Manager
`api.pl`	Perl API function library

1. On Win32 systems, executable program files have the suffix `.exe`.
2. The `.o` and `.a` suffixes shown here are the ones that are used on UNIX operating systems. On Win32 systems the `.o` files instead have the suffix `.obj` and the `.a` files instead have the suffix `.lib`. All the other suffixes are the same on each operating system.
3. On UNIX systems the thread-safe components can be built to support either POSIX threads or UI threads (but not both at the same time).
   On Win32 systems the thread-safe components are built for use with the multi-threaded DLL version of the C runtime support library.

In addition to the components described for TETware-Lite, Distributed TETware includes the following components:

- A Test Case Controller daemon which performs test case processing actions on behalf of the Test Case Controller (`tccd`).
- A Test Case Controller daemon bootstrap program for use on Windows NT systems (`tccdstart`).
- A Synchronisation daemon which handles synchronisation requests from different parts of a distributed test case (`tetsyncd`).
- An Execution Results daemon which handles journal entries made by test cases (`tetxresd`).
- The C API includes a remote executed process interface module (`tcmrem.o`).

In Distributed TETware, the C and C++ APIs may be used to build both distributed and non-distributed test cases. The other APIs may be used to build non-distributed test cases which may, nevertheless, be processed on both the local system and on remote systems.

The distributed Test Case Controller can process test cases written to use either the Distributed or Lite versions of all the APIs. However, the TETware-Lite Test Case Controller cannot process test cases written to use the Distributed versions of the C and C++ APIs.

Although Distributed TETware is a superset of TETware-Lite, the way in which the Test Case Controller operates is rather different in the two TETware versions, as follows:

- In TETware-Lite, the Test Case Controller itself performs all the actions that are required to process test cases.
- By contrast, Distributed TETware uses a client-server architecture. The distributed version of the Test Case Controller does not itself perform the actions required to process test cases. Instead, it issues requests to servers running on each participating system which perform the required actions in its behalf. Thus the distributed version of the Test Case Controller can process test cases on the local system, on one or more remote systems, or on some combination of the two.

Reference manual pages for TETware programs and file formats are presented in an appendix to this guide. Test suite authors should refer to the TETware Programmers Guide for information about the TETware APIs.

2.5 Features and facilities

TETware provides facilities to execute test cases in several ways as follows. All these facilities are available in Distributed TETware. Facilities marked with a † are not available in TETware-Lite.

- Execution of non-distributed test cases on the local system (i.e., local test cases).
- Execution of non-distributed test cases on a single remote system (i.e., remote test cases).†
- Concurrent execution of non-distributed test cases on several remote systems.†
- Execution of distributed test cases with the parts of each test case executing simultaneously on either the local system and one or more remote systems, or entirely on two or more remote systems.†

- Execution of a single test case selected at random from a list of test cases.
- Combinations of the above elements executing in parallel.
- Sequences of the above elements executing a specified number of times or until some time period has expired.

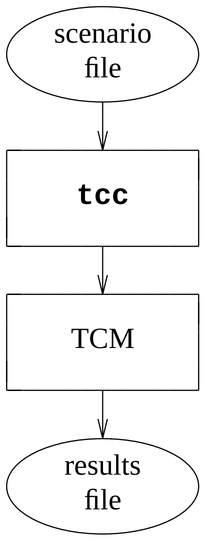
2.6 Simple TETware architecture diagrams

2.6.1 Introduction

The diagrams presented in this section provide a simplified view of how the different TETware components relate to each other. More complete diagrams are presented in the appendix entitled ‘‘Conceptual models used by TETware’’ in the TETware Programmers Guide.

2.6.2 TETware-Lite architecture

The following diagram provides a simplified view of how the different components relate to each other in TETware-Lite:

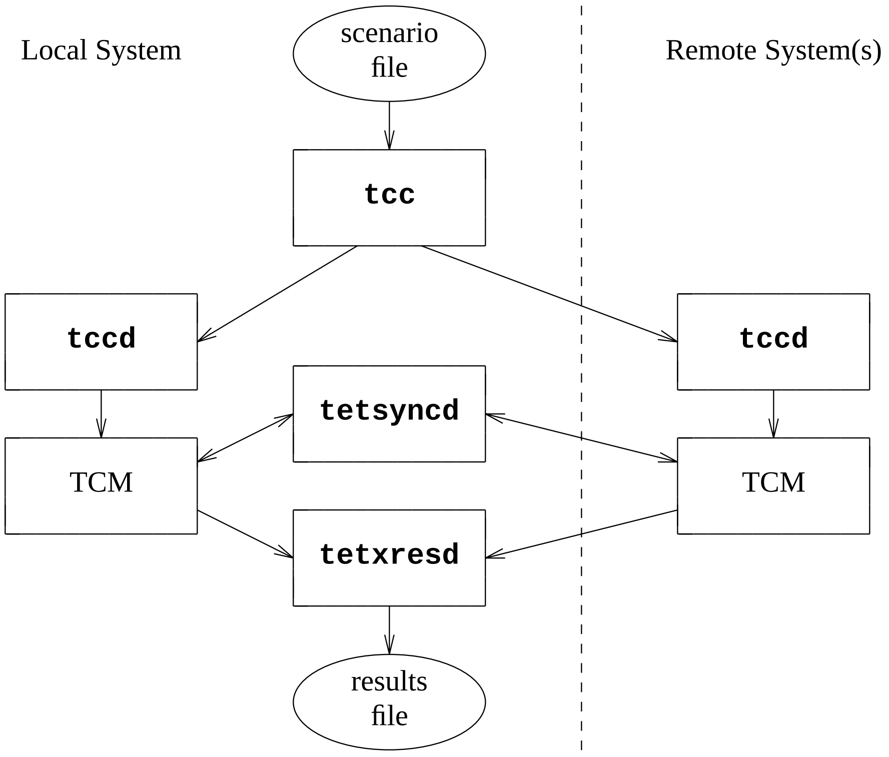


`tcc`	–	TETware-Lite Test Case Controller
TCM	–	Test Case Manager + test case parts

It will be seen that all the processing takes place on a single system (the local system) and that a client/server architecture is not used. Thus, remote and distributed test cases cannot be processed by TETware-Lite.

2.6.3 Distributed TETware architecture

The following diagram provides a simplified view of how the different components relate to each other when a distributed test case is executed by Distributed TETware:



tcc	– Distributed TETware Test Case Controller
tccd	– Test Case Controller daemon
tetsyncd	– Synchronisation daemon
tetxresd	– Execution results daemon
TCM	– Test Case Manager + test case parts

Processing is similar when Distributed TETware executes a non-distributed test case; the main difference is that the Synchronisation daemon `tetsyncd` does not participate in this type of processing.

It will be seen from this diagram that, unlike dTET2, the architecture used by Distributed TETware is fully symmetrical. That is, there is no longer a distinction between master and slave systems when test cases are executed. `tcc` does not itself perform test case management functions but instead requests `tccd` to do so on its behalf. Consequently it does not matter whether a (non-distributed) test case or a (distributed) test case part is processed on the local system[4] or on a remote system; the processing logic is the same in each case.

4. That is: the system on which `tcc` runs.

2.7 Distributed TETware systems and network options

When the Distributed TETware version of `tcc` processes test cases, it may do so on a **local system** and/or one or more **remote systems**. This is illustrated by the diagram presented in the previous section which shows that several server processes take part in this processing.

Although it is usual for each TETware system to reside on a different physical machine, the way that systems are defined makes it possible for more than one logical TETware system to map to a physical machine. Indeed, it would be possible for all the logical systems participating in a distributed test case to coexist on a single physical machine if such a configuration were to be required. The way that the mapping of logical systems to physical machines is performed is described in the chapter entitled ‘‘Using TETware’’ elsewhere in this guide.

TETware server processes communicate with each other using some kind of network interface. Support is provided for several different network interfaces and the decision as to which one to use must be made before TETware is built. This is described further in the chapter entitled ‘‘Building TETware’’ in each of the platform-specific versions of the TETware Installation Guide.

2.8 Supported operating system types

TETware-Lite has been implemented for use on computers which run UNIX operating systems and also on computers which run the Windows NT and Windows 95 operating systems. Distributed TETware has been implemented for use on computers which run UNIX operating systems and also on computers which run the Windows NT operating system.

The design of TETware and its predecessors has been influenced to a large extent by the facilities that are available on UNIX systems. While care has been taken to ensure that the UNIX and Win32 implementations operate in as similar a way as possible, it is inevitable that certain differences exist between them. These differences are discussed further in the appendix entitled ‘‘Implementation notes for TETware on Win32 systems’’ at the end of this guide.

3. Installing TETware

There is one version of the TETware Installation Guide for each family of operating systems on which TETware is implemented. For details of how to build and install TETware, please refer to the version of the TETware Installation Guide that is appropriate for your computer system.

4. Running the TETware demonstrations

4.1 Introduction

This chapter describes how to run the TETware demonstration test suites. Before you can run the demonstrations, you should install TETware on all the systems that you want to use to run local, remote or distributed tests.

There are several demonstration test suites included with TETware as follows:

TETware demonstration test suites		
Name	**Path**	**Description**
`demo`	*tet-root*`/contrib/`	Simple demonstration test suite for the C API. This is documented in the TET Programmers' Guide.
`cplusdemo`	*tet-root*`/contrib/`	Simple demonstration test suite for the C++ API. This is the C API demonstration suite modified to use the C++ API.
`perldemo`	*tet-root*`/contrib/`	Demonstration test suite for the Perl API. This is a Perl API demonstration suite similar to the simple C API demonstration.
`SHELL-API`	*tet-root*`/contrib/`	Demonstration test suite for the Shell API. This is a Shell API demonstration suite.
`demo`	*tet-root*`/src/tet3/`	Distributed demonstration test suite for the C API. This is the official demonstration test suite for Distributed TETware.

Note that the contributed demonstrations below *tet-root*`/contrib` have been designed by their contributors to work on UNIX systems and it is likely that a small amount of porting effort will be required in order to cause these demonstrations to install and/or function correctly on a Win32 system. In most cases this effort includes (at least) the modification of the build, clean and installation mechanisms in order to cater for the different file names and suffixes which must be used on Win32 systems.

4.2 System requirements

The distributed demonstration must be run using Distributed TETware. The other demonstrations can be run using either Distributed TETware or TETware-Lite.

You should follow the instructions in **one** of the following subsections at this level, depending on which version of TETware you have installed.

4.2.1 Using TETware-Lite

There is no system configuration required when you run the demonstrations using TETware-Lite. Note that since TETware-Lite is not able to process distributed test cases, it cannot be used to run the distributed demonstration.

4.2.2 Using Distributed TETware

4.2.2.1 Configuration

If you have installed Distributed TETware, you will need to make use of two machines connected to the same local area network in order to run the distributed demonstration program. When the distributed demonstration is run, one of these machines will act as the local (or **master**) system and the other will act as the remote (or **slave**) system. The master system is defined as the one on which you will run `tcc`.

Although the simple C, Perl and C++ demonstrations can be run on a single machine, they still require the following setup when run with Distributed TETware.

You must perform the following actions:

1. Ensure that Distributed TETware is installed on each system that will participate in the demonstrations.
2. Verify that the `tccd` daemon is running on the system(s) and that it has permission to access and modify both the **tet root** directory and the **test suite root** directory hierarchy.
3. Ensure that you know the name of the *tet-root* directory on each system.
4. If you have installed the version of Distributed TETware which uses the socket network interface, you should ensure that an entry for each participating system exists in the `systems.equiv` file on each system. On a UNIX system this file is located in the `tet` user's home directory. On a Windows NT system this file is located in the directory specified by the value of the `HOME` environment variable which is in effect when `tccd` is started, or in `tccd`'s current working directory if no `HOME` variable is present.

4.2.2.2 The `systems` file

Before you can run the demonstration programs, you must customise the `systems` file.

In order to do this, you should log in to the master system and change directory to *tet-root*. The file *tet-root*`/systems` contains the mappings that assign system identifiers to host names. An example systems file is supplied with the demonstration in *tet-root*`/src/tet3/demo/systems` which contains lines similar to the following:

```
#       Example system file for demonstration
000     master
001     slave
```

You should copy this file to *tet-root*`/systems` on the master system. Then, edit the copy, replacing `master` with the host name of the master system and `slave` with the host name of the slave system. You should ensure that these host names are in the host database on both the master and the slave system (often in the file `/etc/hosts` on UNIX systems).

Once you have customised the `systems` file on the master system, you should copy it to the **tet root** directory on the slave machine.

4.3 The C API demonstration

This section describes how to run the simple demonstration TET test suite in *tet-root*`/contrib/demo`. This is the standard TET 1.10 demonstration. This test suite executes only on a single system and does not require remote systems to be set up.

The following instructions will cause the test suite to be built, executed and cleaned:

1. To start, if you are not already there, change to the directory in which the demonstration test suite resides, thus:

   ```
   cd tet-root/contrib/demo
   ```

2. Make sure that *tet-root*`/bin` is in your `PATH`.

3. Set your `TET_ROOT` environment variable to refer to *tet-root*, thus:

   ```
   TET_ROOT=tet-root
   export TET_ROOT
   ```

4. Review the contents of `ts/makefile` and ensure that it is correct for your system.

5. Type the following command to run the demonstration:

   ```
   tcc -bec contrib/demo
   ```

As the demonstration executes, it displays the following message:

```
Journal file is: tet-root/contrib/demo/results/0001bec/journal
```

You can then look in *tet-root*`/contrib/demo/results/0001bec/journal` to see the results of the demonstration. Note that, when using Distributed TETware, standard output from test cases is redirected to the `tccd` log file (defaults to `/tmp/tccdlog`).

A sample filter to generate a results summary is contained in *tet-root*`/contrib/usltools/vres`. This filter is an `awk` script. Type the following commands in order to install and run this script:

```
cp $TET_ROOT/contrib/usltools/vres $TET_ROOT/bin/vres
chmod a+x $TET_ROOT/bin/vres
vres results/0001bec/journal
```

When run, `vres` produces the following output:

```
Results: results/0001bec/journal
Total tests:            3       PASS = 3        FAIL = 0

Pass Breakdown:
Number of successes:    3       Number of warnings:     0
Number unsupported:     0       Number not in use:      0
Number of untested:     0       Number of FIP:          0

Failure Breakdown:
Number of failures:     0       Number unresolved:      0
Number uninitiated:     0       Number unreported:      0
```

An alternative execution scenario file is provided to demonstrate some of the ETET-derived features of TETware. The name of this file is *tet-root*`/contrib/demo/etet_scen` and its contents are as follows:

```
#       Demonstration test suite.
#       A default execution scenario to demo the ETET features
tests
        "starting scenario"
        /ts/tc1
        /ts/tc2
        "next is the last test case"
        /ts/tc3
        "done"

all
        "pick a random test"
        :random:^tests
        "repeat four times /ts/tc1"
        :repeat,4:@/ts/tc1
        "run all the tests in parallel"
        :parallel:^tests
        "pick random tests for 35 seconds"
        :timed_loop,35;random:^tests
```

The following commands will build and execute the test suite using the ETET-derived features:

```
tcc -b contrib/demo
tcc -e -p -v TET_COMPAT=etet -s etet_scen contrib/demo
```

4.4 The C++ API demonstration

This section describes how to run the simple C++ demonstration TET test suite in *tet-root*`/contrib/cplusdemo`. This test suite executes only on a single system and does not require remote systems to be set up.

The following instructions will cause the test suite to be built, executed and cleaned:

1. To start, if you are not already there, change to the directory in which the demonstration test suite resides, thus:

   ```
   cd tet-root/contrib/cplusdemo
   ```

2. There is a makefile in each test case directory below the `ts` directory. Review the contents of each makefile and ensure that it is correct for your system.

3. Make sure that *tet-root*`/bin` is in your `PATH`.

4. Set your `TET_ROOT` environment variable to refer to *tet-root*, thus:

   ```
   TET_ROOT=tet-root
   export TET_ROOT
   ```

5. Type the following command to run the demonstration:

   ```
   tcc -bec contrib/cplusdemo
   ```

As the demonstration executes, it displays the following message:

```
journal file name is: tet-root/contrib/cplusdemo/results/0001bec/journal
```

You can then look in *tet-root*`/contrib/cplusdemo/results/0001bec/journal` to see the results of the demonstration.

4.5 The Perl API demonstration

This section describes how to run the simple Perl demonstration TET test suite in *tet-root*`/contrib/perldemo`. This test suite executes only on a single system and does not require remote systems to be set up.

The following instructions will cause the test suite to be built, executed and cleaned:

1. To start, if you are not already there, change to the directory in which the demonstration test suite resides, thus:

   ```
   cd tet-root/contrib/perldemo
   ```

2. Review the contents of `ts/makefile` and ensure that it is correct for your system.
3. Make sure that *tet-root*`/bin` is in your `PATH`.
4. Set your `TET_ROOT` environment variable to refer to *tet-root*, thus:

   ```
   TET_ROOT=tet-root
   export TET_ROOT
   ```

5. Type the following command to run the demonstration:

   ```
   tcc -bec contrib/perldemo
   ```

As the demonstration executes, it displays the following message:

```
journal file name is: tet-root/contrib/perldemo/results/0001bec/journal
```

You can then look in *tet-root*`/contrib/perldemo/results/0001bec/journal` to see the results of the demonstration.

4.6 The Shell API demonstration

This section describes how to run the SHELL-API demonstration TET test suite in *tet-root*`/contrib/SHELL-API`. This test suite is deliberately simple and tests the user-level commands `uname` and `chmod`. Sample tests include checking a returned error code and error message against and expected error code and error message, and printing out system specific information for verification by the tester.

This test suite executes only on a single system and does not require remote systems to be set up.

The following instructions will cause the test suite to be built and executed:

1. To start, if you are not already there, change to the directory in which the demonstration test suite resides, thus:

   ```
   cd tet-root/contrib/SHELL-API
   ```

2. Review the contents of the `README` file.
3. Make sure that *tet-root*`/bin` and *tet-root*`/contrib/SHELL-API/bin` are in your `PATH`.

4. Set your `TET_ROOT` environment variable to refer to *tet-root*, thus:

   ```
   TET_ROOT=tet-root
   export TET_ROOT
   ```

5. Make sure that the commands in *tet-root*`/contrib/SHELL-API/bin` are executable, thus:

   ```
   chmod a+x bin/buildtool bin/cleantool bin/install
   ```

6. Type the following command to install the suite:

   ```
   ./bin/install
   ```

7. Type the following command to build the suite:

   ```
   tcc -p -b -a ‘pwd‘/ts_exec contrib/SHELL-API
   ```

8. Type the following command to execute the suite:

   ```
   tcc -p -e -a ‘pwd‘/ts_exec contrib/SHELL-API
   ```

As the demonstration executes, it displays the following messages:

```
tcc: journal file is tet-root/contrib/SHELL-API/results/0002e/journal
10:21:27  Execute /ts/chmod/chmod-tc on system 000
10:21:31  Execute /ts/uname/uname-tc on system 000
```

You can then look in *tet-root*`/contrib/SHELL-API/results/0002e/journal` to see the results of the demonstration.

4.7 The Distributed C API demonstration

4.7.1 Introduction

The TETware distributed demonstration program is a simple test suite consisting of three distributed test cases that are built, executed and cleaned on each system participating in the test. You may run the demonstration program to verify that Distributed TETware has installed correctly.

This 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.

The instructions presented here for building and installing the demonstration on a Windows NT system assume that you are using the same defined build environment that is used when TETware is built on such a system; that is: Microsoft Visual C++ and the MKS Toolkit. Since the demonstration has been designed for use in this environment on a Windows NT system, it is likely that you will have to make changes to the source code and/or the installation method in order to use a different environment. For further details of the defined build environment, refer to the section entitled ‘‘System requirements’’ in the TETware Installation Guide for the Windows NT and Windows 95 Operating Systems.

It will be seen that all the parameters whose values depend on the type of host operating system are specified either in the makefiles or in the configuration files which reside on each system. This demonstration provides an example of one way in which TETware features may be used in order to maintain maximum test case portability between the two types of operating system.

4.7.2 Installation

You must perform the following actions to install the demonstration program on each system:

1. Log on to the system that you want to use as the master system.

2. Copy the master part of the demonstration to *tet-root*/demo on the master system, thus:

   ```
   cd tet-root/src/tet3/demo/master
   find . -print | cpio -pdv tet-root/demo
   ```

3. Log on to the system that you want to use as the slave system.

4. Copy the slave part of the demonstration to the **test suite root** directory on the slave system, thus:

   ```
   cd tet-root/src/tet3/demo/slave
   find . -print | cpio -pdv tet-root/demo
   ```

4.7.3 Configuration

4.7.3.1 Introduction

Before you can run the demonstration programs, you must customise the systems file, as described earlier in this document, and also the distributed configuration file tetdist.cfg. In addition, if either part of the demonstration is running on a Windows NT system, you will need to customise the tetbuild.cfg and tetclean.cfg files on the master system and (optionally) the slave system.

In order to do this, you should log in to the master system and change directory to *tet-root*. Then, you should work through the instructions presented in the following subsections.

4.7.3.2 The systems file

The file *tet-root*/systems contains the mappings that assign system identifiers to host names. An example systems file is supplied with the demonstration in *tet-root*/src/tet3/demo/systems which contains lines similar to the following:

```
#       Example system file for demonstration
000     master
001     slave
```

You should copy this file to *tet-root*/systems on the master system. Then, edit the copy, replacing master with the host name of the master system and slave with the host name of the slave system. You should ensure that these host names are in the host database on both the master and the slave system (often in the file /etc/hosts).

If you have built TETware on the master system to use the XTI network interface, you must add a third field to the entry for each system. This field should contain the XTI address string of the Test Case Controller daemon tccd that is running on that system. The format of XTI address strings is described in the section entitled ‘‘System definitions’’ in the TETware Programmers Guide.

Once you have customised the systems file on the master system, you should copy it to the **tet root** directory on the slave system.

4.7.3.3 The `tetdist.cfg` file

The file *tet-root*`/demo/tetdist.cfg` contains variable assignments for slave systems that are equivalent to those on the master system that may be specified in environment variables. As supplied with the demonstration, this file contains lines similar to the following:

```
TET_REM001_TET_ROOT=/home/tet3/
TET_REM001_TET_TSROOT=/home/tet3/demo

# The following variables are only referenced by XTI based TCCs

TET_XTI_TPI=/dev/tcp
TET_XTI_MODE=tcp
TET_LOCALHOST=89.02.01.00
```

If the **tet root** directory on the slave system is not `/home/tet3`, you must change the first two variables to refer to the **tet root** and **test suite root** directories on the slave system.

The last three variables are only required if TETware on the master system has been built to use the XTI network interface. You must substitute values for these variables which are correct for your system. Note that the value that you specify for `TET_LOCALHOST` must be the machine's external IP address and not the address of the loopback interface.

4.7.3.4 The `tetbuild.cfg` and `tetclean.cfg` files

As distributed these files contain values which are appropriate when you run the demonstration on two UNIX systems. In order to accommodate different file name suffixes, you must edit 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 these files on both the master and the slave system.

4.7.3.5 Makefiles

You must check the test case makefiles on each system to ensure that they will build the test cases correctly.

You may need to add the names of libraries to the `SYSLIBS` variable in order to resolve external function names used in the TETware Test Case Manager and API. For example, if you have built TETware to use the socket network interface, you may need to append `−lsocket` and `−lnsl` to this variable on some systems.

If you are running either part of the distributed demonstration on a Windows NT system, you will need to customise the file name suffix variables in the makefiles as well as the `SYSLIBS` variable. Details of the changes that you must make are presented in comments contained in the makefiles.

4.7.4 Running the distributed demonstration

Once you have installed and configured the distributed demonstration to run on your systems, you are ready to run the demonstration.

You should perform the following operations on the master system to run the demonstration program:

1. Log in to the master system and change directory to *tet-root*.

2. Make sure that the setting of your `PATH` environment variable includes *tet-root*`/bin`.
3. Set your `TET_ROOT` environment variable to refer to *tet-root*, thus:

   ```
   TET_ROOT=tet-root
   export TET_ROOT
   ```

4. Type the following command to run the demonstration program:

   ```
   tcc -bec demo
   ```

   When `tcc` starts, it will respond with a line similar to the following:

   ```
   Journal file is: tet-root/demo/results/0001bec/journal
   ```

   Then the demonstration test suite will be built, executed and cleaned up.
5. Once `tcc` has finished execution, you can examine the journal file and verify the results of the demonstration. Test cases 1 and 3 are expected to pass, test case 2 is expected to fail.

4.7.5 Problem diagnosis

If the demonstration did not run as expected, it is likely that one or more TETware components will have generated diagnostic messages describing what went wrong. You should check for diagnostic information in locations listed in the following table:

Location of diagnostic information	Source of diagnostic message

`tcc`'s standard error stream
(appears on the terminal unless redirected)
	`tcc`
	`tetsyncd`
	`tetxresd`

Journal file (on the master system)	`tcc` Test Case Managers on each system APIs on each system
`tccd` log file on each system (defaults to `/tmp/tccdlog`)	`tccd` Test Case Managers on each system APIs on each system

5. Using TETware

5.1 Introduction

This chapter describes how to use TETware. Before you can use TETware, you must install TETware on all the systems that you want to use to run local, remote or distributed tests, as described in the TETware Installation Guide.

5.2 TETware concepts

5.2.1 Introduction

This section describes some concepts employed in TETware.

5.2.2 TETware components

The components that make up TETware are described in the chapter entitled ‘‘TETware overview’’ elsewhere in this guide. You should read this chapter now if you have not already done so.

5.2.3 Modes of operation

The Test Case Controller (`tcc`) operates in one of several modes as follows:

build mode	– in this mode, test cases are built
execute mode	– in this mode, invocable components are executed
clean mode	– in this mode, test cases are cleaned; i.e., the test case directory is returned to the state that it was in before the test case was built and/or executed

One or more modes of operation are specified by command-line options each time that `tcc` is invoked.

5.2.4 TETware-Lite

5.2.4.1 Local and remote systems

The Lite version of `tcc` is able to process non-distributed test cases on the local system. It cannot process distributed test cases or test cases on remote systems.

5.2.4.2 Test case types

TETware-Lite is able to process **local** test cases. It cannot process **remote** test cases or **distributed** test cases.

5.2.5 Distributed TETware

5.2.5.1 Local and remote systems

The distributed version of `tcc` is able to process non-distributed test cases on both the local system and on one or more remote systems. In addition, `tcc` is able to process distributed test cases where parts of each test case execute on the local (or **master**) system and/or on one or more remote (or **slave**) systems.

Distributed TETware systems are identified by a three-digit **system ID**. System ID zero refers to the local system and other system ID values refer to remote systems. In this context, the local system is defined as the one on which `tcc` is invoked.

Usually, different TETware systems are located on different physical machines. However, there is no reason why different system IDs should not refer to the same machine.

5.2.5.2 Test case types

Distributed TETware is able to process **local** test cases, **remote** test cases and **distributed** test cases. For a description of what is meant by these different test case types, refer to the chapter entitled ‘‘TETware overview’’ elsewhere in this guide.

5.2.6 TETware directory layout

The whole of TETware resides in a single directory hierarchy on each system that is to participate in local, remote or distributed testing. The top of this hierarchy is known as the **tet root** directory. Below **tet root** there are directories containing TETware executables and documentation.

In addition, directory hierarchies that contain the test suites that are to be used with TETware reside below a single directory per test suite. The top of each test suite directory hierarchy is known as the **test suite root** directory. By default, the test suite root directory resides below the **tet root** directory on each system.

By default, `tcc` looks for test cases below the test suite root directory. However, it is possible to specify an **alternate execution directory** or a runtime directory; if such a directory is specified, `tcc` looks for test cases below that directory instead.

It is possible to ask `tcc` to save files and/or directory subtrees that may be generated during test case processing. When files are saved in this way, they are copied to the **saved files directory**.

An example TETware directory hierarchy is presented in the appendix entitled ‘‘TETware directory structure’’ at the end of this guide.

5.2.7 Environment variables

The following environment variables are used to control the operation of `tcc`. These variables are used by both TETware-Lite and Distributed TETware. When Distributed TETware is used, these variables control the operation of `tcc` on the local system (that is: the system on which `tcc` is run). Configuration variables which perform similar functions for remote systems to those performed by the environment variables described here may be specified in the distributed configuration file `tetdist.cfg`. The meanings of these distributed configuration variables is described further in the section entitled ‘‘Configuration variables used for distributed testing’’ elsewhere in this chapter.

`TET_ROOT`	Specifies the location of the **tet root** directory. This environment variable must always be set when `tcc` is run.
`TET_EXECUTE`	If this variable is set, it specifies the location of the alternate execution directory.
`TET_TMP_DIR`	If this variable is set, it specifies the location of `tcc`’s temporary directory.

`TET_SUITE_ROOT` If this variable is set, it specifies an alternate location below which the test suite root is located.

`TET_RUN` If this variable is set, it specifies the location of a runtime directory. `tcc` copies the entire test suite directory hierarchy to a location below the runtime directory and processes the test suite in that location. This variable is useful when the test suite directory hierarchy resides on a read-only file system or NFS-mounted from a central server whose file system should not be modified by the system under test.

5.2.8 Configuration variables

You may specify sets of configuration variables for each of `tcc`'s modes of operation. Each variable specification takes the following form:

variable=*value*

Some configuration variables alter the behaviour of TETware, whereas others are specific to the test suite being processed and may be accessed by test cases and by build and clean tools.

Configuration variables may be specified in configuration files or by means of a `tcc` command-line option. A configuration variable specified on the `tcc` command line takes precedence over a variable of the same name that is specified in a configuration file.

Configuration variables that alter the behaviour of TETware are described further in the section entitled ‘‘Configuration files’’ later in this chapter.

5.2.9 Test purpose results

Each non-distributed test purpose, and each part of a distributed test purpose, is expected to register exactly one test purpose result code. A **result code** is a numeric value whose meaning is defined in a result code table. Each result code has an **action** associated with it which indicates what action the Test Case Manager (TCM) should take when a test purpose reports that result code. Possible values for this action are `Continue` which means that the TCM should continue to process the next test purpose (if any), or `Abort` which indicates that the TCM should abort the current test case.

The following result codes are defined in TETware:

Result code	Meaning	Action
0	PASS	Continue
1	FAIL	Continue
2	UNRESOLVED	Continue
3	NOTINUSE	Continue
4	UNSUPPORTED	Continue
5	UNTESTED	Continue
6	UNINITIATED	Continue
7	NORESULT	Continue

A test suite may provide a results code file defining these and other testsuite-specific result codes. This file is described further in the section entitled ‘‘The results code file’’ later in this chapter.

5.3 TETware data files

5.3.1 Introduction

This section describes the formats and locations of data files used by TETware.

5.3.2 The scenario file

5.3.2.1 Description

The scenario file contains one or more test scenarios for a test suite. By default, the name of this file is `tet_scen` and is located in the test suite root directory on the local system. (However, a different scenario file name may be specified by means of a `tcc` command-line option; for details, refer to the `tcc` manual page at the end of this guide.)

A scenario file should contain (at least) a scenario named `all`; by convention, this causes all the test cases in the test suite to be processed.

Each entry in a scenario file starts with the **scenario name** and contains one or more **scenario directives** and **scenario elements**. Each scenario name in the file starts at column 1 and lines containing scenario directives and elements have a blank in column 1. Blank lines and lines beginning with # are ignored.

Simple descriptions of these directives and elements are presented in the following sections, together with some examples. A more complete description of the format of the scenario file and some more comprehensive examples are presented in the chapter entitled ‘‘The scenario file’’ in the TETware Programmers Guide.

5.3.2.2 Scenario directives

Directives have **scope** within the scenario. Each directive affects the way in which `tcc` processes other directives and elements which appear within the scope of that directive. Note that some directives may not appear within the scope of other directives.

These directives may appear in scenario files and are processed by `tcc` as follows:

`:include:/`*file-name*

The test cases listed (one per line) in the named file are interpolated in the scenario at the point where this directive appears.

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

The test cases listed (one per line) in the named file are processed in parallel. If the optional *count* argument is specified, *count* copies of each test case listed in the file are processed in parallel.

`:parallel[,`*count*`]:`
`/`*test-case-name*
. . .
`:endparallel:`

The named test-cases are processed in parallel.

The precise behaviour of the `:parallel:` directive may be affected by the value of the `TET_COMPAT` configuration variable. Details of the way in which this directive behaves are presented in the chapter entitled ‘‘The scenario file’’ in the TETware Programmers Guide.

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

In build and clean modes, the test cases listed (one per line) in the named file are processed once in sequence. In execute mode, the test cases listed in the named file are executed sequentially *count* times.

`:repeat,`*count*`:`
`/`*test-case-name*
. . .
`:endrepeat:`

In build and clean modes, the named *test-cases* are processed once in sequence. In execute mode, the named *test-cases* are executed sequentially *count* times.

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

In build and clean modes, the test cases listed (one per line) in the named file are processed once in sequence. In execute mode, the test cases listed in the named file are executed sequentially until the specified number of *seconds* has expired.

`:timed_loop,`*seconds*`:`
`/`*test-case-name*
. . .
`:endtimed_loop:`

In build and clean modes, the named test cases are processed once in sequence. In execute mode, the named test cases are executed sequentially until the specified number of *seconds* has expired.

`:random:/`*file-name*

The behaviour of this directive depends on the selected modes of operation and on whether or not it appears within the scope of a looping directive.

When execute mode is selected:

— If this directive appears within the scope of at least one looping directive:

- If build mode is selected, each of the test cases listed (one per line) in the named file are built once in sequence.
- For each iteration of each of the enclosing looping directives, one of the test cases listed in the named file is selected at random and executed.
- If clean mode is selected, each of the test cases listed in the named file are cleaned once in sequence.

— Otherwise:

- One of the test cases listed in the named file is selected at random and processed according to the selected modes of operation.

When execute mode is not selected:

— Each of the test cases listed in the named file are processed once in sequence according to the selected modes of operation.

`:random:`
`/`*test-case-name*
...
`:endrandom:`

The named test cases are processed in the same way as that described above for the first format of the `:random:` directive.

These directives are only supported by the Distributed version of `tcc` and are processed as follows:

`:remote,`*nnn*$_1$`[,`*nnn*$_2$...`]:/`*file-name*

Each test case listed (one per line) in the named file is processed in sequence. As each test case is processed this processing takes place on all of the systems *nnn*$_1$, *nnn*$_2$ etc. at once.

`:remote,`*nnn*$_1$`[,`*nnn*$_2$...`]:`
`/`*test-case-name*
...
`:endremote:`

Each named test case is processed in sequence. As each test case is processed this processing takes place on all of the systems *nnn*$_1$, *nnn*$_2$ etc. at once.

If the list of systems specified with a `:remote:` directive includes system ID zero, then the named test cases are processed as **distributed** test cases; otherwise, test cases are processed as (non-distributed) **remote** test cases.

`:distributed,`*nnn*$_1$`[,`*nnn*$_2$`,`...`]:/`*file-name*

Each test case listed (one per line) in the named file is processed in sequence. As each test case is processed this processing takes place on all of the systems *nnn*$_1$, *nnn*$_2$ etc. at once.

`:distributed,`*nnn*$_1$`[,`*nnn*$_2$`,`...`]:`
`/`*test-case-name*
...
`:enddistributed:`

Each named test case is processed in sequence. As each test case is processed this processing takes place on all of the systems *nnn*$_1$, *nnn*$_2$ etc. at once.

Test cases within the scope of a `:distributed:` directive are always processed as **distributed** test cases irrespective of whether system ID zero is specified in the list of systems.

Directives may be combined by the `;` operator. When directives are combined in this way, the scope of each directive is the same as if each directive had been specified separately but the syntax used is the same as that which may be used for a single directive, as follows:

`:`*directive*$_1$`[;`*directive*$_2$...`]:/`*file-name*

The test cases listed (one per line) in the named file are processed within the scope of *directive*$_1$, *directive*$_2$ etc.

`:`*directive*$_1$`[;`*directive*$_2$`...]:`
`/`*test-case-name*
...
`:[...end`*directive*$_2$`;]end`*directive*$_1$`:`

The named test cases are processed within the scope of *directive*$_1$, *directive*$_2$ etc. Note that when the `;` operator is used in this type of construct, `end`*directive*$_1$, `end`*directive*$_2$ etc. must appear in reverse order so that each one matches its corresponding *directive*$_1$, *directive*$_2$ etc.

Where a file name name is associated with a scenario directive, this name is interpreted relative to the test suite root directory.

5.3.2.3 Scenario elements

The following elements may appear in scenario files and are processed by `tcc` as follows:

`"`*text*`"`
text is printed to the journal file as a scenario information line.

`^`*scenario-name*
The directives and elements which comprise the named scenario are interpolated at the point where this element appears. A `^`*scenario-name* may appear in a scenario in any place that a `/`*test-case-name* or a `/`*file-name* may appear.

`/`*test-case-name*
The named test case is processed according to each of `tcc`'s selected modes of operation.

`:`*directive*`:@/`*test-case-name*
The `@/`*test-case-name* element may be used in place of a `/`*file-name* which is **attached** to a directive[5]. The single named test case is processed within the scope of the associated *directive* without the need for a matching `:end`*directive*`:` to be specified.

5.3.2.4 Test case names

Test case names are interpreted by `tcc` on each system as follows:

- In build or clean mode, test case names are interpreted relative to the test suite root directory.
- In execute mode, test case names are interpreted relative to the alternate execution directory if one has been specified, otherwise they are interpreted relative to the test suite root directory.

A test case name may be followed by a list of invocable component numbers enclosed in braces. If this is done, only the invocable components specified in the list are executed in execute mode. A list of invocable components consists of one or more comma-separated numbers or ranges of numbers. A range of numbers consists of two numbers separated by a `-` character.

5. That is: not separated from the directive by white space.

For example, the following line specifies invocable components 1, 3, and 5 through 8 for the named test case:

```
                /tset/tc1{1,3,5-8}
```

5.3.2.5 Simple scenario examples

This section presents some simple test scenario examples. The first two examples may be processed either by TETware-Lite or by Distributed TETware.

In the following example, the three test cases are processed sequentially on the local system:

```
all
        /tset/tc1
        /tset/tc2
        /tset/tc3
```

In the following example, the three test cases are processed concurrently on the local system:

```
all
        :parallel:
        /tset/tc1
        /tset/tc2
        /tset/tc3
        :endparallel:
```

Since the next two examples specify the processing of remote and distributed test cases, they can only be processed by Distributed TETware and not by TETware-Lite.

In the following example, instances of each test case are processed concurrently on two remote systems:

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

In the following example, four distributed test cases are processed sequentially. Parts of each test case are executed concurrently on the local system and on two remote systems.

```
all
        :remote,000,001,002:
        /tset/dtc1
        /tset/dtc2
        /tset/dtc3
        /tset/dtc4
        :endremote:
```

5.3.3 Configuration files

5.3.3.1 Introduction

The configuration files hold configuration variable assignments for each of `tcc`'s modes of operation. In addition, configuration files may hold configuration variable assignments for use by test cases and by build and clean tools.

5.3.3.2 Configuration file format

Each line in a configuration file specifies a configuration variable assignment in the following format:

variable=*value*

Lines beginning with # and blank lines are ignored.

5.3.3.3 Configuration file names

By default, configuration files for each test suite are located in the test suite root directory on each system. However, if an alternate execution directory is specified on the local system, 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`.

The names of these files on the local system may be overridden by `tcc` command-line options if desired; for details, refer to the `tcc` manual page at the end of this guide.

5.3.3.4 Setting configuration variables on local and remote systems when using Distributed TETware

This subsection describes the interaction between variables that are specified in configuration files on local and remote systems when Distributed TETware is used. The capabilities described here are not available when TETware-Lite is used.

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).

5.3.3.5 Configuration variables used by TETware

This section describes some of the configuration variables that affect `tcc`'s operation. The list presented here only includes those variables that it might be sensible for a TETware user to change. Other TETware configuration variables and testsuite-specific variables may be specified by the test suite author. Refer to the TETware Programmers Guide for a complete list of configuration variables and their meanings.

Except where noted, variables affect the operation of both the Distributed TETware and TETware-Lite versions of `tcc`.

When Distributed TETware is used, each of the operation mode-specific sets of configuration variables that are specified without the `TET_REM`*nnn*`_` prefix on the local system is known as the **master configuration** for the related mode. Variables in the master configuration for each mode are merged with system-specific configuration variables[6] to form the **per-system configuration** for that mode. The master configuration variables affect the way that `tcc` processes test cases on all participating systems, whereas the per-system configuration variables affect the way that `tcc` processes test cases on each system individually.

Some of the variables described below are read from the master configuration and so affect `tcc`'s operation on each system in the same way, whereas others are read from the per-system configurations and so can affect `tcc`'s operation differently on different systems. The scope of

6. That is: variables specified on remote systems, or specified on the local system with a `TET_REM`*nnn*`_` prefix.

each variable described here is indicated at the end of the variable's description. Variables noted as being read from the master configuration will be ignored if constrained to a particular system by use of a `TET_REM`*nnn*`_` prefix or by being specified in a configuration file on a remote system.

`TET_EXEC_IN_PLACE`	Possible values: `True` or `False`; default: `False`. 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. In Distributed TETware the value of this variable is read from the master configuration.
`TET_OUTPUT_CAPTURE`	Possible values: `True` or `False`; default: `False`. 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. In Distributed TETware the value of this variable is read from the master configuration.
`TET_API_COMPLIANT`	Possible values: `True` or `False`; default: inverse of the value of `TET_OUTPUT_CAPTURE`. Specifies whether or not test cases and tools use a TETware API. If true, test cases and tools are expected 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. `tcc` 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). In Distributed TETware the value of this variable is read from the master configuration.
`TET_PASS_TC_NAME`	Possible values: `True` or `False`; default: the value of `TET_OUTPUT_CAPTURE`. 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. In Distributed TETware the value of this variable is read from the per-system configuration.
`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.

	In Distributed TETware the value of this variable is read from the per-system configuration.
`TET_TRANSFER_SAVE_FILES`	Possible values: `True` or `False`; default: `False`. If true, files processed 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 value of this variable has no effect in TETware-Lite. In Distributed TETware the value of this variable is read from the per-system configuration.

5.3.3.6 Example configuration file

Here is an example of a simple build mode configuration file for a test suite that uses `make` as its build tool:

```
    # example build mode configuration file

    TET_BUILD_TOOL=make
    TET_BUILD_FILE=-f makefile
    TET_OUTPUT_CAPTURE=True
```

5.3.3.7 Distributed configuration file

When Distributed TETware is used, the file `tetdist.cfg` in the test suite root directory on the local system holds configuration variables used for remote and distributed testing. This file is not used by TETware-Lite.

Distributed configuration variables are used to specify parameters for remote systems that are equivalent to those parameters on the local system that `tcc` obtains from environment variables or deduces from its current working directory. In addition, the distributed configuration file may include variable assignments relating to the TETware network interface.

These variables are not accessible by test cases or tools using API configuration variable lookup functions.

5.3.3.8 Configuration variables used for distributed testing

The following configuration variables are used by the Distributed version of `tcc` when processing test cases on remote systems. In each case, *nnn* is the number of the system to which the variable relates.

These variables are specified in the distributed configuration file `tetdist.cfg` on the local system. The distributed configuration file and the variables described here are not used by TETware-Lite.

`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_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.

`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_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 (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_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_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 (refer to the section entitled ‘‘Environment variables’’ earlier in this chapter).

In addition, the following distributed configuration variables may be specified when the XTI network interface is used:

`TET_XTI_TPI` The name of the XTI transport provider identifier on the local system. If this variable is not specified, its value defaults to `/dev/tcp`.

`TET_XTI_MODE`	Possible values: `TCP` (to indicate TCP/IP) or `OSICO` (to indicate OSI connection-oriented transport). The value of this variable indicates the underlying transport provider to be used. This variable must always be specified 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.

5.3.3.9 Example distributed configuration file

Here is an example distributed configuration file for a distributed test suite whose parts run on the local system and two remote systems:

```
# example distributed configuration file

TET_REM001_TET_ROOT=/user1/tet
TET_REM001_TET_TSROOT=/user1/tet/testsuite
TET_REM002_TET_ROOT=/user6/project/tet3
TET_REM002_TET_TSROOT=/user6/project/tet3/testsuite
```

5.3.4 The results code file

When a TETware program needs to interpret a test purpose result, it does so by referring to a table of results codes. Initially, this table contains a list of code values which have standard meanings which may (optionally) be augmented by lists contained in files provided by the test suite author. Files provided in this way may reside in both the **tet root** directory and the **test suite root** directory. Typically the file at the **tet root** level might contain codes for use by all test suites whereas the file at the **test suite root** level might contain codes for use with an individual test suite.

The default name for the file at each level is `tet_code` but this name may be changed by means of the `TET_RESCODES_FILE` configuration variable[7]. In Distributed TETware any results code files should only be provided on the local system.

Each line in a results code file consists of (blank-separated) fields as follows:

Result code value
Result name
Action indicator

7. Note that although this variable is specified in configuration files which are per operation mode, the variable itself is interpreted per `tcc` invocation and not per operation mode. Therefore it is an error for this variable to be specified with different values in different mode-specific configuration files which are read by a particular `tcc` invocation.

The result code value is a positive number between 0 and 127. Values between 0 and 31 are reserved for use by TETware. The result name is a text string enclosed between double quotes and indicates the name of the result. The text string may contain embedded blanks. The action indicator informs the TCM what to do when a test purpose returns the corresponding result and should be one of `Continue` to cause the TCM to continue processing the next test purpose, or `Abort` to cause the TCM to abandon processing the test case.

Blank lines and lines beginning with # are ignored.

Here is an example of a result code file:

```
# example result code file

# first, the standard codes required by TETware
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, the codes specific to this test suite
32      "INSPECT"       Continue
33      "STOP RUN"      Abort
```

When `tcc` starts up, its internal table contains entries for only the standard result codes. Then, if a file is provided at the **tet root** level, entries in that file are added to the table. Finally, if a file is provided at the **test suite root** level, entries in that file are added to the table. Thus, entries in the file at the **test suite root** level have precedence over entries in the file at the **tet root** level and entries in both files have precedence over the default entries which initially populate the internal table. Therefore, it is an error for either of the optional results code files to contain entries for one of the standard result codes with an incorrect result name.

5.3.5 The journal file

At the start of each test run, `tcc` creates a directory in the *test-suite-root*`/results` directory on the local system, whose name consists of an ascending sequence number followed by one or more of `b`, `e` and `c`, indicating which operation mode(s) was being used by `tcc`. By default, the journal for the test run is placed in a file called `journal` in this directory.

For example, if the 5th `tcc` run was in build and execute mode, the name of the default journal file would be

tet-root`/results/0005be/journal`

However, a different journal file name may be specified by means of a `tcc` command-line option; for details, refer to the `tcc` manual page at the end of this guide.

Each line in the journal file consists of three fields separated by `|` characters. The first field contains a number which indicates the type of the journal line. The second field contains information depending on the type of the journal line. The third field generally contains some kind of message text. Descriptions of journal lines are presented in the appendix entitled ‘‘TETware journal lines’’ at the end of this guide.

5.3.6 The `systems` and `systems.equiv` files

These files are only used by Distributed TETware processes. The are not used by TETware-Lite.

The `systems` file resides in the **tet root** directory on each system and is used by Distributed TETware processes to map TETware system IDs to some information (such as a host name) that can be used by the network interface to identify a machine. The format of the `systems` file varies according to which network interface is used by TETware.

The `systems.equiv` file resides in the `tccd` user's home directory on each system (usually that of the user `tet` on a UNIX system). This file may be used by `tccd` to determine whether or not to accept a request from a remote system. Note that not all TETware network interfaces make use of this file.

Example `systems` and `systems.equiv` files are included in the TETware distribution. Refer to the manual pages at the end of this guide for details of the formats of these files.

5.4 Network security considerations for Distributed TETware

This section describes issues for consideration by users and system administrators when using Distributed TETware.

Since Distributed TETware is intended for use in a testing environment which is not accessible from an external network, network security issues are not really addressed by Distributed TETware.

`tccd` offers services to any network entity that connects to its well-known port and can satisfy its logon criteria. Some of the services offered by `tccd` may present a potential security hazard; in particular, the ability to execute other processes. When `tccd` is built to use the socket network interface, users can exercise some control over login requests from another machine by means of entries in the `systems.equiv` file. However, when `tccd` is built to use the XTI network interface, users cannot control whether or not `tccd` accepts logon requests from other machines.

`tetsyncd` and `tetxresd` both offer services to other processes without any authentication, apart from the server logon procedure which does not perform host name checking. However, they do not use well-known ports and so are protected to some extent by ‘‘security through obscurity’’.

As a result of all this:

> **Users are strongly advised against running Distributed TETware on machines that can be accessed from an external network, unless they are satisfied that adequate measures are in place to prevent unauthorised access to networks that serve those machines.**

6. TETware programs

6.1 Introduction

This chapter describes the function and use of TETware programs. Sections which describe the Test Case Controller and Test Case Manager are applicable to both TETware-Lite and Distributed TETware, whereas sections which describe server (or daemon) processes are applicable only to Distributed TETware.

6.2 The Test Case Controller `tcc`

6.2.1 Description

The Test Case Controller (`tcc`) undertakes test suite scenario processing. `tcc` processes test cases in accordance with one or more modes of operation, as described in the section entitled ‘‘Modes of operation’’ earlier in this chapter. The default behaviour of `tcc` can be modified by many command-line options, details of which are presented in a manual page at the end of this guide.

Although the user interface to `tcc` is the same in both Distributed TETware and TETware-Lite, the operation of `tcc` is quite different in the two TETware versions. In TETware-Lite there is only one system (the local system) and `tcc` itself performs all the actions required to process test cases. However, in Distributed TETware there may be more than one system (local and/or remote) and `tcc` does not perform the actions required to process test cases itself. Instead, it sends requests to server processes which perform each required action on the appropriate system. Note that the distributed version of `tcc` always runs on the local system; indeed, in TETware terminology the local system is defined as the system on which `tcc` runs.

6.2.2 Required environment variables

Before you invoke `tcc`, you must ensure that the value of your `TET_ROOT` environment variable points to the **tet root** directory on the local system.

6.2.3 Normal scenario processing

6.2.3.1 Description

When `tcc` is invoked, it processes each test case in the specified scenario. (If no scenario is specified, then `tcc` processes each test case in the scenario named `all`). Test case processing is performed for each operational mode currently in force. For example, if `tcc` is invoked in **build** mode, each test case in the scenario is built or, if `tcc` is invoked in **build** and **execute** mode, then each test case in the scenario is first built and then executed.

6.2.3.2 Examples

Here are some simple examples of how to use `tcc` to perform normal scenario processing:

1. To build all test cases in a test suite called `mytestsuite`:

   ```
   tcc -b mytestsuite
   ```

   or

```
cd tet-root/mytestsuite
tcc -b
```

The journal is placed in *tet-root*`/mytestsuite/results/`*nnnn*`b/journal`.

2. To build, execute and clean all test cases in a test suite called `mytestsuite`:

   ```
   tcc -bec mytestsuite
   ```

   or

   ```
   cd tet-root/mytestsuite
   tcc -bec
   ```

   The journal is placed in *tet-root*`/mytestsuite/results/`*nnnn*`bec/journal`.

3. To execute test cases in a test suite called `mytestsuite` that are listed in a scenario called `myscenario` in the default scenario file:

   ```
   tcc -e mytestsuite myscenario
   ```

   The journal is placed in *tet-root*`/mytestsuite/results/`*nnnn*`e/journal`.

4. To build and execute test cases in a test suite called `mytestsuite` that are listed in the default scenario in a file called `myscenfile` in the current directory, place the journal in a file called `myjournal` in the current directory and print a trace of test case building and execution:

   ```
   tcc -be -p -s myscenfile -j myjournal mytestsuite
   ```

6.2.4 Rerun and Resume processing

6.2.4.1 Description

In addition to the normal scenario processing described above, `tcc` can re-run or resume processing of a previous test run. When you invoke `tcc` in **rerun** or **resume** mode, you specify the name of the scenario and the journal file from the previous run, and a list of test purpose result code names or `tcc` operation modes. Needless to say, the scenario that you specify must be the same as the one that was used to generate the old journal file. (If the scenario name is not specified explicitly, `tcc` re-processes the scenario named `all`).

When you run `tcc` in **rerun** mode, `tcc` re-processes test cases in the scenario whose results in the old journal file match one of those specified in the result code list. When you run `tcc` in **resume** mode, `tcc` re-processes all test cases in the scenario starting from the first test case whose result in the old journal file matches one of those specified in the result code list.

The result code list may consist of one of the following:

i. A (comma separated) list of result code names.

ii. A (comma separated) list containing one or more of the letters `b`, `e` and `c`, representing build mode, execute mode and clean mode, respectively, in the previous run.

If the list contains result code names, then test purposes which report one of these results are matched. If the list contains `tcc` operating modes, then test purposes which did not report PASS in the named operating mode(s) are matched.

When performing **rerun** and **resume** processing in build and clean modes, `tcc` performs processing at the test case level in build and clean mode, and at the invocable component level in execute mode.

When you run `tcc` in **rerun** mode, the old journal file may either be the result of a normal test run or of a previous rerun. It is not necessary for the selected modes of operation to be the same as those specified for the previous run. However, you should avoid specifying impossible operation mode combinations. For example, if you perform a test run which includes clean mode and then attempt to rerun without specifying build mode or rebuilding the test cases to be rerun in some other way, attempts to re-execute selected test cases are bound to fail.

When you run `tcc` in **resume** mode, the old journal file must be the result of a normal test run. You cannot resume using an old journal file which is the result of a previous **resume** run. In addition it is necessary for the selected modes of operation to be the same as those specified for the test run that generated the old journal file.

6.2.4.2 Examples

Here are some simple examples of how to use `tcc` to perform **rerun** and **resume** processing:

1. To re-build all test cases in the test suite named `mytestsuite` that previously failed to build in the run whose journal file was named `oldjournal`:

   ```
   tcc -b -r b oldjournal mytestsuite
   ```

2. To resume building of the first test case and execution from the first invocable component in the test suite named `mytestsuite` which reported FAIL or UNRESOLVED in the journal file named `oldjournal`:

   ```
   tcc -be -m FAIL,UNRESOLVED oldjournal mytestsuite
   ```

6.2.5 Test case locking

When `tcc` processes a test case, it prevents possible interference from other instances of `tcc` by acquiring one or more exclusive or shared locks. These locks are acquired in the test case source directory and (optionally) in the test case execution directory.

An exclusive lock is acquired by creating a file called `tet_lock` in the directory that is to be locked. A shared lock is acquired by creating a file with a unique name below a directory called `tet_lock` in the directory that is to be locked, creating the `tet_lock` directory if necessary as well. A diagnostic is printed to the journal file if `tcc` is unable to acquire the locks that it needs.

Although each instance of `tcc` is always careful to remove the locks that it created when they are no longer required, it is possible that old locks might remain after some system or process malfunction. In the event of this happening it is necessary to remove the old locks by hand.

6.2.6 Saved files processing

As indicated earlier in the section entitled ‘‘Configuration variables used by TETware’’, `tcc` can be instructed to save certain files and/or directories that may be created during test case execution. This action is controlled by the value of the configuration variable `TET_SAVE_FILES` and (when Distributed TETware is used) by the value of `TET_TRANSFER_SAVE_FILES`.

If `TET_TRANSFER_SAVE_FILES` is false when Distributed TETware is used, files are saved on the system on which they were created (be it local or remote). However, if `TET_TRANSFER_SAVE_FILES` is true, files that are to be saved are transferred from remote systems to the local system and saved there. The setting of `TET_TRANSFER_SAVE_FILES` has no effect when TETware-Lite is used.

When files are to be saved without transfer on a particular system, `tcc` creates a **saved files directory** in the *test-suite-root*`/results` directory on that system, whose name consists of an ascending sequence number followed by one or more of `b`, `e` and `c`, indicating which operation mode(s) was being used by `tcc`, thus:[8]

> *test-suite-root*`/results/`*nnnn*`[bec]`

However, when files are to be transferred and saved on the local system by Distributed TETware, `tcc` instead creates additional directories named `REMOTE`*nnn* below this directory on the local system, where *nnn* indicates the system ID (either local or remote) from which the files are to be transferred, thus:

> *test-suite-root*`/results/`*nnnn*`[bec]/REMOTE`*nnn*

Each directory thus created becomes the top of the saved files directory hierarchy for its respective system.

The variable `TET_SAVE_FILES` may be set to 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 hierarchy. Directories are created as required below the saved files directory so that, after each file has been saved, the saved file's path name below the top of the saved files directory hierarchy is the same as the portion of the file's path name below the test case's execution directory.

If a directory is found that matches one of the names, then its contents are transferred recursively.

When Distributed TETware is used, the processing described above is undertaken on each system for which a `TET_SAVE_FILES` variable has been specified.

6.2.7 Test session interruption

You can send certain keyboard-generated signals to `tcc` in order to cause it to interrupt the processing of an individual test case or of an entire test run.

When `tcc` receives a `SIGINT` signal, it aborts processing of the current test case. On a UNIX system this signal can usually be generated by typing a control-C or DEL character on the terminal where `tcc` is running. On a Win32 system this signal is always generated by typing control-C.

When `tcc` receives a `SIGQUIT` signal (on a UNIX system) or a `SIGBREAK` signal (on a Win32 system), it aborts processing of the entire test run. On a UNIX system this signal can usually be generated by typing a control-\ character on the terminal where `tcc` is running. On a Win32 system this signal is always generated by typing control-BREAK.

Users should be aware that the act of interrupting a test case can take some time to complete, particularly when the test case itself does not immediately respond to the termination signal sent by `tcc` or if there is a substantial amount of journal file or saved files processing to perform when the test case terminates.

8. This is the same directory name as that used for the directory on the local system in which the journal file is placed.

In addition, users should be aware that the use of test session interruption facilities on a Win32 system can have an adverse effect on the subsequent operation of the system. Refer to the appendix entitled ‘‘Implementation notes for TETware on Win32 systems’’ towards the end of this guide for further details.

6.3 The Test Case Controller daemon `tccd`

6.3.1 Description

The Test Case Controller daemon (`tccd`) is a server that performs functions on behalf of the Test Case Controller when Distributed TETware is used. As with all subsections describing Distributed TETware servers, this subsection is not applicable to TETware-Lite.

`tccd` must be started on each system (both local and remote) on which test cases are to be processed, before `tcc` is invoked on the local system.

When you install Distributed TETware on a UNIX system, it is recommended that you should ask your system administrator to arrange for `tccd` to be started automatically as part of each machine’s startup procedure. When you install Distributed TETware on a Windows NT system, you must arrange to start `tccd` yourself by running the `tccd` bootstrap program `tccdstart`. Details of how to start `tccd` are presented in the section entitled ‘‘Starting `tccd`’’ in the TETware Installation Guide for each type of operating system.

A manual page describing the various `tccd` command-line options is presented at the end of this guide.

6.3.2 `tccd` versions and modes of operation

On a UNIX system it is possible to build several versions of `tccd` from the source code; the version that is built is determined by a decision that must be made at compile time. Two of the versions of `tccd` are known as the **rc** version and the **inittab** version. The **rc** version is intended to be started by an entry in one of the `/etc/rc` system startup scripts, and the **inittab** version is intended to be started by an entry in the file `/etc/inittab`. The only difference between these two versions is that the **rc** version puts itself in the background when invoked, and the **inittab** version does not.

A third version of the Test Case Controller daemon may be built on systems that use the socket network interface, which is suitable for use in conjunction with the `inetd` super-server which runs on many UNIX systems. This version is known as the **inetd** version and is named `in.tccd` in accordance with established conventions.

The **rc** and **inittab** versions of `tccd` listen for requests from client processes, much like other network service daemons. When a request is received, `tccd` forks off a child copy of itself to service the client’s requests and continues to listen for requests as before. The child `tccd` processes requests from the client and terminates automatically when it is no longer required.

The **inetd** version, `in.tccd`, operates slightly differently. `inetd` listens for client requests on behalf of all the servers that it knows about. When `inetd` receives a request on the well-known **tcc** port, `inetd` forks off a copy of itself which immediately executes `in.tccd`. `in.tccd` then processes requests from the client and terminates automatically when it is no longer required. A consequence of this is that there is no `in.tccd` permanently running on a UNIX system; an instance of `in.tccd` is only invoked when actually requested from another system.

It is only possible to build a single version of `tccd` on a Windows NT system. Of the three versions of `tccd` that may be run on a UNIX system, the version of `tccd` that runs on a Windows NT system most closely resembles the **inetd** version of `tccd` described previously. On a Windows NT system, `tccd` is started on demand by a bootstrap program called `tccdstart` which performs a function similar to that performed by `inetd` on a UNIX system.

A consequence of the way in which `tccd` operates is that a machine running `tccd` can service requests from more than one client once, since each client obtains its own copy of `tccd` to use.

Since the different versions of the Test Case Controller daemon only differ in their operation on startup, references to `tccd` in the rest of this section and throughout this guide may be taken to apply equally to all versions of `tccd` unless explicitly stated to the contrary.

6.3.3 `tccd` user and group ID

By default, when `tccd` is executed on a UNIX system, it attempts to change its user and group IDs to those specified for the user `tet` in the system password database. Failure to change the user or group ID is treated as fatal if `tccd` is invoked initially with administrative privilege (i.e., with a user or group ID of less than 100). `tccd` then changes directory to the home directory of the user `tet` and sets its `HOME` environment variable to refer to that directory.

A different user name may be specified if `tccd` is invoked with the `−u` command-line option. If `tccd` is invoked in this way, it performs all of these operations using the specified user name instead of the user `tet`.

When `tccd` is executed on a Windows NT system, it does not attempt to change its user ID but instead runs with the ID of the user who starts it. In addition, if the `HOME` environment variable is not defined, `tccd` sets it to the name of the directory in which `tccd` is invoked.

6.3.4 `tccd` log file

When `tccd` is executed, it connects its standard output and standard error streams to the file `/tmp/tccdlog` on a UNIX system or to `c:/tmp/tccdlog` on a Windows NT system. In addition, `tccd` connects its standard input to `/dev/null` on a UNIX system or to `nul` on a Windows NT system. If a different log file is specified by the `-l` command-line option, `tccd` reconnects its standard output and error streams to that file as soon as possible. The result of this is that test cases and other processes which may be executed by `tccd` start execution with their standard input, standard output and standard error streams connected to these files as well[9].

All messages printed by `tccd` to the log file include the process ID and a time stamp, so that it is possible to determine the origin of a message when more than one `tccd` executes on the same machine. If a message cannot be printed to the log file for some reason, `tccd` attempts to send the message to `/dev/console` (on a UNIX system) or `con` (on a Windows NT system) as a last resort.

When the parent `tccd` daemon (but not the **inetd** version of `tccd`) starts, it prints a START message to the log file.

9. If a test case or a build or clean tool is executed with `TET_OUTPUT_CAPTURE` set to `True`, then `tcc` arranges for `tccd` to execute the tool with standard output and standard error streams connected to a different file. The contents of this file are copied to the journal file when when the tool finishes execution.

When `tccd` receives the initial logon request from a client, it prints a message to the log file indicating the origin of the message and which system ID has been assigned to the system.

6.3.5 Terminating `tccd`

Each child `tccd` terminates automatically after its client process disconnects from it. The parent `tccd` on a UNIX system may be terminated by sending it a `SIGTERM` signal.

6.4 The Test Case Manager

The Test Case Manager (TCM) is not a separate program but is instead part of each TETware API.

Distinct versions of the C and C++ APIs are provided with TETware-Lite and Distributed TETware, since the distributed versions provide support for remote and distributed testing, whereas the Lite versions do not. However, the same versions of the other APIs are provided with both TETware-Lite and Distributed TETware since these APIs do not provide support for distributed testing.

The C and C++ TCMs are linked with the compiled user-supplied test code and the C API library to form an executable file. When a distributed test case is executed by Distributed TETware, the TCMs on all the participating systems synchronise with each other at certain times during test execution, so as to ensure that test case parts running on the various systems keep in step with each other. Refer to the chapter entitled ‘‘Test case synchronisation’’ elsewhere in this guide for details of test case synchronisation.

The C and C++ APIs send diagnostic output to the journal file as test case manager messages. If this is not possible, each API writes diagnostic output to the TCM’s standard error stream[10].

The Shell, Korn Shell and Perl TCMs are each sourced by the script containing the user-supplied test code. These TCMs do not support distributed testing.

6.5 The Synchronisation daemon `tetsyncd`

6.5.1 Introduction

The Synchronisation daemon `tetsyncd` provides support for distributed test case execution when Distributed TETware is used. As with all subsections describing Distributed TETware servers, this subsection is not applicable to TETware-Lite.

6.5.2 Description

`tetsyncd` is a server that runs on the local system[11] and undertakes the processing of automatic and user synchronisation requests. An automatic synchronisation request is generated by the Test Case Manager, and a user synchronisation request is made when a test purpose makes calls to the `tet_remsync()` API library routine.

10. By default the TCM’s standard error stream is connected to the `tccd` log file when Distributed TETware is used.

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

`tetsyncd` cannot be started interactively but is started for you when required by `tcc`. `tetsyncd` terminates automatically approximately 60 seconds after the last client process has disconnected from it.

`tetsyncd` sends diagnostic output to its standard error stream. This stream is inherited from `tcc`; thus, diagnostics generated by `tetsyncd` appear wherever `tcc`'s standard error stream is directed.

6.6 The Execution Results daemon `tetxresd`

6.6.1 Introduction

The Execution Results daemon `tetxresd` provides support for test case execution when Distributed TETware is used. As with all subsections describing Distributed TETware servers, this subsection is not applicable to TETware-Lite.

6.6.2 Description

`tetxresd` is a server that runs on the local system and undertakes the processing of journal output and test purpose results for API-conforming test cases. In addition, `tetxresd` performs certain administrative functions on the local system on behalf of processes running on remote systems, mostly related to **transfer save files** processing.

`tetxresd` cannot be started interactively but is started for you when required by `tcc`. `tetxresd` terminates automatically approximately 60 seconds after the last client process has disconnected from it.

`tetxresd` sends diagnostic output to its standard error stream. This stream is inherited from `tcc`; thus, diagnostics generated by `tetxresd` appear wherever `tcc`'s standard error stream is directed.

7. Test case synchronisation

7.1 Introduction

This chapter describes how systems synchronise with each other when Distributed TETware is used, and explains how to interpret diagnostic messages which are generated when synchronisation requests do not complete successfully.

The facilities described in this chapter are not available when TETware-Lite is used.

7.2 Synchronisation request concepts

7.2.1 Request types

There are two types of synchronisation performed by Distributed TETware processes. Automatic synchronisation requests are generated when Test Case Managers synchronise with each other at certain pre-defined points during test case execution. User synchronisation requests are generated when different parts of a distributed test purpose call the `tet_remsync()` API library routine.

7.2.2 Request parameters

Each synchronisation request is accompanied by a **sync point number**, a **system ID list**, a **sync vote** and an optional **timeout**. In addition, a request may include an indication that the requesting process wishes to send or receive **sync message data** during the synchronisation operation.

Processes on systems which want to synchronise with each other send requests to the TETware Synchronisation daemon (`tetsyncd`). `tetsyncd` waits until all systems have submitted their requests and then notifies each participating process of the result.

The value of the **sync vote** specified in a synchronisation request can be either `yes` or `no`. `tetsyncd` notifies all participating processes of how each system voted in each request.

If a process specifies a **timeout** when making a request, then `tetsyncd` starts a per-process timeout as soon as the request is received. Each per-process timeout is reset to its initial value as each subsequent request is received from other participating systems; however, if the timeout for any process expires before all systems have submitted their requests then the synchronisation is considered to have failed.

It is possible for one system making a synchronisation request to send **sync message data** with the request. If the synchronisation is successful, then `tetsyncd` returns this data to other participating systems which have indicated willingness to receive such data when synchronisation is complete.

7.2.3 Sync events

`tetsyncd` defines a new **sync event** when the first system makes a request to synchronise to a particular **sync point** with a group of other systems. A **sync event** is considered to have completed as soon as one of the following conditions are met:

1. All of the systems that are expected to synchronise have done so.
2. One of the systems that has synchronised times out after having done so.
3. A process that has made a synchronisation request disconnects from `tetsyncd` before all the other systems that are expected to synchronise have done so.

When the event completes, all processes that have participated in the event are notified of the result. An event is considered to have succeeded if all systems that are expected to participate in the event submit requests with a `yes` vote. If a process on any of the participating systems submits a `no` vote, times out or disconnects from `tetsyncd` before the event completes, then the event is considered to have failed.

7.2.4 Sync states

`tetsyncd` maintains a set of **sync states** for each sync event. One sync state in this set is maintained for each system that is expected to participate in a sync event.

The sync state of a system is indicated by one of the following mnemonics:

`SYNC-YES`	The system has synchronised with a `yes` sync vote.
`SYNC-NO`	The system has synchronised with a `no` vote.
`NOT-SYNCED`	The system has not yet participated in this sync event.
`TIMED-OUT`	The system has synchronised but the associated timeout has expired before the sync event completed.
`DEAD`	The system has synchronised but the participating process has disconnected from `tetsyncd` before the sync event completed.

These mnemonics are used in diagnostic messages that relate to synchronisation request failures and other unexpected synchronisation conditions.

7.2.5 With what to synchronise?

As indicated above, when a TETware process makes a synchronisation request, it specifies a list of **system IDs** with which is wishes to synchronise. This means that any one TETware process running on a particular system can participate in a sync event on behalf of that system. It is not possible for a process to use TETware synchronisation facilities to synchronise with a particular process on a named system, or for processes on the same system to use these facilities to synchronise with each other.[12]

7.3 Automatic synchronisation requests

7.3.1 Description

Automatic synchronisation requests are generated by the TETware Test Case Manager, and by the API when a remote executed process is started. The list of systems that are expected to participate in automatic sync events for each distributed test case is defined before the first request is made. Each automatic synchronisation request is accompanied by a **sync ID** which identifies this list of systems. Processes which make automatic synchronisation requests do not send or receive sync message data.

12. Note that the term **system** refers to a logical system ID, not to a physical machine. Therefore, it is possible for two or more co-operating processes with different system IDs running on the same physical machine to use TETware synchronisation facilities to synchronise with each other.

The following subsections describe the circumstances under which automatic synchronisation requests are made, and the parameters that are used in each type of request.

7.3.2 Test case manager synchronisation

When a distributed test case is executed, the TCMs on each participating system synchronise with each other during certain stages of test case processing. The sync point number associated with each request is used to identify which stage is about to begin. The timeout specified with each request depends on which stage is about to begin.

The following table lists these stages, the sync point numbers that are used to identify them and the timeouts that are used:

Stage in test case processing	**Sync point number**[13]	**Timeout (seconds)**
At TCM startup time	1	60
Before the startup function (if any) is called	2	60
At the start of each invocable component	$ICno * 2^{16}$	60
At the start of each test purpose	$(ICno * 2^{16}) + (TPno * 2)$	60
At the end of each test purpose	$(ICno * 2^{16}) + (TPno * 2) + 1$	600
Before the cleanup function (if any) is called	$((ICcount + 1) * 2^{16}) + 2$	60

In this table, *ICno* is the number of the invocable component being processed, *TPno* is the number of the test purpose being processed and *ICcount* is the number of invocable components in the test case.

Normally, TCMs on each participating system specify a `yes` sync vote in each request. However, if a TCM on one system is about to execute a test purpose which has been deleted (by a previous call to `tet_delete()` in that test case), it instead specifies a `no` sync vote in the request made at test purpose start. When all the other TCMs see this `no` vote, they interpret this to mean that the test purpose is deleted and do not execute it.

In addition, if the consolidated result of a test purpose has an action code of `Abort`, the TCM on the master system[14] synchronisises to the end of the last test purpose in the test case using a `no` vote. This causes all the other TCMs to perform the following actions:

i. Any remaining test purposes in the current invocable component are deleted.

ii. No further invocable components are executed, but test case cleanup processing is performed.

13. It will be seen that the way that automatic sync point numbers are calculated imposes a limit of ($2^{15} – 1$) test purposes per test case and ($2^{15} – 2$) invocable components per test case.

14. That is: the first system in the list specified with the associated `:remote:` or `:distributed:` directive.

7.3.3 Remote executed process synchronisation

When a test case starts a remote process by calling `tet_remexec()`, the remote process synchronises with the test case that called `tet_remexec()`. This is to ensure that the test case waits until the remote process has started up before continuing execution. Sync point number 1 and a `yes` sync vote are used in this request and the timeout is set to 60 seconds.

If the remote system's `tccd` is unable to execute the process for some reason, it performs the initial synchronisation operation on behalf of that remote process but instead specifies a `no` vote in the request.

The way that synchronisation with remote executed processes is implemented makes it possible for a test case to start more than one process on the same remote system.

7.3.4 Error handling

There are two classes of error that can occur during automatic synchronisation requests, as follows:

— the request fails as a result of some problem that occurs in the API or in `tetsyncd`; these are described below as **synchronisation request failures**

— some problem is detected with one of the other systems which participated (or should have participated) in the sync event; these are described below as **synchronisation errors**

If an automatic synchronisation request failure occurs, then the TCM emits a single diagnostic indicating which automatic synchronisation request was being attempted and the cause of the failure.

If a problem is detected with one of the other systems involved in a sync event, then the TCM emits one diagnostic for each affected system. Each diagnostic indicates which automatic synchronisation request was being attempted and system ID and sync state of the affected system.

7.3.5 Example error messages

In the following examples, suppose that parts of a distributed test case are being executed on systems 0, 1 and 2.

7.3.5.1 Example 1

Suppose that a test case could not be started on system 1 for some reason. The TCM on (say) system 0 will time out waiting for system 1 to synchronise at TCM startup time, and will generate the following message:

```
system 0, reply code = ER_TIMEDOUT: initial sync error, \
        sysid = 1, state = NOT-SYNCED
```

The TCM that started successfully on system 2 will generate the following message:

```
system 2, reply code = ER_SYNCERR: initial sync error, \
        sysid = 0, state = TIMED-OUT
system 2, reply code = ER_SYNCERR: initial sync error, \
        sysid = 1, state = NOT-SYNCED
```

7.3.5.2 Example 2

If the TCMs on systems 1 and 2 synchronise to the end of (say) test purpose 4 and the TCM on system 1 times out before the TCM on system 0 reaches the same point, the TCM on system 1 will generate the following message:

```
system 1, reply code = ER_TIMEDOUT: Auto Sync error at end of TP 4, \
        sysid = 0, state = NOT-SYNCED
```

and the TCM on system 2 will generate the following message:

```
system 2, reply code = ER_SYNCERR: Auto Sync error at end of TP 4, \
        sysid = 0, state = NOT-SYNCED
system 2, reply code = ER_SYNCERR: Auto Sync error at end of TP 4, \
        sysid = 1, state = TIMED-OUT
```

At this point, the sync event is considered to have completed.

When the TCM on system 0 finally makes its synchronisation request at the end of test purpose 4, it will generate the following message:

```
system 0, reply code = ER_DONE: Auto Sync failed at end of TP 4
```

This indicates that the TCM on system 0 has missed the sync event because the event has already completed.

7.4 User synchronisation requests

7.4.1 Description

A user synchronisation request is generated when a test purpose in a distributed test case calls the `tet_remsync()` API library routine[15]. The sync point number, system ID list, vote and timeout are specified in each call. In addition to these parameters, a process can send, or indicate willingness to receive, sync message data. When this is done and all participating systems use the same sync point number, message data sent by the sending system may be returned to the receiving systems on completion of the event.

`tetsyncd` defines a separate sequence of user sync events for each distinct system ID list specified in `tet_remsync()` calls made by test purposes in a particular distributed test case. Thus, a user sync event will only be successful if the test purposes on all systems that are expected to participate in the event all specify the same system ID list in their `tet_remsync()` calls.

User sync events have lower precedence than automatic sync events. Therefore, if the test purpose on one system returns control to the TCM while test purposes on other systems are waiting on a user sync event that includes that system, the user sync event is considered to have completed unsuccessfully and participating processes are notified accordingly.

15. The `tet_remsync()` function replaces the `tet_sync()` and `tet_msync()` functions that have been implemented in previous DTET and dTET2 releases. In order to provide backward compatibility with existing test suites, `tet_sync()` and `tet_msync()` are still supported in TETware but these functions are now marked ‘‘to be withdrawn’’.

7.4.2 Error handling

Synchronisation request failures and synchronisation errors for user synchronisation requests are defined in the same way as for automatic synchronisation requests. In addition, if one or more of the participating systems specifies a `no` vote, this causes a failure indication on all systems.

By default, the API prints a test case manager message to the journal file when a user synchronisation request is unsuccessful. Each diagnostic indicates the sync point number of the request that was unsuccessful and the system IDs and sync states of the systems which failed to synchronise or timed out. The formats of diagnostics printed to the journal file are different from those generated for unsuccessful automatic synchronisation requests. Examples of the formats that may be used to report an unsuccessful user synchronisation request are presented in the next section.

However, it is possible for the test suite author to arrange for a different action to be taken when a user synchronisation request is unsuccessful. When this is done, the default action is not taken and the API does not generate journal messages of the type described in the next section.

7.4.3 Example error messages

The following subsections contain examples of the error messages that are generated by the API's default error handler routine. These messages are not generated when a user-supplied error handler routine is specified.

In the following examples, suppose that parts of a distributed test case are being executed on systems 0, 1 and 2. Suppose that sync point number 12 is being used in each case and that the timeout is set to 30 seconds.

7.4.3.1 Example 1

Suppose the test purpose on system 0 expects to synchronise with the test purpose on system 1, but the test purpose on system 1 returns control to the TCM without making a synchronisation request. The API on system 0 will generate the following messages:

```
system 0: sync operation failed, syncptno = 12, \
        other system did not sync or timed out
system 0: system =  1, state = NOT-SYNCED
```

7.4.3.2 Example 2

Suppose that all the systems expect to synchronise with each other but that system 1 times out before system 0 reaches the sync point. The API on system 1 will generate the following messages:

```
system 1: sync operation failed, syncptno = 12, \
        request timed out after waittime of 30 seconds
system 1: system =  0, state = NOT-SYNCED
system 1: system =  2, state = SYNC-YES
```

and the API on system 2 will generate the following messages:

```
system 2: sync operation failed, syncptno = 12, \
        one or more of the other systems did not sync \
        or timed out
system 2: system =  0, state = NOT-SYNCED
system 2: system =  1, state = TIMED-OUT
```

At this point the event is to considered to have completed.

When the test purpose on system 0 finally makes its synchronisation request, the request will fail because the associated event has already happened. The API on system 0 will generate the following message:

```
system 0: sync operation failed, syncptno = 12, event already happened
```

This indicates that the part of the test purpose on system 0 has missed the sync event because the event has already completed.

7.4.3.3 Example 3

Suppose that the test case on system 1 terminates unexpectedly before the sync event completes. The API on system 0 will generate the following messages:

```
system 0: sync operation failed, syncptno = 12, \
        one or more of the other systems did not sync \
        or timed out
system 0: system =  1, state = DEAD
system 0: system =  2, state = SYNC-YES
```

and the API on system 2 will generate the following messages:

```
system 2: sync operation failed, syncptno = 12, \
        one or more of the other systems did not sync \
        or timed out
system 2: system =  0, state = SYNC-YES
system 2: system =  1, state = DEAD
```

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.

B. TETware directory structure

The following diagram illustrates the directory structure used by TETware. Each directory level is represented by a level of indentation. Directory names are followed by a `/` character. Some directories and files shown here are part of the TETware distribution, whereas others are part of the user-supplied test suites or are created in the course of TETware's operation.

The suffixes shown in this diagram are the ones that are used on UNIX systems. On Win32 systems, object files (`.o` files) have the suffix `.obj`, archive library files (`.a` files) have the suffix `.lib` and executable files have the suffix `.exe`. Other suffixes shown here are the same on both systems.

The structure shown here is the complete one used when a source code distribution of Distributed TETware is built to support all of the available options. Therefore not all of the directories and files shown here are present in TETware-Lite or if a binary distribution is installed.

```
tet-root/
    bin/
        tcc
        tccd or in.tccd∂
        tccdstart∂
        tetsyncd∂
        tetxresd∂
        other TETware executables
    contrib/
        directories containing user-contributed software
    doc/
        TETware documentation
    inc/
        tet3/
            TETware API header files
        directories containing compatibility-mode links to API header files§
    lib/
        ksh/
            tetapi.ksh
            tcm.ksh
        perl/
            api.pl*
            tcm.pl*
```

∂ Not present in TETware-Lite.

§ Only on UNIX systems where TETware has been installed with an option which provides source code compatibility with for test suites written for use with previous TET versions.

* Only on systems which support Perl.

```
        tet3/
            Ctcm.o†
            Ctcmchild.o†
            Cthrtcm.o†‡
            Cthrtcmchild.o†‡
            libapi.a
            libthrapi.a‡
            tcm.o
            tcmchild.o
            tcmrem.o∂
            thrtcm.o‡
            thrtcmchild.o‡
        xpg3sh/
            tetapi.sh
            tcm.sh
        directories containing compatibility-mode links to API object and library files§
    src/
        defines/
            platform-specific makefile definition files
        ksh/
            Korn Shell API source tree
        perl/
            Perl API source tree
        tet3/
            TETware C source tree
        xpg3sh/
            Shell API source tree
    systems∂
    systems.equiv∂
    test-suite-root/
        tet_code
        tet_scen
        tetbuild.cfg
        tetclean.cfg
        tetdist.cfg
        tetexec.cfg
        results/
            nnnn[bec]/
                REMOTEnnn/∂
```

† Only on systems which support C++.

‡ Only on systems which support threads.

```
                    transfer save files
                journal
                other results files
        tet_tmp_dir/
            nnnnnx/
        test case files and directories
```

C. TETware journal lines

This appendix describes the various journal lines that may appear in a TETware journal file.

Test Case Controller Start

`0|`*version time date*`|`*who*

This message is generated by `tcc` at the beginning of each TCC execution. The parameters include the version of `tcc` used, the time at which TCC execution started and the current date. The message area contains information about who ran the test and lists the `tcc` command-line that was used.

Local System Information

`5|`*sysname nodename release version machine*`|`*text*

This message follows the Test Case Controller Start message. On a UNIX system the parameters are obtained from the information returned by the `uname()` system call. On a Win32 system the parameters identify the name of the operating system, the computer name, the operating system's major and minor version numbers and the processor type. In Distributed TETware this information relates to the local system.

Test Case Start

`10|`*activity testcase time*`|`*IClist*

A test case start message is generated by `tcc` for each test case executed in a scenario. The parameters are the sequence number of this TCC activity, the path name of the test case and the time the test was started. If the corresponding scenario line contains a list of invocable components to be executed, then that IC list is included in the text area of the message.

Test Case Manager Start

`15|`*activity version ICcount*`|`*text*

The TCM writes this message during its initialisation. The parameters include the sequence number of this TCC activity, the version of the TCM being used and the number of invocable components to be executed.

Local System Configuration Start

`20|`*pathname mode*`|`*text*

Configuration start messages are placed in the journal by `tcc` at test session startup for each of the selected modes of operation. Parameters are the path name of the configuration file being referenced and the mode to which this configuration information applies.

Remote System Configuration Start

`20|`*nnn mode*`|`*text*

Where *nnn* describes the remote system ID instead of the path name of the configuration file.

Configuration Variable Setting

`30||`*variable*=*value*

A configuration variable setting line is written to the journal by `tcc` for each variable set for the current mode. The message area indicates the name of the configuration variable and the value to which it was set.

Configuration End

`40||`*text*

A configuration end message is placed in the journal by `tcc` to indicate the end of configuration options for a particular mode.

Test Case Controller Message

`50||`*text*

`tcc` generates messages when it encounters execution problems. The message area gives an indication of the nature of the problem.

Scenario Information

`70||`*text*

`tcc` generates a scenario information line when it encounters a journal message in the scenario file being processed. The text of the message is included in the message area.

Test Case End

`80|`*activity status time*|*text*

A test case end message is generated by `tcc` after each test case completes execution. The parameters are the TCC activity number, the time the execution completed and an indication of the TCM's completion status.

User Abort

`90|`*time*|*text*

`tcc` generates an operator abort message when its execution is interrupted by the user.

Captured Output

`100|`*activity*|*text*

`tcc` generates a captured message for each line of output captured when the output capture mode is enabled. The parameter is the TCC activity counter.

Build Start

`110|`*activity testcase time*|*text*

A build start message is generated by `tcc` prior to build tool execution. The parameters are the TCC activity counter, the path name of the test case being built and the time the build started.

Build End

`130`|*activity status time*|*text*

A build end message is generated by `tcc` upon completion of build tool execution. It contains the TCC activity number, an indication of the completion status of the build tool and the time of that completion.

Test Purpose Start

`200`|*activity TPnumber time*|*text*

The TCM generates a test purpose start message for each test purpose executed. The message contains the TCC activity number, the test purpose number and the time execution started.

Test Purpose Result

`220`|*activity TPnumber result time*|*result-name*

The TCM generates a result message for each test purpose executed. Parameters for this message are the TCC activity number, the test purpose number, the result code and the time execution completed.

Clean Start

`300`|*activity testcase time*|*text*

`tcc` writes a clean start message to the journal before invoking the clean tool for a given test case.

Clean End

`320`|*activity status time*|*text*

After a clean operation on a test case is complete, `tcc` records the TCC activity number, the completion status and the time at which the operation was completed.

Invocable Component Start

`400`|*activity ICnumber TPcount time*|*text*

The TCM generates an invocable component start message for each invocable component that is executed during a test case run. This message contains the TCC activity counter, the invocable component number, number of test purposes to be executed and the time invocable component execution started.

Invocable Component End

`410`|*activity ICnumber TPcount time*|*text*

The TCM generates an invocable component end message for each invocable component that is completed. The message contains the TCC activity counter, the invocable component number, the number of test purposes actually executed and the time of the invocable component completion.

Test Case Manager Message

`510`|*activity*|*text*

If the TCM or API encounters a problem, a TCM message will be written to the journal. The parameter indicates the TCC activity counter and the message area contains a brief description of the problem.

Test Case Information

`520|`*activity TPnumber context block sequence*`|`*text*

When a test case outputs information to the execution results file it is recorded as test case information. The message specified by the test case is in the text area of this line.

Parallel Start

`600|`*count*`|`*text*

The scenario file `parallel` directive start marker.

Parallel End

`620||`*text*

The scenario file `parallel` directive end marker.

Implied Sequential Start

`630||`*text*

Marks the start of an Implied Sequential directive generated by `tcc` when in ETET compatibility mode.

Implied Sequential End

`640||`*text*

Marks the end of an Implied Sequential directive generated by `tcc` when in ETET compatibility mode.

Repeat Start

`700|`*count*`|`*text*

The scenario file `repeat` directive start marker, where *count* is the number of times the test cases within the scope of the `repeat` directive are to be repeated.

Repeat End

`720||`*text*

The scenario file `repeat` directive end marker.

Timed Loop Start

`730|`*seconds*`|`*text*

The scenario file `timed_loop` directive start marker. Execution of test cases within the scope of this directive is repeated until the specified number of *seconds* have elapsed.

Timed Loop End

`740||`*text*

The scenario file `timed_loop` directive end marker.

Random Start

`750||`*text*

The scenario file `random` directive start marker.

Random End

`760||`*text*

The scenario file `random` directive end marker.

Remote Start

`800|`nnn_1`,`nnn_2`,...|`*text*

The scenario file `remote` directive start marker. nnn_1, nnn_2 etc. are the system IDs specified with this directive.

Remote End

`820||`*text*

The scenario file `remote` directive end marker.

Distributed Start

`830|`nnn_1`,`nnn_2`,...|`*text*

The scenario file `distributed` directive start marker. nnn_1, nnn_2 etc. are the system IDs specified with this directive.

Distributed End

`840||`*text*

The scenario file `distributed` directive end marker.

Test Case Controller End

`900|`*time*`|`*text*

When `tcc` finishes processing the scenario, it generates a TCC end message. This message indicates the time execution completed.

D. TETware demonstration journal file

D.1 Introduction

Instructions on how to run the Distributed TETware demonstration test suite are presented in one of the sections in the chapter entitled ‘‘Running the TETware demonstrations’’ elsewhere in this guide. This appendix contains an example journal file that might be generated when this test suite is built, executed and cleaned by `tcc` on a pair of UNIX systems.

D.2 Example journal file

```
0|3.0b 20:09:33 19961128|User: tet (105) TCC Start, \
        Command line: tcc -bec demo
20|/home/tet/tet3/demo/tetbuild.cfg 0|Config Start
30||TET_BUILD_TOOL=make
30||TET_BUILD_FILE=-f makefile
30||TET_OUTPUT_CAPTURE=True
30||TET_API_COMPLIANT=False
30||TET_PASS_TC_NAME=True
30||TET_VERSION=3.0b
40||Config End
20|/home/tet/tet3/demo/tetexec.cfg 1|Config Start
30||TET_OUTPUT_CAPTURE=False
30||TET_EXEC_IN_PLACE=False
30||TET_API_COMPLIANT=True
30||TET_PASS_TC_NAME=False
30||TET_VERSION=3.0b
40||Config End
20|/home/tet/tet3/demo/tetclean.cfg 2|Config Start
30||TET_CLEAN_TOOL=rm
30||TET_CLEAN_FILE=-f
30||TET_OUTPUT_CAPTURE=True
30||TET_API_COMPLIANT=False
30||TET_PASS_TC_NAME=True
30||TET_VERSION=3.0b
40||Config End
20|/home/tet/tet3/demo/tetdist.cfg 3|Config Start
30||TET_REM000_TET_ROOT=/home/tet/tet3
30||TET_REM000_TET_SUITE_ROOT=/home/tet/tet3
30||TET_REM000_TET_TSROOT=/home/tet/tet3/demo
30||TET_REM000_TET_TMP_DIR=/home/tet/tet3/demo/tet_tmp_dir
30||TET_REM001_TET_ROOT=/home/tet
30||TET_REM001_TET_TSROOT=/home/tet/demo
30||TET_REM001_TET_TMP_DIR=/home/tet/demo/tet_tmp_dir
40||Config End
20|remote_001 0|Config Start
30||TET_BUILD_TOOL=make
30||TET_BUILD_FILE=-f makefile
30||TET_OUTPUT_CAPTURE=True
30||TET_API_COMPLIANT=False
30||TET_PASS_TC_NAME=True
30||TET_VERSION=3.0b
40||Config End
```

```
20|remote_001 1|Config Start
30||TET_OUTPUT_CAPTURE=False
30||TET_EXEC_IN_PLACE=False
30||TET_API_COMPLIANT=True
30||TET_PASS_TC_NAME=False
30||TET_VERSION=3.0b
40||Config End
20|remote_001 2|Config Start
30||TET_CLEAN_TOOL=rm
30||TET_CLEAN_FILE=-f
30||TET_OUTPUT_CAPTURE=True
30||TET_API_COMPLIANT=False
30||TET_PASS_TC_NAME=True
30||TET_VERSION=3.0b
40||Config End
70||"starting scenario"
800|000,001|Remote Start, scenario ref 2-0
110|0 /ts/tc1 20:09:34|Build Start, scenario ref 3-0
100|0|  cc -I../../inc/tet3 -Xa -o tc1 tc1.c ../../lib/tet3/tcm.o \
        ../../lib/tet3/libapi.a \
100|0|          -lsocket -lnsl
130|0 0 20:09:55|Build End, scenario ref 3-0
110|1 /ts/tc1 20:09:34|Build Start, scenario ref 3-1
100|1|  cc -I../../inc/tet3 -Xa -o tc1 tc1.c ../../lib/tet3/tcm.o \
        ../../lib/tet3/libapi.a \
100|1|          -lsocket -lnsl
130|1 0 20:09:55|Build End, scenario ref 3-1
10|2 /ts/tc1 20:09:55|TC Start, scenario ref 3-0
15|2 3.0b 1|TCM Start
400|2 1 1 20:09:58|IC Start
200|2 1 20:09:58|TP Start
520|2 1 00104958 1 1|This is the first test case (tc1)
520|2 1 00004957 1 1|This is the first test case (tc1)
220|2 1 0 20:09:59|PASS
410|2 1 1 20:09:59|IC End
80|2 0 20:09:59|TC End, scenario ref 3-0
300|3 /ts/tc1 20:10:00|Clean Start, scenario ref 3-0
320|3 0 20:10:02|Clean End, scenario ref 3-0
300|4 /ts/tc1 20:10:00|Clean Start, scenario ref 3-1
320|4 0 20:10:02|Clean End, scenario ref 3-1
110|5 /ts/tc2 20:10:02|Build Start, scenario ref 4-0
100|5|  cc -I../../inc/tet3 -Xa -o tc2 tc2.c ../../lib/tet3/tcm.o \
        ../../lib/tet3/libapi.a \
100|5|          -lsocket -lnsl
130|5 0 20:10:17|Build End, scenario ref 4-0
110|6 /ts/tc2 20:10:02|Build Start, scenario ref 4-1
100|6|  cc -I../../inc/tet3 -Xa -o tc2 tc2.c ../../lib/tet3/tcm.o \
        ../../lib/tet3/libapi.a \
100|6|          -lsocket -lnsl
130|6 0 20:10:17|Build End, scenario ref 4-1
10|7 /ts/tc2 20:10:17|TC Start, scenario ref 4-0
15|7 3.0b 1|TCM Start
400|7 1 1 20:10:20|IC Start
200|7 1 20:10:20|TP Start
520|7 1 00105006 1 1|This is the second test case (tc2, slave)
```

```
520|7 1 00005005 1 1|This is the second test case (tc2, master).
520|7 1 00005005 1 2|
520|7 1 00005005 1 3|The master part of this test purpose reports PASS
520|7 1 00005005 1 4|but the slave part of this test purpose reports FAIL
520|7 1 00005005 1 5|so the consolidated result of the test purpose is FAIL.
520|7 1 00005005 1 6|
520|7 1 00005005 1 7|The lines in this block of text are printed by a single
520|7 1 00005005 1 8|call to tet_minfoline() in the master part of the test
520|7 1 00005005 1 9|purpose so output from the slave part of the test purpose
520|7 1 00005005 1 10|won’t be mixed up with these lines.
220|7 1 1 20:10:20|FAIL
410|7 1 1 20:10:20|IC End
80|7 0 20:10:22|TC End, scenario ref 4-0
300|8 /ts/tc2 20:10:24|Clean Start, scenario ref 4-0
320|8 0 20:10:25|Clean End, scenario ref 4-0
300|9 /ts/tc2 20:10:24|Clean Start, scenario ref 4-1
320|9 0 20:10:25|Clean End, scenario ref 4-1
70||"next is the last test case"
110|10 /ts/tc3 20:10:25|Build Start, scenario ref 6-0
100|10| cc -I../../inc/tet3 -Xa -o tc3 tc3.c ../../lib/tet3/tcm.o \
        ../../lib/tet3/libapi.a \
100|10|         -lsocket -lnsl
130|10 0 20:10:41|Build End, scenario ref 6-0
110|11 /ts/tc3 20:10:25|Build Start, scenario ref 6-1
100|11| cc -I../../inc/tet3 -Xa -o tc3 tc3.c ../../lib/tet3/tcm.o \
        ../../lib/tet3/libapi.a \
100|11|         -lsocket -lnsl
130|11 0 20:10:41|Build End, scenario ref 6-1
10|12 /ts/tc3 20:10:42|TC Start, scenario ref 6-0
15|12 3.0b 2|TCM Start
400|12 1 1 20:10:50|IC Start
200|12 1 20:10:50|TP Start
520|12 1 00005056 1 1|This is tp1 in the third test case (tc3, master)
520|12 1 00005056 1 2|sync with slave (sysid: 1)
520|12 1 00105063 1 1|This is tp1 in the third test case (tc3, slave)
520|12 1 00105063 1 2|sync with master (sysid: 0)
220|12 1 0 20:10:50|PASS
410|12 1 1 20:10:50|IC End
400|12 2 1 20:10:50|IC Start
200|12 2 20:10:50|TP Start
520|12 2 00005056 1 1|This is tp2 in the third test case (tc3, master)
520|12 2 00005056 1 2|send message "test data" to slave (sysid: 1)
520|12 2 00105063 1 1|This is tp2 in the third test case (tc3, slave)
520|12 2 00105063 1 2|sync with master (sysid: 0) and receive data
520|12 2 00105063 1 3|received message "test data" from master
220|12 2 0 20:10:50|PASS
410|12 2 1 20:10:50|IC End
80|12 0 20:10:51|TC End, scenario ref 6-0
300|13 /ts/tc3 20:10:52|Clean Start, scenario ref 6-0
320|13 0 20:10:54|Clean End, scenario ref 6-0
300|14 /ts/tc3 20:10:52|Clean Start, scenario ref 6-1
320|14 0 20:10:54|Clean End, scenario ref 6-1
820||Remote End, scenario ref 2-0
70||"done"
900|20:10:54|TCC End
```

E. Server reply codes

E.1 Introduction

When a TETware client process makes a request of a server, a server reply code is returned to the client which indicates the success or failure of the request. Many of these reply codes are interpreted by the client and so are never visible to the user. However, some reply codes may be printed in TETware diagnostic messages which appear in the journal file on the master system or in the `tccd` log file on one of the slave systems.

In addition, when an API function call is unsuccessful, the API places a value in the global variable `tet_errno` which indicates the reason for the failure. The symbolic names corresponding to the values which may appear in `tet_errno` are defined in *tet-root*`/inc/tet3/tet_api.h` and are formed by prepending the string `TET_` to each of the names listed here. The meaning of each API error code thus defined is the same as that of the corresponding server reply code described here.

A client-server architecture is not used in TETware-Lite. In situations where the Distributed version of `tcc` sends an action function request to `tccd`, the Lite version of `tcc` performs the action itself. However, the control logic in both versions of `tcc` is the same. Therefore an action function invoked within the Lite version of `tcc` returns the same reply code to the control logic as when that action function is invoked remotely in `tccd` as a result of a request sent from the Distributed version of `tcc`. Thus it is possible for certain diagnostic messages generated by the Lite version of `tcc` to include a server reply code even though the associated processing does not involve interaction with a server.

E.2 List of server reply codes

The following is a list of possible server reply codes and their meanings:

`ER_OK`	No error. The request completed successfully.
`ER_2BIG`	Argument list too long. The request could not be processed because the size of an argument list presented to one of the `exec` or `spawn` system calls was greater than the maximum permitted by the system.
`ER_ABORT`	Abort test case. The action associated with the consolidated result of a distributed test purpose indicated that the test case should be aborted.
`ER_CONTEXT`	Request out of context. A request arrived before a pre-requisite message had been received by the server.
`ER_DONE`	Event already happened. A synchronisation request arrived after the related sync event had completed, or IC or TP start or end requests arrived in the wrong order, or a CONFIG request violated the configuration variable exchange protocol.

`ER_DUPS`	Duplicate system IDs in list. The list of system IDs specified in the request contained duplicate entries.
`ER_ERR`	General error code. Diagnostics which include this reply code are always preceded by another message describing the error in more detail. If the error actually occurred on a slave system, it is likely that the more detailed message will appear in the `tccd` log file on the slave system rather than in the journal file on the master system.
`ER_FID`	Bad file ID. An invalid file ID was specified in a file transfer request. This usually indicates an internal program error of some kind.
`ER_FORK`	Can’t fork. `tccd` was unable to fork while processing an `OP_EXEC` request. A message describing the reason for the failure is printed in the `tccd` log file on the affected system.
`ER_INPROGRESS`	Event in progress. An attempt was made to modify the system ID list associated with an event after the event had started, or IC or TP start or end requests arrived in the wrong order, or a CONFIG request violated the configuration variable exchange protocol.
`ER_INTERN`	Internal server error. A server request processing function neglected to supply a message reply code. This usually indicates an internal program error of some kind.
`ER_INVAL`	Invalid request parameter. The value of one of the request parameters was not valid for this request type. This error may be raised both in the client’s server function interface code and in the server’s request processing code.
`ER_LOGON`	Logon protocol error. A client issued a server request before having first logged on to the server, or attempted to log on to a server more than once. This error may be raised both in the client’s server function interface code and in the server’s request processing code. This usually indicates an internal program error of some kind.
`ER_MAGIC`	Bad magic number. An interprocess message header contained an invalid magic number. This can occur if the client and server data streams become out of sync or when the client and server are incompatible (e.g., because they are derived from different TETware releases).
`ER_NOENT`	No such file or directory. A file specified in an `OP_EXEC` request could not be found, or a saved files directory could not be accessed.

`ER_PERM`	No permission. A request was received from a client that was not authorised to send it, or the server did not have the permissions necessary to perform the requested action.
`ER_PID`	Invalid process ID. The process ID specified in an `OP_WAIT` or `OP_KILL` request did not refer to a process started by the requesting system.
`ER_RCVERR`	Message receive error. A server could not process a request because an I/O error occurred while the message was being received.
`ER_REQ`	Invalid request. A request was attempted that was not valid for this server type. This error may be raised both in the client's server function interface code and in the server's request processing code. This usually indicates an internal program error of some kind.
`ER_SIGNUM`	Invalid signal number. The signal number specified in an `OP_KILL` request was not valid on the target system.
`ER_SNID`	Bad sync ID. A request to `tetsyncd` specified an invalid sync identifier. This usually indicates an internal program error of some kind.
`ER_SYNCERR`	Synchronisation was unsuccessful. A sync event completed unsuccessfully because one or more of the participants did not synchronise, submitted a `no` sync vote, timed out or terminated unexpectedly.
`ER_SYSID`	Bad system ID. A client running on a system which is not participating in a distributed test purpose tried to participate in an event relating to that test purpose, or no process from a system that was supposed to be participating in an event was logged on to the server.
`ER_TIMEDOUT`	Request timed out. The timeout specified with the request expired before the request could complete.
`ER_TRACE`	Tracing not configured. An `OP_TRACE` request was sent to a server which had been compiled with tracing disabled.
`ER_WAIT`	No un-waited-for children to report. An `OP_WAIT` request with a zero timeout found that the specified process had not yet exited.
`ER_XRID`	Bad execution results file ID. A request to `tetxresd` specified an invalid execution results file identifier. This usually indicates an internal program error of some kind.

F. Mnemonics used in TETware diagnostics

F.1 Introduction

This appendix describes some of the mnemonics that may be used in diagnostic and trace messages generated by TETware programs. Some mnemonics are used both in Distributed TETware and TETware-Lite, whereas others are only used in Distributed TETware.

F.2 Process types

Each type of process is assigned a process type identifier.

The following table lists TETware process types and their meanings:

Process type	Program name	Description
`MTCC`	`tcc`	Master test case controller
`MTCM`	*test-case-name*	Master test case manager
`STAND`	*program-name*	Stand-alone programs
`STCC`	`tccd`	Test case controller daemon
`STCM`	*test-case-name*	Slave test case manager
`SYNCD`	`tetsyncd`	Synchronisation daemon
`XRESD`	`tetxresd`	Execution results daemon

F.3 Process states

The client and server interface code in Distributed TETware maintains a process state for each connection to another process. The meanings of many of the states depend on whether the connected process is a client or a server.

The following table lists TETware process states and their meanings:

Process	Connected to	Connected process state	Description
all clients	any server	`CONNECT`	Connecting to server
all clients	any server	`DEAD`	Server connection has closed
all servers	any client	`DEAD`	Client has disconnected
all clients	any server	`IDLE`	Awaiting reply from server
all servers	any client	`IDLE`	Awaiting request from client
all clients	any server	`PROCESS`	Performing normal processing
all servers	any client	`PROCESS`	Processing a request from client
all processes	any process	`RCVMSG`	Receiving a message
all processes	any process	`SNDMSG`	Sending a message
`tetsyncd`	any client	`WAITSYNC`	Waiting for a sync event to complete

F.4 Server request codes

Each request sent by a client to a server process includes a request code. Some requests are implemented in all TETware servers, whereas others are specific to a particular server.

The following table lists TETware server request codes and their meanings:

Request code	Processed by	Description
`OP_ACCESS`	`tccd`	Check accessibility of a file
`OP_ASYNC`	`tetsyncd`	Auto sync request
`OP_CFNAME`	`tccd`	Register configuration file name
`OP_CHDIR`	`tccd`	Change directory
`OP_CODESF`	`tetxresd`	Send results code file name
`OP_CONFIG`	`tccd`	Assign configuration variables
`OP_EXEC`	`tccd`	Execute a process
`OP_FCLOSE`	`tccd`, `tetxresd`	Close a text file
`OP_FOPEN`	`tccd`, `tetxresd`	Open a text file
`OP_GETS`	`tetxresd`	Read strings from a text file
`OP_ICEND`	`tetxresd`	Signal IC end
`OP_ICSTART`	`tetxresd`	Signal IC start
`OP_KILL`	`tccd`	Send signal to process
`OP_LOCKFILE`	`tccd`	Create a lock file
`OP_LOGOFF`	all servers	Log off server
`OP_LOGON`	all servers	Log on to server
`OP_MKALLDIRS`	`tccd`	Make directories recursively
`OP_MKDIR`	`tccd`	Make a directory
`OP_MKSDIR`	`tccd`	Make save files directory
`OP_MKTMPDIR`	`tccd`	Make a temporary subdirectory
`OP_NULL`	all servers	Discard data successfully
`OP_PUTENV`	`tccd`	Put strings in the environment
`OP_PUTS`	`tccd`	Write strings to a text file
`OP_RCFNAME`	`tetxresd`	Return configuration file name
`OP_RCOPY`	`tccd`	Copy save files locally
`OP_RCVCONF`	`tccd`	Receive configuration information
`OP_RESULT`	`tetxresd`	Send a test purpose result
`OP_RMALLDIRS`	`tccd`	Remove directories recursively
`OP_RMDIR`	`tccd`	Remove a directory
`OP_RXFILE`	`tccd`	Remote file transfer
`OP_SETCONF`	`tccd`	Set configuration mode
`OP_SHARELOCK`	`tccd`	Create a shared lock
`OP_SNDCONF`	`tccd`	Send configuration information
`OP_SNGET`	`tetsyncd`	Get sync ID for an auto sync session
`OP_SNRM`	`tetsyncd`	Remove an auto sync session
`OP_SNSYS`	`tetsyncd`	Send auto sync system name list
`OP_SYSID`	`tccd`	Assign system ID
`OP_SYSNAME`	`tccd`	Send system name list
`OP_TFCLOSE`	`tetxresd`	Close a transfer file
`OP_TFOPEN`	`tetxresd`	Open a transfer file
`OP_TFWRITE`	`tetxresd`	Write to a transfer file
`OP_TIME`	`tccd`	Return the system time

Request code	Processed by	Description
`OP_TPEND`	`tetxresd`	Signal TP end
`OP_TPSTART`	`tetxresd`	Signal TP start
`OP_TRACE`	all servers	Send trace flags
`OP_TSFILES`	`tccd`	Transfer save files
`OP_TSINFO`	`tccd`	Send transport-specific information
`OP_UNLINK`	`tccd`	Unlink a file
`OP_USYNC`	`tetsyncd`	User sync request
`OP_WAIT`	`tccd`	Wait for a process
`OP_XRES`	`tetxresd`	Write data to an execution results file
`OP_XRCLOSE`	`tetxresd`	Close an execution results file
`OP_XROPEN`	`tetxresd`	Open an execution results file and return an xres ID
`OP_XRSEND`	`tetxresd`	Associate xres ID with this logon
`OP_XRSYS`	`tetxresd`	Send system name list

F.5 Server reply codes

Each reply sent by a server to a client process includes a reply code. This code indicates whether or not the corresponding request was successful.

A list of server reply codes and their meanings is presented in the appendix entitled ‘‘Server reply codes’’ elsewhere in this guide.

F.6 Sync states

`tetsyncd` maintains a set of sync states for each sync event. One state in this set is maintained for each system that is expected to participate in the event.

A list of sync states and their meanings is presented in the chapter entitled ‘‘Test case synchronisation’’ elsewhere in this guide.

F.7 Execution result states

`tetxresd` maintains a set of process states for each execution results file. One state in this set is maintained for each system that is expected to contribute to the consolidated result of a distributed test purpose.

The following table lists execution results states and their meanings:

Execution results state	Description
`DEAD`	The TCM on this system has disconnected from `tetxresd`
`NOTREPORTED`	The system has not submitted a result for the current test purpose
`REPORTED`	The system has submitted a result for the current test purpose

G. Trace and debugging facilities

G.1 Introduction

Each TETware program contains certain facilities which may be used by an experienced software engineer to trace its operation and for debugging purposes.

When tracing facilities are enabled, messages describing the processing being undertaken are printed to the standard error stream. Although an explanation of the meanings of these messages is beyond the scope of a User Guide, the information in this appendix is presented in the belief that it may be of some use to experienced TETware users.

G.2 Caveats

Trace messages are intended to be interpreted by an experienced software engineer who is familiar with the internal operation of TETware processes. In most cases, they need to be interpreted in conjunction with the TETware source code.

Some of the tracing facilities generate huge volumes of output, so you should ensure that you are prepared to handle the output before you enable tracing. In most cases, it is best to enable only those messages that you actually need to diagnose a particular problem. Whenever possible, you should invoke tracing while processing only a small scenario containing a few simple test purposes, rather than attempting to trace the processing of a fully-featured test suite.

G.3 Description

In each TETware process there are several trace flags that can be used to control the emission of debug output. Each trace flag has a value associated with it; generally speaking, the higher the value, the greater the volume of output produced.

The following table lists these flags and the trace message types controlled by them. The flags marked with a † are not present in TETware-Lite.

Trace flag name	Trace message type
`tet_Tbuf`	memory allocation
`tet_Texec`	operation of the `tcc` execution engine
`tet_Tio`†	message i/o operations
`tet_Tloop`†	start and end of the client and server loops
`tet_Tscen`	operation of the `tcc` scenario parser
`tet_Tserv`†	generic server operation
`tet_Tsyncd`†	`tetsyncd` functions
`tet_Ttcc`	miscellaneous `tcc` functions
`tet_Ttccd`†	`tccd` functions
`tet_Ttcm`	TCM functions
`tet_Ttrace`	trace subsystem operation
`tet_Txresd`†	`tetxresd` functions

In addition, when the `tet_Ttrace` flag has a non-zero value, a time-stamp is included in each message generated by the trace subsystem.

Trace flags can be set from the command line (for `tcc`, `tccd`, `tetsyncd`, `tetxresd` and stand-alone programs) and can also be passed to servers in an `OP_TRACE` message. TCM processes may receive trace flags from `tcc` and `tccd` in the `TET_TIARGS` environment variable.

At present, trace flags are propagated as follows:

- `tcc` passes trace flags to `tetsyncd` and `tetxresd` on the command line, to `tccd` in an `OP_TRACE` message, and to TCM processes in `TET_TIARGS`.
- `tccd` passes trace flags to TCM processes in `TET_TIARGS`.
- TCM processes pass trace flags to `tccd` in an `OP_TRACE` message.
- On a Windows NT system, `tccdstart` passes trace flags to `tccd` on the command line.

Each process maintains two copies of each trace flag; one (the global value) is passed to other processes as required, while the other (the local value) is made available in the flag variables described above. It is therefore the local value that controls the emission of debug information.

Each global value has a set of bits associated with it which determines to which other processes it will be propagated. If a process receives a trace flag (together with its associated set of bits) and one of the bits matches the process type of the receiving process, then the process copies the global value into its local value.

When a trace flag appears on the command line, its format is as follows:

command `-T`[*X*. . . `,`]*yn* . . .

More that one `–T` option may appear on a command line. The meanings of `–T` suboptions are as follows:

X. . . `,` This part is optional but, if it appears, it is a comma-terminated process indicator list indicating to which process the flag should be propagated.

The following process indicators are understood:

`M`	`tcc`
`S`	`tccd`
`C`	master TCM/API[16]
`D`	slave TCM/API[17]
`X`	`tetxresd`
`Y`	`tetsyncd`
`T`	stand-alone programs[18]

16. A TCM is a **master** TCM if it is either managing a non-distributed test case or managing the part of a distributed test case that is running on the first (or only) system that is specified in the test case's system list.
17. A TCM is a **slave** TCM if it is managing part of a distributed test case that is running on the second or subsequent system that is specified in the test case's system list.
18. For the purposes of this description a **stand-alone** program is one that is not one of the components of the TETware architecture. For example, the TCC daemon bootstrap program (`tccdstart`) which is used on the Windows NT system comes into this category.

y Indicates which trace flag should be set. This is a single trace flag indicator, or `all` to indicate that all flags should be set.

The following trace flag indicators are understood:

`b`	`tet_Tbuf`
`c`	`tet_Ttcm`
`e`	`tet_Tserv`
`g`	`tet_Texec`
`i`	`tet_Tio`
`l`	`tet_Tloop`
`m`	`tet_Ttcc`
`p`	`tet_Tscen`
`s`	`tet_Ttccd`
`t`	`tet_Ttrace`
`x`	`tet_Txresd`
`y`	`tet_Tsyncd`

n The value to which the flag should be set. If no value is specified, it defaults to 0.

Function tracing is performed by calls to the `TRACE`*n*`()`, `TDUMP()` and `BUFCHK()` macros.

G.4 Examples

Here are some examples of how the user can set trace flags from the `tcc` command line:

`tcc -Tp4 ...`	Trace certain operations that are performed by `tcc`'s scenario parser by setting the `tet_Tscen` flag to 4.
`tcc -TS,s4 ...`	Set the global `tet_Ttccd` flag to 4 and pass it to `tccd`, which copies the global flag to its local flag.
`tcc -TXY,i6 ...`	Set the global `tet_Tio` flag to 6 and pass it to `tetsyncd` and `tetxresd`, each of which copy it to their local flag.
`tcc -Ts2 ...`	Set both the local and global `tet_Tserv` flags to 2; the global flag is passed to all processes, each of which copies it to its local flag.
`tcc -Tall10 ...`	Set both the global and local versions of each flag to 10. Each global flag is passed to all processes, each of which copies it to its local flag.

Trace options may also be passed on the command line when `tccd` is started on a particular system; however, when this is done, trace flags are only propagated to processes that are started by `tccd` on that system.

On a Windows NT system, trace options may be specified for `tccd` by including them on the `tccdstart` command line.

H. Implementation notes for TETware on Win32 systems

H.1 Introduction

TETware has been implemented on the Windows NT and Windows 95 operating systems, as well as on the UNIX systems on which previous TET versions have been implemented. In this appendix and 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.

This appendix contains details of some features which are specific to the Win32 implementation of TETware and discusses some of the ways in which the implementation of TETware for Win32 differs from the UNIX implementation.

H.2 Supported TETware versions

TETware-Lite is supported on both Windows NT and Windows 95. Distributed TETware is supported only on Windows NT.

H.3 File naming conventions

H.3.1 Directory separator character

On a UNIX system, the `/` character is used as the directory separator character whereas, on a Win32 system, either the `/` or the `\` character may be used as the directory separator character.

While TETware will interpret file names correctly which use either character, it is recommended that you only use `/` as a directory separator character. This is particularly important when specifying a file name in a configuration file on a Win32 system which might be interpreted on a UNIX system when a remote or distributed test case is to be processed.

H.3.2 Full path names

When you specify a full path name on a Win32 system, you can specify it either as `/`*path*`/`*file* or as *x*`:/`*path*`/`*file*.

If you do not include a drive specifier[19] in the path name, the current (or **default**) drive will be used. This will work as expected provided you do not specify file names on more than one drive in connection with a particular test suite. If you specify file names on more than one drive in connection with a particular test suite, all full path names specified for that test suite must include a drive specifier.

19. That is: the initial *x*`:` sequence.

H.3.3 Relative path names

When you specify a relative path name on a Win32 system, you can specify it either as *path*`/`*file* or as *x*`:`*path*`/`*file*.[20]

You should not specify a relative path name that includes a drive specifier, since unpredictable results can occur when a TETware process attempts to use such a path name after changing its current working directory. For the same reason you should not specify a simple file name as *x*`:`*file*.

H.3.4 Test case names

The format of a test case name in a scenario file is defined by the syntax of the scenario language. This syntax is not modified by the file naming conventions of the host operating system.

Thus a `/` must always be used as the directory separator character when a test case name appears in a scenario file. `\` is not accepted as a directory separator character in a test case name on Win32 systems.

H.4 Executable files

On a Win32 system the execute bit in a file's permission is not significant when determining whether or not a file is executable. Instead the system determines how to execute a file from the file name suffix. The `spawn` and `exec` family of routines in the Microsoft C runtime support library understand certain suffixes as indicating executable files and will invoke the appropriate command interpreter when called upon to execute a `.cmd` or a `.bat` file.

In addition to the suffixes recognised by the C runtime support library, the TETware test case and tool execution subsystem recognises a file with a `.ksh` suffix as a Korn Shell script and a file with a `.pl` suffix as a Perl script. The execution subsystem will invoke `sh.exe` or `perl.exe` as appropriate when called upon to execute one of these files. These interpreters are located using the `PATH` environment variable, so script execution will be unsuccessful if the search path does not include the location of the interpreter that is required to execute the script.

The way in which file names are interpreted on a Win32 system affects the way in which you should specify a test case name in a scenario file, or a tool name using one of the tool-specific configuration variables, as follows:

- To specify a test case or tool which is a `.com`, `.exe`, `.cmd` or `.bat` file, you may either include or exclude the file's suffix.
- When you specify a Shell (`xpg3sh`) or Korn Shell (`ksh`) API test case to be executed, you must either arrange for the file name to have a `.ksh` suffix or set the `TET_EXEC_TOOL` configuration variable to `sh`.
- Likewise, the file name of a Perl test case must either have a `.pl` suffix or you must set the `TET_EXEC_TOOL` configuration variable to `perl`.
- If you specify a test case without a suffix in a scenario file and you use a command such as `make` to build and/or clean the test case, you must arrange for the build and clean tools to

20. Note that the second form does not have a `/` character after the drive specifier.

append the correct suffix to the test case name before passing it to the `make` command.

- If you specify a tool which is a shell script, you may either name it with a `.ksh` suffix, or you may specify `sh` as the tool and the script as the tool's instruction file. For example, if you specify a build tool which is a shell script, you can make one of the following assignments; either:

  ```
  TET_BUILD_TOOL=mybuildtool.ksh
  ```

  or:

  ```
  TET_BUILD_TOOL=sh
  TET_BUILD_FILE=mybuildtool
  ```

  When the name of the build tool does not contain a directory separator character, TETware locates the build tool using the `PATH` environment variable. When a build file is specified, it should either be specified relative to the test case source directory or should be specified as an absolute path name, so as to enable it to be accessed when the build tool is executed.

  The same considerations apply to the way in which you make assignments to the clean tool and its optional instruction file, and the optional prebuild, build fail and exec tools and their optional instruction files when any of these tools are shell scripts.

- Likewise, if you specify a tool which is a Perl script, you may either name it with a `.pl` suffix or you may specify `perl` as the tool and the script as the tool's instruction file.

Whether or not you decide to specify suffixes or use tools to add them depends on the type of testing that you want to perform. Some issues to consider are as follows:

- If you are developing tests which will only run on Win32 systems, it is probably most simple to use suffixes throughout.
- If you are porting tests from a UNIX system to a Win32 system and need to maintain a common source base, it is probably best not to use suffixes. Instead you should use a build or clean tool to append a suitable suffix to the test case name at the start of processing, and an exec tool to perform the correct type of test case execution. Note that this will require some care if you are processing a scenario which contains a combination of executable and interpreted test cases.
- If you are processing remote or distributed test cases on a combination of UNIX and Win32 systems, you must either arrange for the names of the test cases to be the same on all systems or use tools to perform the required name translations on Win32 systems.

An example of a possible solution to this problem is contained in the Distributed C API demonstration test suite, which is included with the TETware distribution.

It should be noted that the `#!` convention that may be used to select the interpreter for script files on many UNIX systems is not understood by the C runtime support library on a Win32 system. It is particularly important to remember this point when porting a TETware test suite from a UNIX system to a Win32 system.

H.5 Application types

Each TETware program which runs on a Win32 system is compiled as a console application. Test cases compiled as Win32 GUI applications are not supported by TETware.

H.6 Signal handling

H.6.1 Keyboard signals

On a UNIX system it is possible to instruct `tcc` to abort a test case or a test run by generating an interrupt or a quit signal from the keyboard. On a Win32 system these instructions may be sent from the keyboard to `tcc` by using control-C and control-BREAK. These keystrokes are mapped by the C runtime support library on to `SIGINT` and `SIGBREAK` signals, respectively, which cause the appropriate action to be taken when caught by `tcc`.

It should be noted that the Microsoft documentation for the `signal()` function states that `SIGINT` is not supported for any Win32 application which runs on Windows 95 or Windows NT. It would appear that the same caveat applies to `SIGBREAK` as well. Therefore the reliable operation of these functions cannot be guaranteed and it is possible that unpredictable behaviour may occur when a test case or test run is interrupted by control-C or control-BREAK.

H.6.2 TCM signal trapping

The C, C++ and Perl TCMs do not attempt to trap unexpected signals on Win32 systems.

H.7 Context numbers

H.7.1 Introduction

When a Test Case Information line is printed in the journal, one of the subfields in the second field contains a context number. The TET specification requires this context number to be unique during the lifetime of each test purpose. Traditionally the context number has been derived from the process ID of each process which make up a test purpose.

It is possible for a test purpose to launch one or more child processes and the purpose of the context number is to distinguish between journal output generated by the different processes. In particular, the context number is used when `tcc` reorders the journal lines at the end of each test case invocation.

When a C language test purpose generates a child process on a UNIX system (whether by calling `tet_fork()` or `tet_spawn()`), the API calls `tet_setcontext()` in order to set the context number in the child process to the new process ID.

H.7.2 Generating unique context numbers

The use of the process ID to generate a context number in a child process poses a problem on Win32 systems.

Process numbers are allocated rather more frequently on Win32 systems than they are on UNIX systems. Indeed, it is quite possible for the operating system to re-allocate a process ID as soon as the previous process which used the ID has terminated. So, if a test purpose spawns a child process, waits for the process to terminate and then spawns another child process, it is likely that the two child processes will be allocated the same process ID. When this occurs it is impossible to use a context number based on process ID to distinguish between the journal outputs that have been generated by the two child processes.

In order to overcome this problem, the Win32 version of the child process controller does not make a call to `tet_setcontext()` but instead generates a context number that is based on the system time (in 100ths of a second) as well as on the process ID. The algorithm used makes it very unlikely that two child processes will use the same context number, although that possibility cannot be ruled out completely. In any event, the context number in the child process is guaranteed to be different from the context number in the parent process.

H.7.3 Number of digits in a context number

On many UNIX systems, a process ID can be expressed in five digits. Indeed, the API ensures that the context number contains at least five digits of process ID. However, on a Windows 95 system the process ID is actually an address in high memory, so it is likely that more than five digits will be required to express this quantity.

The TET specification states that a numeric field in a journal file may contain up to ten digits. When a context number is generated by the C API, the first three digits contain the system ID of the originating system and the remaining digits contain the process ID (or, in the case of a child process on a Win32 system, the value calculated as described previously). On a Win32 system the API truncates the process ID part of the context number to seven digits in order to ensure that the maximum number of digits allowed by the specification is not exceeded.

Report writer authors are reminded that they should allow for the possibility that the context number read from the journal on a Win32 system might contain more digits than the five or eight that have traditionally been generated by TET implementations running on UNIX systems.

H.8 Test case termination

When `tcc` needs to terminate the execution of a test case or tool, it does so by means of a call to the `TerminateProcess()` function which is part of the Win32 API. This function is also called by the `tet_kill()` and `tet_remkill()` API functions to terminate a process which is running on a Win32 system.

However, it is understood that the operation of `TerminateProcess()` can be unpredictable and may leave the system in a strange state, possibly causing the system to malfunction at some later time. Therefore, test case authors are discouraged from using `tet_kill()` and `tet_remkill()` to terminate a process which is running on a Win32 system. For the same reason the use of a timeout specified by `tcc −t` to force termination of a test case or tool is discouraged.

H.9 API considerations

H.9.1 Unimplemented interfaces

Some of the interfaces in the C and C++ APIs are not implemented on Win32 systems. Details are presented under the ‘‘Portability’’ heading in the descriptions of each of the API functions in the TETware Programmers Guide.

In particular, `tet_fork()` is not implemented on Win32 systems. This is because the Microsoft C runtime support library does not provide the `fork()` function that is required for the implementation of `tet_fork()`. Therefore, when porting a test case which uses `tet_fork()` from a UNIX system to a Win32 system, it is necessary to re-design the test case to use `tet_spawn()` and `tet_wait()` instead.

H.9.2 Use of the DLL version of the C runtime support library

You should link a test case with the thread-safe versions of the TETware C TCM and C API library if any of the following are true:

— you link the test case with the DLL, multi-threaded C runtime support library;

— you link the test case with any other library (either static or DLL) that has been built to use the DLL, multi-threaded C runtime support library.

You should do this even if you do not intend to use threads in your test case.

This is because:

- The standard TCM/API uses the static, single-threaded C runtime support library.
- The thread-safe TCM/API uses the DLL, multi-threaded C runtime support library.
- It is unwise to mix static and DLL versions of the same library in a single process.

The same considerations apply if you are using the TETware C++ TCM/API.

Note that the use of the static multi-threaded C runtime support library is not supported in TETware.

H.10 The TCC daemon `tccd`

H.10.1 Starting `tccd`

It is not possible to start `tccd` directly from the command line on a Windows NT system. Instead, `tccd` must be started on demand by a bootstrap program called `tccdstart`.

H.10.2 User ID

`tccd` always runs with the user ID of the user who invokes the `tccdstart` bootstrap program. The default user ID of `tet` and the ability to change this using the `−u` command-line option are not supported on a Windows NT system.

H.10.3 File creation mask setting

The `−m` command-line option is not supported on a Windows NT system.

H.11 The TCC daemon bootstrap program `tccdstart`

On a Windows NT system, `tccd` is started on demand by a bootstrap program called `tccdstart`. This program listens for service requests at the port indicated in the **tcc** service specification and starts an instance of `tccd` each time that a connection is received. The `PATH` environment variable is used to locate `tccd` so you should ensure that your search `PATH` includes `$TET_ROOT/bin`.

You can use certain command-line options to modify the behaviour of `tccdstart` and the instances of `tccd` that are launched as connections are received. Refer to the `tccdstart` manual page at the back of this guide for details of the syntax for this command.

You should invoke `tccdstart` once on each Windows NT system on which Distributed TETware is installed, in a new Korn Shell window. When invoked, `tccdstart` prints a message on the standard output similar to the following:

```
     tccdstart: 15 Oct 10:35:20: accepting connections
```

Each time a connection is received on the listen port, `tccdstart` prints a message similar to the following:

```
     tccdstart: 15 Oct 10:35:23: connection received from hostname
```

To stop `tccdstart` you should type a control-C in the window in which `tccdstart` is running. When you do this, `tccdstart` prints a message similar to the following:

```
     tccdstart: 15 Oct 10:37:40: going down on signal 2
```

For the reasons indicated in the section entitled ‘‘Keyboard signals’’ above, it is possible for a race condition to occur in the Winsock library when `tccdstart` is interrupted by a control-C. When this occurs it is possible for the shutdown message to be accompanied by another error message related to the `accept()` function which can safely be ignored.

I. Hints and tips for users of previous TET implementations

I.1 Introduction

The following sections contain information which may be helpful to users experienced with previous TET implementations but not yet familiar with TETware.

I.2 TETware for TET 1.10 and ETET users

1. Two versions of TETware are available; namely, TETware-Lite and Distributed TETware. Of the two, TETware-Lite is the version which most closely resembles TET 1.10 and ETET.
2. The build and installation procedure is completely different for TETware. Be sure to follow the instructions contained in the TETware Installation Guide.
3. TET 1.10 test case binaries and associated scenario and configuration files may be run under the control of both TETware versions of `tcc`, (as non-distributed test cases) without modification.[21]
4. ETET binaries and scenario files may be run under the control of both TETware versions of `tcc`, (as non-distributed test cases) without modification. It may be necessary to include an assignment for the `TET_COMPAT` variable in each mode-specific configuration file[22] in order to cause certain ETET-specific syntax in a scenario file to be interpreted correctly.
5. TET users will benefit from additional scenario directives which are provided to support repeated and parallel test case execution. Both TET and ETET users will benefit from additional directives which are provided to support processing of distributed and non-distributed test cases on remote systems when Distributed TETware is used.
6. TET users will benefit from additional configuration variables which provide better control over the way in which `tcc` interacts with test cases and tools.
7. The C and C++ APIs in Distributed TETware provide additional functions to allow synchronisation between parts of a distributed test case, supply information about local and remote system designations and control execution of processes on remote systems.
8. In Distributed TETware there is an additional configuration file – `tetdist.cfg` – which must be supplied on the local system when remote or distributed testing is to be performed.
9. Test cases built using the C and C++ APIs in Distributed TETware must be run under control of the Distributed version of `tcc`. They cannot be run stand-alone or under the control of the TETware-Lite `tcc`. However, test cases built using the C and C++ APIs in TETware-Lite may be run either stand-alone or under the control of either type of `tcc`.

21. However, you should be aware of the disposition of uncaptured test case and tool output when Distributed TETware is used; this is described in a subsequent note.
22. These files are: the build mode, execute mode and clean mode configuration files.

10. There is no support for distributed testing when the Shell, Korn Shell or Perl APIs are used. However, (non-distributed) test cases which use these APIs may be run stand-alone or under control of either type of `tcc`. In addition, such test cases can be run on remote systems under control of the Distributed `tcc`.

11. When a test case is built using the C and C++ APIs in Distributed TETware, it is best not to call `exit()` directly. Instead, each API provides a function `tet_exit()` which has the same effect; for details, refer to the section entitled ‘‘Executed process functions’’ in the TETware Programmers Guide. In TETware-Lite a call to `tet_exit()` is functionally equivalent to calling `exit()`.

12. When TETware-Lite is used, uncaptured test case or tool output appears on the terminal where `tcc` is invoked, as is the case in TET 1.10 and ETET. However, when Distributed TETware is used, test cases and tools on all systems run detached from the terminal where `tcc` is invoked. Instead, uncaptured output appears in the `tccd` log file on each system where test cases are processed.

13. Since TETware is intended to be used as a replacement for other TET versions, TETware should not be installed with the same value of *tet-root* as that used by an existing TET or ETET distribution; otherwise, some existing files will be overwritten.

I.3 TETware for dTET2 users

1. Two versions of TETware are available; namely, TETware-Lite and Distributed TETware. Of the two, Distributed TETware is the version which most closely resembles dTET2.

2. The build and installation procedure for TETware is similar to that used in dTET2. However, the `make` definition file `defines.mk` is located directly below *tet-root*`/src` rather than at a lower level directory as was previously the case, and the configuration script `dtetcfg` has been renamed `tetconfig` and moved to *tet-root*`/src`.

3. The distinction between master and slave systems has (for the most part) gone. Instead, a system is identified either as the local system (that is, the system on which `tcc` is run), or as a remote system.

4. When Distributed TETware is used, `tccd` must always be run on the local system (as well as on any remote systems) when a scenario contains test cases or test case parts which are to be processed on the local system. Since TETware-Lite cannot process remote or distributed test cases, `tccd` is not required when TETware-Lite is used.

5. It may be necessary to include an assignment for the `TET_COMPAT` variable in each mode-specific configuration file[23] in order to cause certain dTET2-specific syntax in a scenario file to be interpreted correctly.

6. The `distributed` scenario directive enables a distributed test case to be run entirely on remote systems.

7. The API functions `tet_sync()` and `tet_msync()` have been replaced by a new function called `tet_remsync()`.[24] This function provides the test suite author access to

23. These files are: the build mode, execute mode and clean mode configuration files.

24. Although `tet_sync()` and `tet_msync()` will continue to be supported so as to provide backwards compatibility with existing dTET2 test suites, these interfaces are now marked ‘‘to be withdrawn’’ and test suite authors are encouraged to use `tet_remsync()` in new test cases.

all the capabilities of the TETware synchronisation subsystem and is more suitable for use with the symmetrical system model implemented in TETware.

8. There are additional configuration variables which provide better control over the way in which `tcc` interacts with test cases and tools.

9. `tccd` on the local system receives all of `tcc`'s environment variables soon after `tcc` logs on to it. Therefore, environment variables that are in force when `tccd` starts up (including variables specified with `−e` command-line options) may be overwritten by whatever variables happen to be present in the environment inherited by `tcc`.

10. The dTET2 trace subsystem is now used throughout TETware instead of just in the client/server code.

11. Since TETware is intended to be used as a replacement for other TET versions, TETware should not be installed with the same value of *tet-root* as that used by an existing dTET2 distribution; otherwise, some existing files will be overwritten.

J. TETware manual pages

This appendix contains manual pages for TETware programs.