Distributed Test Environment Toolkit Version 2

dTET2 Programmers Guide
Revision 1.0
DT2-PGDE-1.0

Released: 8th October 1993

X/Open Company Limited

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

1. Introduction

1.1 Overview

1.1.1 Intended audience

This guide is intended to be read by programmers who are developing test suites and who have had some exposure to the concepts of system conformance testing.

Systems administrators who are to install dTET2 on their computer systems and software testing engineers who are to use dTET2 to run verification test suites should refer to the dTET2 Installation and User Guide.

1.1.2 Product definition

This document describes the Distributed Test Environment Toolkit Version 2 (dTET2), a set of tools for the development and execution of system and unit tests. The goal behind creating dTET2 and its predecessors was to produce a test driver that would accommodate the future testing needs of the test development community. To achieve this goal, input from a wide sample of the test development community was used for the specification and development of TET's functionality and interfaces.

1.1.3 Background and goals

1.1.3.1 The Test Environment Toolkit

The TET project started in September of 1989, when the Open Software Foundation, UNIX International, and X/Open entered into an announced agreement to produce a specification for a test environment. These organisations agreed to develop and make freely available an implementation written to that specification; additionally, the three organisations committed to producing test suites for execution within that environment.

In the process of developing a specification, the project invited interested members of the test software development community to discuss their requirements for a test driver. It was the belief of the project that, through careful study of these requirements, a reasonably comprehensive and effective test driver could be specified. Having achieved this, the project expected that a substantial portion of the test development community would begin using TET for the development of conformance testing software.

1.1.3.2 The Distributed Test Environment Toolkit

The first set of major extensions to the TET was the Distributed Test Environment Toolkit (DTET) project that started in October 1991. The objective of the project was to extend the functionality of the TET to support the execution of **distributed test cases** and be backwards compatible with the TET. The DTET defined a distributed test case as a test case executing partly on a **master** system and partly on one or more **slave** systems. In such a test case, synchronisation between the test case controlling software on the multiple systems is required.

Initially, the DTET was designed for use as the underlying test harness for the development of a number of network testing requirements, including the X.400 Application Programming Interface (API), the OSF Distributed Computing Environment (DCE) and the X/Open Network File System (XNFS) test suites. Following this, the DTET was installed at other sites and has proved to be portable across a wide range of different systems.

The DTET was also able to execute **non-distributed** test cases (on either the master system or on a single or multiple remote systems). However, to do this the test cases had to be linked with the TET API library `(libapi.a)` and not the DTET library `(libdapi.a)`. Depending on whether you were writing distributed or non-distributed test cases, you had to be aware of which library to use when linking your test case. However, many users found the ability of the DTET to execute TET test cases an advantage because they did not have to recompile or relink their test suites.

1.1.3.3 The Distributed Test Environment Toolkit Version 2

X/Open then wished to enhance the DTET by incorporating all the features of the TET to produce a common toolkit called dTET2. The dTET2 toolkit rationalised the differences in the TET and DTET toolkits by providing:

- A single toolkit for writing distributed and non-distributed tests, using only a single API.
- New Users' and Programmers' Guides.
- Support for the X/Open Transport Interface (XTI) in addition to Berkeley Sockets in the transport-specific parts of the toolkit.
- Fixes to problems inherited from the DTET and the TET.

It is hoped that dTET2 will become the industry standard test harness that is easy to use, well documented and contains sufficient functionality for the broad range of testing required by the industry.

1.1.4 Distributed and non-distributed test cases

An important distinction in dTET2 is whether or not a test case is distributed, defined as follows:

- A test case that contains parts executing on a master system and one or more slave systems is called a **distributed test case** (DTC). In this mode, synchronisation between the test case controlling software is required in order that each part of the distributed test case reach an agreed point in execution.
- A test case that executes only on the master system is a **non-distributed test case**. In this mode, dTET2 behaves similarly to the TET.
- A test case that executes only on a single remote system is a **non-distributed test case**. In this mode, dTET2 behaves similarly to the TET except that the execution of the test case on the remote system is initiated from the master system.

The last definition can be extended to include test cases executing concurrently on multiple remote systems. In this case there is no master system defined and the test cases execute independently of each other. In particular, no synchronisation occurs between the test parts executing on the systems, and no overall test results are established.

In the case of distributed test cases, it is not necessary for the systems participating in the test to be of the same architecture or operating system.

1.1.5 General testing model and methodology

The project developed specifications and an implementation for the following:

- A test case controller.
- A test report generator utility.
- Various Application Programming Interfaces (APIs) to be used in the development of test cases.

The project has tried to allow for future extensibility and flexibility of dTET2 so that test developers can work within the structures of the driver and still produce useful, effective tests. What the project has not addressed is the formulation, content, or scope of test cases. A test case can be arbitrarily complex or simple and still function under dTET2. The structure of a test case can be based on assertions, functions, modules, or any other appropriate abstraction. The test suite developer must choose the most useful structure for a given test suite. This choice will often be based on requirements of various conformance specifications. Because of the overriding concern for providing flexibility for the test developer, dTET2 imposes a minimum amount of structure on test cases (see the chapters entitled ‘‘Testing structure’’ and ‘‘Test suite files’’ later in this guide for information on test suite structure).

1.1.6 Organisation

This manual is organised as follows:

- Chapter 1 describes the manual’s organisation and documentation conventions. It also provides some history regarding dTET2, a discussion of testing methodology, and a feature list.
- Chapter 2 describes the basic structure of test cases and test suites.
- Chapter 3 describes test development, including some basic information about the Application Programming Interface (API) and an explanation of dTET2’s various file formats.
- Chapter 4 explains test execution, focusing on the Test Case Controller (MTCC) and its implementation, the `tcc` command. It describes the MTCC’s various modes of operation and the parameters for configuring dTET2 to meet the needs of different environments.
- Chapter 5 describes how reporting and journaling is performed by dTET2.
- Chapter 6 discusses portability and interoperability. Specifically, it describes how conforming to the API allows test developers to write more portable, platform-independent tests.
- Chapter 7 describes an example test suite that conforms to dTET2’s C language binding of the API.
- Chapter 8 describes an example test suite that conforms to dTET2’s Shell language binding of the API.
- Chapter 9 presents a walk-through of the demonstration distributed test suite that forms part of the dTET2 distribution.
- Appendix A contains a listing of source files that compose the sample C language test suite described in chapter 7.

- Appendix B contains a listing of source files that compose the sample shell language test suite described in chapter 8.

1.1.7 Conventions

The following conventions are used for describing the syntax of commands in this manual:

Convention	Meaning
`|`	The pipe symbol separates syntactic elements from which you must choose one element.
`{ }`	The enclosed syntactic elements can be repeated zero or more times.
`[ ]`	The enclosed syntactic elements are optional in the command.
`( )`	Parentheses enclose items that must be kept together. A group of items can be enclosed in parentheses for clarity or to enforce the desired sequence of processing.
italics	All variables, such as optional command line arguments, appear in italic type.
boldface	Key words are in boldface type.
`Courier font`	Command names, literals, file names, data that you are instructed to enter, output produced by dTET2 or by the operating system (such as error messages), program prompts, and code fragments, appear in courier font type.

Some code examples include line breaks that have been modified to facilitate inclusion of the text in the manual; a backslash (\) at the very end of the line marks these changes. If you type such an example, you should be type it in all on one line and omit the \ character.

1.2 Architecture and features

The architecture of the Distributed Test Environment Toolkit Version 2 provides simplicity and portability, which helps to increase the transportability of test suites. The architecture can be described in terms of four objects:

- The Terminology, which loosely defines the elements of the dTET2 conceptual model.
- The System Requirements, which define minimum system facilities necessary to support this implementation of dTET2 functionality.
- The dTET2 Structure description, which shows how elements in dTET2 implementation link together.
- The Test Suite Structural Requirements, which show how a test suite must be constructed to function within a conforming dTET2 implementation.

These objects are described in the subsections that follow.

1.2.1 Terminology

DTC

Distributed test case.

MTCC

The Master Test Case Controller.

MTCM

The Master Test Case Manager.

STCC

The Slave Test Case Controller.

STCM

The Slave Test Case Manager.

TCC

Either the MTCC or STCC when it is not necessary to distinguish them.

TCM

Either the MTCM or STCM when it is not necessary to distinguish them.

The following terms are used throughout this document.

Application programming interface (API)

An application programming interface is the set of software interfaces between an application and the system. In the case of dTET2, the API libraries (for both the C and Shell languages) offer specific facilities for use by test cases.

Build file

A build file is a set of test suite defined instructions to the build tool for use in constructing executable versions of test cases.

Build tool

A build tool controls how test cases are prepared for execution, such as in creating binaries from source files. If configuration variables are needed by the build tool, it must use the dTET2 API.

Build fail tool

A test suite defined utility which is executed by the test case controller when the build tool indicates that a test case build has failed.

Clean file

A clean file is a set of instructions implemented by the clean tool for use in cleaning up test cases after execution.

Clean tool

A clean tool controls how files or conditions created for or during execution of the test cases are removed, such as in removing binary versions of test cases and any object files that were built when the binaries were created. If configuration variables are needed by the clean tool, it must use the dTET2 API.

Communication variables

Communication variables are used to provide information to the build tool, clean tool, and test cases during execution. Communication variables are restricted to environment variables starting with `TET_`. Most communication variables are only used between the TCC and the Test Case Manager (TCM) component of the API.

Configuration variables

Configuration variables are used to change the execution behaviour of the TCC, the build tool, and the clean tool. Configuration variables are set via configuration variable files and via the TCC user interface. dTET2 defined configuration variable names all begin with the sequence `TET_`. Test suite authors are cautioned to use obvious and consistent naming conventions to avoid potential conflicts with other configuration variables.

In fact, two types of configuration variables are used by dTET2. Those that are used to inform the MTCC of specific details that it needs to interact properly with each of the slave systems, and, those that are used by the MTCC and API to change the operational behaviour of dTET2 in the same fashion as the TET.

Distributed test case

A test case that utilises the distributed features of dTET2.

Exec file

An exec file is a set of test suite defined instructions for use in executing test cases under control of an exec tool.

Exec tool

A tool used for executing test cases under special control, e.g. a debugger. Normally no exec tool is specified, which means test cases are executed directly.

Execution results file

Test cases, via the API, place their results into an execution results file. For distributed and non-distributed test cases the results are written into the file as soon as they are passed to the API. Note that distributed test cases using the master system and the MTCM have a combined execution results file. For non-distributed test cases, entries are transferred from each of the local execution results files to the journal on completion of processing for the test case (build, execute or clean).

Invocable component

An invocable component is the smallest unit that the TCCs can execute individually. Invocable components are made up of one or more test purposes.

Journal

A journal is the file into which test results and tracking data are deposited by the MTCC. This file is later processed by the report generator and a test suite supplied treatment filter to create formatted reports of test results.

Master system

The system from which the building, execution and cleaning of the tests is controlled. This system contains the test scenario for a particular dTET2 invocation and transmits information to each of the slave systems in order that they undertake the necessary tasks as specified in the scenario file. Each dTET2 invocation has exactly one master system.

MTCC

The TCC run by the user on the master system is called the MTCC. When invoked in build mode, the MTCC sends requests, to each of the remote systems involved, to build the relevant parts of the distributed test case. If appropriate, the MTCC will also build those parts of the test case which reside on the master system. The MTCC performs the corresponding functions in execute mode and clean mode. In the absence of a remote directive in the scenario file, the MTCC behaves in a manner identical to the TET TCC.

The dTET2 MTCC is implemented by the `tcc` command.

MTCM

The MTCM is a component of the dTET2 API. This component interfaces between the MTCC and the test suite specific part of the test case which is executing on the master system.

Prebuild file

A prebuild file is a set of test suite defined instructions to the prebuild tool for use in preparing for the building of executable versions of test cases.

Prebuild tool

The MTCC uses a test suite defined prebuild tool to undertake the preparation for the distributed build. The prebuild phase is only undertaken on the master system, whereas the build phase is undertaken on the master system and each of the slave systems concurrently. The prebuild phase is optional and allows actions such as the transfer of source files from the master to the slave systems in preparation for the local build phase. (This assumes that a file transfer mechanism is available, which is not required by the dTET2 specification).

Result code

A result code is the determination made by a test purpose as to the status of the test it performed. dTET2 supports the result codes defined by IEEE Std 1003.3–1991, as well as additional, test suite defined result codes. The status of test cases, represented by result codes, is recorded via the API, and can be analysed by the report generator and appropriate treatment filters.

Scenario file

A scenario file is a file containing test scenario definitions.

Slave system

The system on which a remote part of the build, execution and cleaning of the test case is undertaken. The remote part may be a stand alone test case or part of a distributed test case. A dTET2 invocation may have any number of slave systems according to the parameter directives in the scenario file. The slave system may be the same as the master system.

STCC

The STCC is the test case controlling interface on the slave system. It receives requests from the MTCC to undertake the local part of the build, clean or execute. The actions of the STCC will be initiated on the receipt of a request from the MTCC.

The dTET2 STCC is implemented as a daemon process (`tccd`) that is installed on each slave system expected to participate in testing. It is normally started automatically from an entry in the system initialisation file `/etc/rc`, or from an entry in the `init` configuration file `/etc/inittab`, or on demand from an entry in the `inetd` configuration file `/etc/inetd.conf`.

STCM

The STCM is a component of the dTET2 API. This component interfaces between the STCC and the test suite specific part of the test case which is executing on the slave system. The STCM participates in synchronisation between the slave and the master system.

Synchronisation

The process of ensuring that each part of the distributed test case residing on the master and slave systems has reached an agreed point in its execution. Certain synchronisation points are negotiated automatically by the STCM and MTCM (e.g. at test purpose start) while others are test case specific and are negotiated by the master with one or more of the slaves.

System ID

dTET2 systems are identified by a three-digit system identification. The system IDs are mapped to some information (such as host names) in the `systems` file which must be located in the test suite root directory on each system participating in the test. The exact format of this mapping is transport dependent.

The master system always has system ID `000`. All other system IDs refer to slave systems.

Test case

A test case is the software which conducts tests. The scope of the term ‘‘test’’ is broad. It may range from a single test purpose for a single function being tested, all the way to a complete suite of conformance tests for a specification. The TCC builds test cases when invoked in build mode, executes the invocable components within test cases when in execute mode, and cleans up any unwanted files when in clean mode.

Test case controller (TCC)

The TCC is the tool that provides structure and control for test cases. The tool handles such functions as sequencing of invocable component execution, unexpected event handling, cleanup, parameter passing, and transferring of test case execution results into the journal.

See also MTCC and STCC (master and slave TCCs).

Test case manager (TCM)

The TCM is a component of the API. This component acts as a ‘‘wrapper’’ for test cases, providing interpretation of the command line, selection of invocable components, and support for the automatic sequencing of test purposes and invocable components, as well as insulation from spurious signals.

See also MTCM and STCM (master and slave TCMs).

Test purpose

A test purpose is the software that represents the smallest level of granularity of a test specification. A test purpose always leads to a single result. In the case of an IEEE Std 1003.3–1991 conforming test suite, for example, test purposes would correspond to assertions.

Test scenario

A test scenario is a sequence of one or more invocable components associated with a single user-exposed name. When the TCC is invoked with a scenario name, all invocable components associated with it are built, executed, and/or cleaned depending on the TCC mode selected.

Test suite

A test suite is a set of test cases and other required and optional files. A test suite must contain at least one test case.

Test suite installer

The test suite installer is used to execute an installation tool supplied with the test suite. dTET2 does not provide this tool; instead test suite authors are responsible for providing and documenting the installation procedures.

1.2.2 Basic system requirements

To be able to build dTET2, the target operating system should be compliant with the following:

- POSIX.1. IEEE Std 1003.1–1990, *Portable Operating System Interface for Computer Environments*, with either standard ‘C’ or common ‘C’ support.
- XPG3, Vol. 1. *X/Open Portability Guide Issue 3, Volume 1: XSI Commands and Utilities* (for the `shell` and `rm` commands)

In addition to this, at least **one** of the following network interfaces is required. When building dTET2, a choice of which transport interface to use is required.

- The set of network-related system calls and/or library routines, generally known as *Berkeley sockets* that provide and interface to TCP/IP network facilities. This interface is typically implemented as in BSD4.3 and SunOS.
- XPG3, Vol. 7. *X/Open Portability Guide Issue 3, Volume 7: Networking Services* (for the X/Open Transport Interface, XTI). For dTET2, a connection oriented transport service is required; this can be implemented over TCP or OSI layer 4 services. There is no reliance on any specific class of service, expedited data, or any other transport specific facility.

  Further information on systems requirements can be found in the dTET2 Installation and User Guide.

1.2.3 Conceptual model

The following two, complete, diagrams describe the files and utilities that interact in the processing of a distributed test case. When a `remote` directive is specified that does not include the master system, the Master Test Case and its associated connections become redundant. For cases where a `remote` directive is not being processed the conceptual model consists of the master system only with no interaction with other systems. In this case, it is similar to that of TET 1.10.

For a more concrete picture of what the various data files can look like and how they relate to each other, see the chapter entitled ‘‘A simple distributed test suite’’ later in this guide.

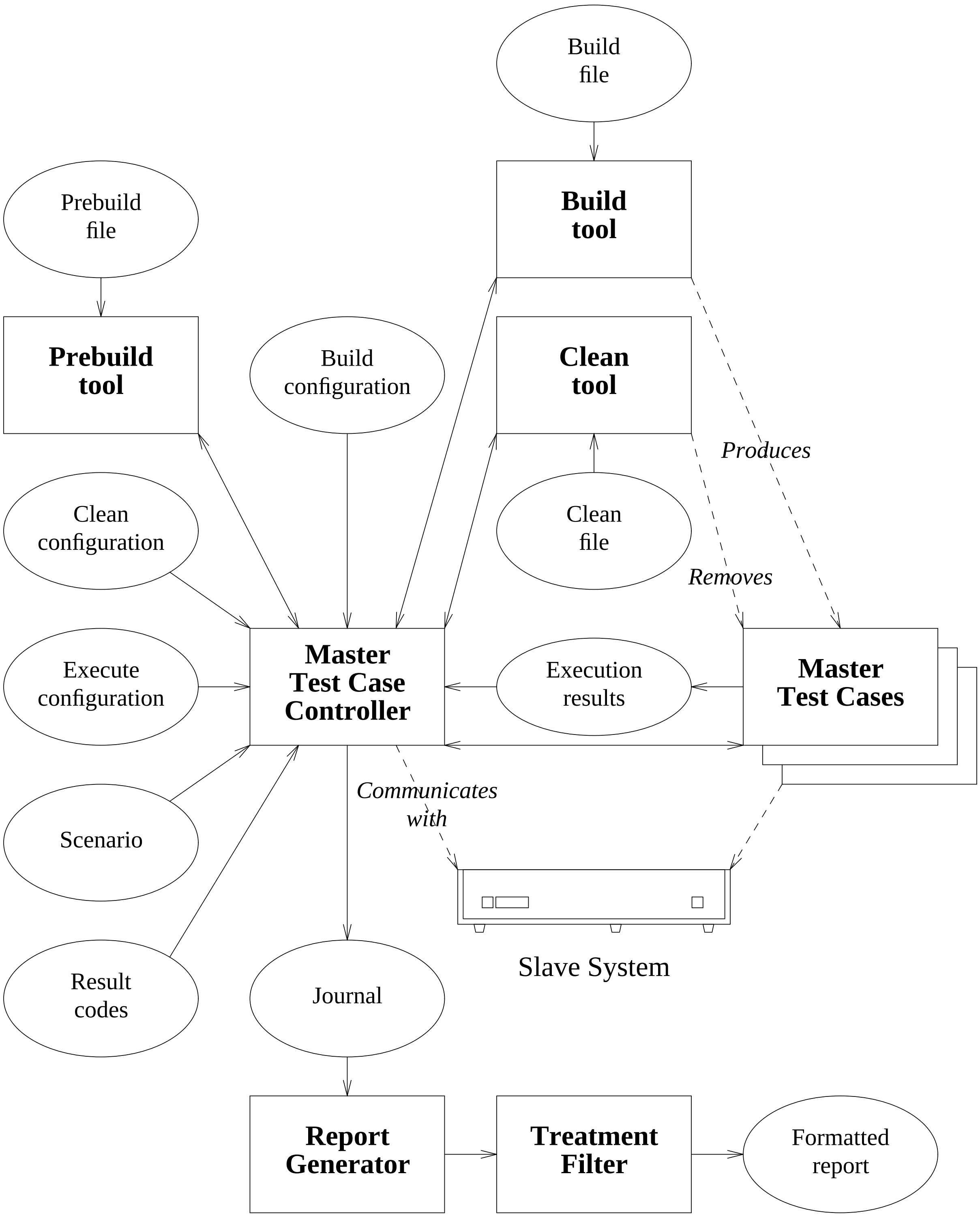

Figure 1. dTET2 conceptual model — master system

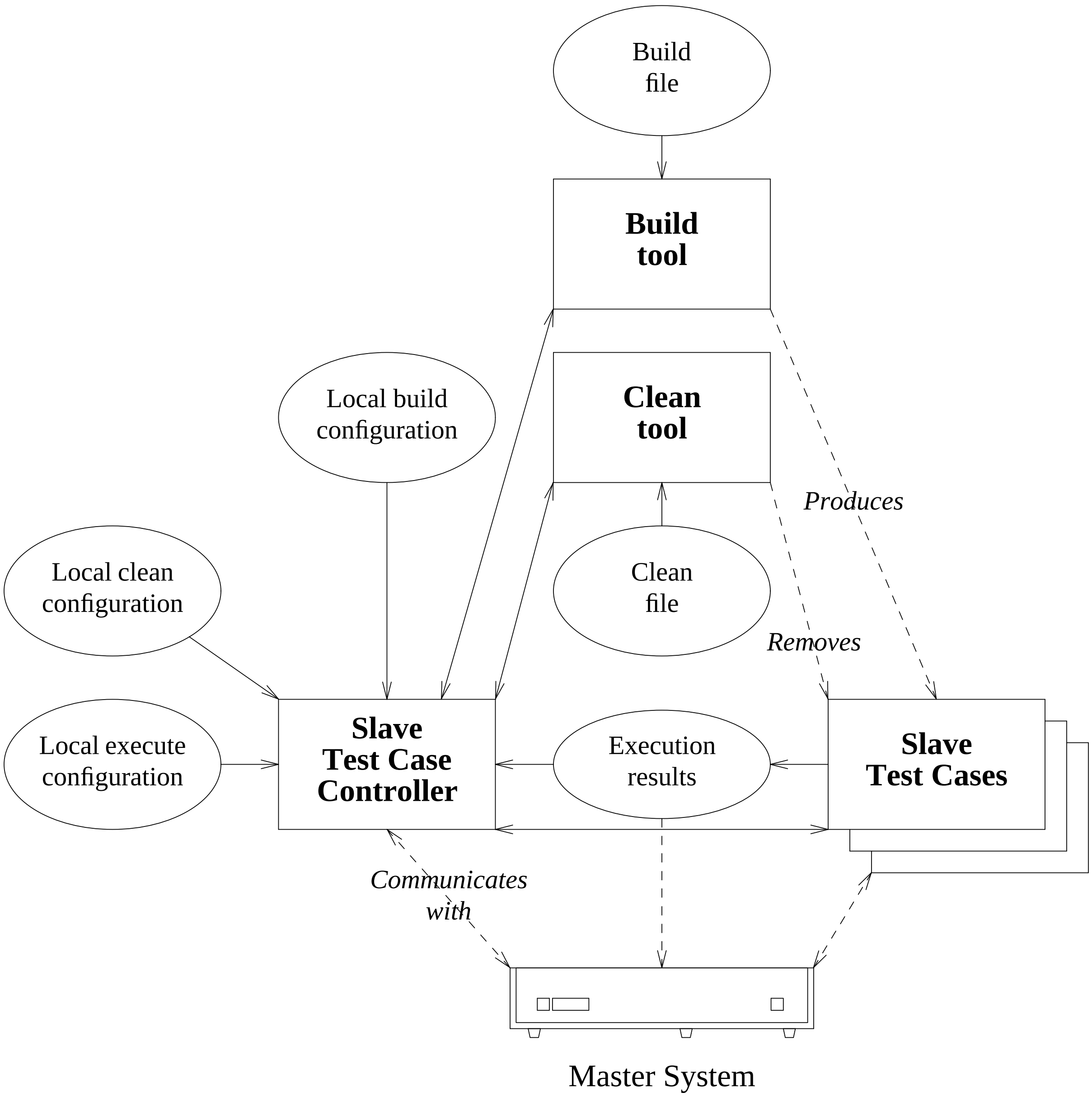

Figure 2. dTET2 conceptual model — slave system

1.2.4 High-level features

The following list highlights the basic features of dTET2.

- **Installation:** Each test suite can include its own installation tool(s). This allows the test suite to have maximum control of its environment and configuration during installation.
- **Configuration:** The TCCs allow for the specification of system specific details at execution time. This means that you can easily change the test suite configuration without changing the individual test cases. Different configurations might include additional libraries required for compilation, different compiler names, alternate header hierarchies, or a selection of various test suite options.
- **Binary Testing:** dTET2 is capable of executing test cases that are distributed as binary executables.
- **Result Codes:** dTET2 can extend its base set of result codes with a test suite specific set; and the action associated with any result code can be changed.
- **Report Generation:** dTET2's report generation utility executes test suite or user supplied report generation filters.

1.2.5 Test Case Controller features

The following list highlights features specific to the master TCC. The master TCC is the main user interface to dTET2.

- **Selective Execution:** The MTCC allows users to select which invocable components are to be run. The names of invocable components must be made available to dTET2 by the test suite. It is common for test suites to come preconfigured with specific sequences of invocable components (test scenarios). Users can also construct their own test scenarios.
- **Test Interruption:** It is possible to interrupt execution of either a test case (by sending a SIGINT to the MTCC), or the test scenario (by sending a SIGQUIT to the MTCC). The execution can be resumed later.
- **Restart of Interrupted Tests:** An interrupted test scenario can be restarted at the same invocable component in the test scenario.
- **Test Case Build/Run:** The TCCs allow all test cases in a test scenario to be built before they are all executed (by means of two successive invocations of the MTCC). Alternatively, individual test cases can be built, run, and cleaned up before subsequent test cases are started. In all cases, unless specifically instructed otherwise, the TCCs keep test case source files separate from the actual tests being run.
- **Selection Criteria:** The criteria for selecting which tests are to be executed by the TCCs are readily definable by users through test scenario creation or by selection on the `tcc` command line.
- **Isolation:** The TCCs do not prevent individual test cases from being executed in parallel. However, each test case execution is protected from like executions through locks placed in the execution directory and (optionally) through copying test cases and associated files to a temporary directory prior to execution.

  The TCCs provide cleanup of test case space, which prevent test cases from corrupting each other or inducing inter-test data collisions.

- **Black-Box Testing:** Test cases communicate with the TCCs only through the API. This ensures that users are verifiably kept separate from test purpose results.
- **Audit Trails:** In addition to providing test cases with journaling, the MTCC marks in the journal the start of each test case execution and the invocable components requested.
- **Test Progress Tracking:** Progress of an executing test scenario can be tracked.
- **Interactive Testing:** The MTCC allows non-distributed tests to interact with an operator. Interactive distributed and remote test cases are not supported.

1.2.6 API features

- **Sequencing:** The TCMs ensure that invocable components are processed in the sequence defined in the test scenario or on the command line.
- **Audit Trails:** The TCMs mark in the results file the number of invocable components expected and executed in each test case, and the number of test purposes expected and executed in each invocable component. Each number executed can be compared against the corresponding number expected by a treatment filter, to ensure that all test purposes returned meaningful result codes.
- **Stand-Alone Tests:** The API does not allow a test to be built in such a way that it can be executed without the MTCC. Such an execution is not impossible, but it is considered that the amount of effort involved in setting up the environment on (possibly) many systems generally outweigh the advantages of testing in this way.
- **Structured Programming:** The API encourages test developers to write tests in a well structured, consistent, and conforming manner.
- **Dynamic Configuration:** The API allows dynamic configuration of run time parameters. By taking advantage of this feature, test cases can alter their testing strategy without recompilation.
- **Exception Handling:** The API provides exception handling mechanisms to deal with unusual system events that occur while test cases are executing (e.g. spurious signals).
- **Test Synchronisation:** The API provides facilities that assist in synchronising multiple processes associated with a single invocable component. In a distributed test case, synchronisation between test cases executing on multiple systems is supported.
- **Result Code Journaling:** The API provides functionality that records messages and result codes to a journal. These messages are delivered to the journal synchronously to ensure that reports from multiple processes associated with a single test purpose do not collide.
- **Result Code Enforcement:** The API ensures that result codes written to the journal are valid and that only one result code is output by each test purpose. In a distributed test case, the result codes from each part of the test purpose are arbitrated to yield a consolidated result for the test purpose.

1.2.7 Journal features

- **Comprehensive Reporting:** The journaling mechanism outputs information during all modes of TCC execution and provides information which includes test case start and end, invocable component start and end, test purpose start and test purpose results.
- **Result Codes:** The toolkit allows for test suite defined result codes to be added on to those defined in the base set (IEEE Std 1003.3–1991 result codes). In particular, the API and MTCC can process result codes outside the scope of the base set.
- **Journal File Format:** The journal file format is well defined, so that test suite authors and end users can easily and quickly develop tools to manipulate the data.

2. Testing structure

2.1 Test case structure

Test cases are constructed by grouping together low-level routines (called **test purposes**) that test specific system features. These test purposes take advantage of support functions provided by the dTET2 API and are invoked by the dTET2 master and slave TCMs. This relationship is illustrated through the following picture:

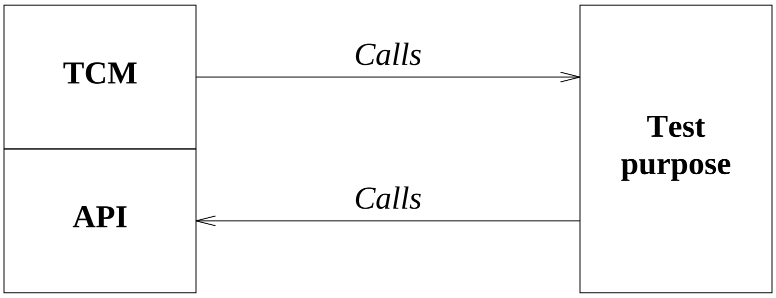

Figure 3. Test case interaction

The picture shows that the TCMs, master and slave as appropriate, call test purposes, and the test purposes in turn call the API. The API performs functions such as fetching configuration variable settings and writing messages into the journal.

Test purposes within a test case can be grouped together into **invocable components**. This ensures that a set of test purposes is always executed together and in the correct sequence. There is no practical limit to the number of test purposes that can be grouped in an invocable component and there is no practical limit to the number of invocable components that can be grouped within a single test case.

The advantage of this structure is that it provides flexibility for developing test cases while still presenting a consistent interface for building and executing test suites.

Details about the functionality provided by the TCM and API can be found in this chapter and the next, which discuss developing and executing tests, respectively. Specific information about constructing test cases can be found in the chapters entitled ‘‘Writing a C language API-conforming test suite’’ and ‘‘Writing a Shell language API-conforming test suite’’, which describe the ‘C’ language and and Shell APIs.

2.1.1 Test Case Manager structure

The TCM is not a separate program, but instead is linked with user-supplied test code and the API library to produce executable test cases. Instructions for linking test cases with the ‘C’ and Shell TCMs (and APIs) are given in the chapter entitled ‘‘Test case support’’ elsewhere in this guide.

2.1.2 API-conforming and non API-conforming test cases

The dTET2 Test Case Controller (`tcc`) is able to execute test cases that do not use the dTET2 API. These test cases are called non API-conforming test cases and are supported for compatibility only. New test cases should use the API.

When executing a non API-conforming test case, `tcc` assumes that the test case writes its results to `stdout` and `stderr` and regards the whole execution as if it were a single invocable component numbered 0 containing a single test purpose. Before executing the test case, `tcc` redirects `stdout` and `stderr` to a file that it creates. When the execution ends, `tcc` generates a result code based on the exit status of the test case. An exit status of `0` means `TET_PASS` and any other value means `TET_FAIL`. Before terminating, `tcc` copies the contents of `stdout` and `stderr` to the journal file, followed by the generated result code.

To indicate that you are executing non API-conforming test cases, you should define the configuration variable `TET_OUTPUT_CAPTURE` with value `True`. For more information on the use of this this configuration variable, see the section entitled ‘‘Configuration variables defined by dTET2’’ elsewhere in this guide.

2.2 Test suite structure

dTET2 imposes minimum structural requirements on test suites. However, some specific files and utilities must be present. A rough representation of this structure follows. The format of the following files is described in the section entitled ‘‘dTET2 file formats’’ elsewhere in this guide.

2.2.1 Required files

Configuration variable settings

dTET2 has several configuration variables that can be used to customise the way in which the master and slave TCCs operate. Test suites may define additional configuration variables as needed. Files containing the test suite settings for these variables are used by the API and TCCs when building, executing, and cleaning test cases.

Test scenario definitions

Each test suite must provide at least one test scenario file. This file (by default named `tet_scen`) contains sequences of invocable components and a related, single name that can be used by the MTCC and dTET2 users when executing the test suite. Many scenario files provide a scenario called `all` which typically lists all test cases and invocable components in a test suite.

Systems definitions file

This is a single file used to define the mapping of system identifier to host name in dTET2, and is located in the *tet-root* directory on each system. The name of the file is `systems`.

Tools directory

Each test suite must have a directory in its top level called `bin`. This directory contains tools that are used by the test suite and by the TCCs.

Results directory

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

2.2.2 Optional utilities

Test suites may provide additional tools for dTET2 users.

Installation utility

No explicit installation utility is provided by dTET2. However, it would be expected that complete installation documentation and procedures be provided by test suite authors. This would include, in particular, test suite specific configuration variables and any other special information required to run the suite.

Build tool

This utility is activated when the TCCs are executed in build mode. It is used to perform any pre-execution functions on the test cases, such as creating binaries from source code. An obvious Build Tool for test suite authors is `make`, although the use of this requires the configuration variable `TET_OUTPUT_CAPTURE` to be enabled.

Clean tool

This utility is activated when the TCCs are executed in clean mode. It is used to perform any post-execution functions on the test cases, such as removing binaries that were created by the build tool or removing temporary files that were created during execution of the test cases. Again, `make` is a good tool for cleaning, although distinct **Clean File**s and **Build File**s will be necessary.

Build fail tool

This utility is activated by TCCs when the build tool indicates that a test case build failed. The build fail tool is invoked in the directory where the build took place with the name of the test case as an argument. One possible use for such a utility would be to provide a skeleton test case that indicates the test case was not built successfully by returning a result of `UNINITIATED`, for example.

Treatment filters

dTET2 produces a well defined journal file containing the results of the test case executions. Test suite authors may wish to provide treatment filters that produce reports in specific formats. The format of the journal file is described in the chapter entitled ‘‘Test reporting and journaling’’ elsewhere in this guide.

3. Test execution

This chapter describes the functionality of the Master Test Case Controller (MTCC). A manual page for the `tcc` command (the implementation of the MTCC) is presented in the dTET2 Installation and User Guide.

The MTCC accepts user-specific and testsuite-specific configuration options and facilitates user control of test sessions. This control includes the building, execution, and clean up of test cases. In addition to the control of test sessions, the MTCC also includes functionality to support internal mechanisms essential to the operation of dTET2. These include sending and receiving requests to and from the slave test case controllers, managing interaction with the MTCM, the processing of results, and the removal of temporary files.

When the MTCC needs to perform an operation on a remote system, it sends a request to the Slave Test Case Controller (STCC) which is running on that system. Each STCC performs operations on behalf of the MTCC when requested to do so and returns control to the MTCC at the end of each request.

3.1 Modes of operation

3.1.1 Description

The MTCC provides modes of operation as follows:

Build mode translates source test cases into executables.

Execute mode loads and executes test cases.

Clean mode removes unwanted files.

The behaviour of these modes can be modified directly by a user, or by configuration options. One special case is the combination of all three of these modes. In that case the MTCC builds, executes, and then cleans each test case prior to moving to the next test case in the test scenario.

Regardless of the mode selected, the MTCC performs the following actions during startup:

1. If a specific journal is named by the user, verifies that it does not exist.
2. Verifies that the test suite selected by the user is available on all systems.
3. Identifies the selected scenario file, and ensures that the scenario selected by the user is present.
4. Identifies configuration variable settings, and integrates custom user settings for those configuration variables.

   In addition to its own local configuration data, the MTCC must also sequentially enter all the configuration data associated with each system identifier referenced anywhere within the scenario.
5. Ensures that the results directory exists on all systems.
6. Tells the user the name of the journal file being used.
7. Writes a start-up message into the journal.
8. Delivers details of option and configuration variable settings for all modes enabled to the journal file.

Should verification fail, for example if the journal file already exists, the MTCC will produce an error message and exit. After startup, the MTCC processes the selected scenario, performing each instruction in turn. The following sections describe MTCC actions in each mode.

3.1.2 Build mode

When the test suite is provided in source form, the MTCC is responsible for building executable versions of test cases. There is no requirement that test suites be provided in source form. Therefore, use of build functionality is at the discretion of the test suite provider and user.

In build mode, the MTCC builds each test case in the specified scenario. For a distributed test case, the MTCC instructs the STCC to initiate the comparable activity on each slave system participating in the test. The processing is as follows:

1. The MTCC delivers a build start message to the journal.
2. Each TCC obtains exclusive locks in the source and execution directories of the test case.
3. Each TCC executes the build tool in the source directory of the test case. If output capture mode is enabled, the arguments passed to the build tool are `TET_BUILD_FILE` and the name of the test case. If output capture mode is disabled, no arguments are passed to the build tool because the tool is expected to be API-conforming and can thus access build mode configuration variables directly. For details of output capture mode, refer to the description of the `TET_OUTPUT_CAPTURE` variable later in this chapter.
4. If output capture mode is enabled, the MTCC transfers the build tool output and error messages to the journal file. If output capture mode is disabled (implying that the build tool uses the API), the MTCC re-orders and copies the contents of the results file to the journal.
5. Each TCC removes the locks set in step 2.
6. The MTCC delivers a build end message to the journal.

If the user has specified an alternate execution directory, each TCC provides that information to the build tool via a communication variable. Build tool and build file authors must ensure that appropriate files are transferred to the alternate execution directory during the build operation.

In build mode, the user is able to:

- Set the scenario file name.
- Set the build configuration file name.
- Set the root directory for the hierarchy of test case executables.
- Set the name of the journal file.
- Set the timeout limit (in seconds) per build tool execution.
- Select the re-run or resume options (see the section entitled ‘‘Major options’’ later in this chapter).
- Set configuration variable values.

3.1.3 Execute mode

In execute mode, the MTCC executes each test case in the specified scenario. For a distributed test case, the MTCC instructs the STCC to initiate the comparable activity on each slave system participating in the test. The processing is as follows:

1. The MTCC delivers a test case start message to the journal.
2. The MTCC initiates the concurrent execution of the test case on all the systems specified.
3. Each TCC obtains shared or exclusive locks in the execution directory of the test case. If `TET_EXEC_IN_PLACE` is false, then the lock is shared; otherwise, the lock is exclusive.
4. If `TET_EXEC_IN_PLACE` is false, each TCC creates a temporary directory in which executions can safely be performed.
5. If `TET_EXEC_IN_PLACE` is false, each TCC recursively copies the files in the test case execution directory to the temporary directory.
6. Each TCC removes the lock set in step 3, if `TET_EXEC_IN_PLACE` is false.
7. If `TET_EXEC_TOOL` is defined, each TCC executes the exec tool with arguments of `TET_EXEC_FILE`, the test case name and the appropriate invocable components; otherwise, each TCC executes the test case directly with arguments of the appropriate invocable components. If `TET_EXEC_IN_PLACE` is true, this execution takes place in the test case execution directory; otherwise, execution takes place in the in the temporary directory.
8. For non-distributed test cases where output capture mode is disabled, the MTCC re-orders and copies the contents of the execution results file to the journal. If any test purpose has not generated a result, the MTCC adds a result message of `NORESULT`.
9. If `TET_TRANSFER_SAVE_FILES` is true, a file transfer mechanism associated with the current transport is used to transfer each of the files matching `TET_SAVE_FILES` to the saved files directory hierarchy on the master system. The saved files directory hierarchy contains subdirectories for each of the remote systems. These subdirectories are named `REMOTE`*nnn*, where *nnn* designates the remote system in the same manner as in the scenario file and the configuration variables.
10. If `TET_TRANSFER_SAVE_FILES` is not provided or is a null string, each TCC treats the `TET_SAVE_FILES` configuration variable in the same manner as in TET 1.10 (each TCC recursively transfers files matching `TET_SAVE_FILES` to the corresponding place in the saved files directory hierarchy on the local system).
11. When processing the parallel directive, the MTCC waits until all the branches of the distributed test case have completed before proceeding to the next stage in processing the scenario file.
12. Each TCC removes the lock set in step 3, if `TET_EXEC_IN_PLACE` is true.
13. Each TCC removes the temporary directory, if `TET_EXEC_IN_PLACE` is false,
14. The MTCC delivers a test case end message to the journal.

In execute mode the user may:

- Set the scenario file name.

- Set the execution configuration file name.
- Set the alternate execution directory.
- Set the root directory for the saved files hierarchy.
- Set the name of the journal file.
- Set the timeout limit (in seconds) per test case execution.
- Set the list of file names which are to be copied to the saved files hierarchy after each test case has completed.
- Select the re-run or resume options (see the section entitled ‘‘Major options’’ later in this chapter).
- Set configuration variable values.

3.1.4 Clean mode

Users can request the MTCC to perform operations on test scenarios to remove unwanted files following test processing sessions. Clean mode will never affect the results of previous test runs.

In clean mode, the MTCC cleans up each test case in the specified scenario. For a distributed test case, the MTCC instructs the STCC to initiate the comparable activity on each slave system participating in the test. The processing is as follows:

1. The MTCC delivers a clean start message to the journal.
2. Each TCC obtains exclusive locks in the source and execution directories of the test case.
3. Each TCC executes the clean tool in the source directory of the test case. If output capture mode is enabled, then the arguments passed to the clean tool are `TET_CLEAN_FILE` and the name of the test case, otherwise no arguments are passed.
4. Each TCC transfers clean tool output and error messages to the journal file, if output capture mode is enabled. If output capture mode is disabled, the MTCC re-orders and copies the contents of the results file to the journal.
5. Each TCC removes the lock set in step 2.
6. The MTCC delivers a clean end message to the journal.

The MTCC waits until the clean has finished on all the systems specified and then sequentially transfers the results of the clean to the journal file.

If any part of the clean has failed, the MTCC will be informed by the appropriate STCC and the whole clean operation will be considered to have failed.

In clean mode the user may:

- Set the scenario file name.
- Set the clean configuration file name.
- Set the root directory for the hierarchy of test case executables.
- Set the timeout limit (in seconds) per clean.
- Select the re-run or resume options (see the section entitled ‘‘Major options’’ later in this chapter).

- Set configuration variable values.

3.2 Major options

3.2.1 Resume option

When invoked with the resume option, the MTCC analyses the old journal file, searching for result codes that match one of a user supplied set. When one of the specified results codes is found in the journal file, the MTCC builds, executes and/or cleans that and all subsequent invocable components in the scenario.

Note that only the result codes for the modes selected are examined; i.e., if only build mode is selected, then execute and clean result codes in the old journal file are ignored. Also note that this option requires the existence of the scenario file that was used to generate the old journal file.

The user specifies the search criteria as a list of result code names and other code letters which the MTCC processes as follows:

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

For a failed test case within a `parallel` directive, the resume option will cause the MTCC to start from the beginning of the appropriate parallel group; that is, as though the scenario started at the `parallel` directive.

For a test case within a `repeat` directive, the resume option will cause the MTCC to start processing the scenario file from the specific test case within the `repeat` directive. Consequently, the matched test case and all the subsequent test cases within the directive will be repeated the number of times specified by the `repeat` directive. Those above the matched test case will be repeated once less.

3.2.2 Re-run option

This option is similar to the previous option with the exception that **only** invocable components with result codes matching one of the user supplied set will be built, executed and/or cleaned.

The following table shows the operations performed on each test case with a matching result code, when the re-run option is used with all possible combinations of build, execute and clean modes:

Modes	Build result match	Execute result match	Clean result match
b	b	-	-
e	-	e	-
c	-	-	c
b+e	b+e	b+e	-
b+c	b+c	-	c
e+c	-	e+c	c
b+e+c	b+e+c	b+e+c	c

For a failed test case within a `parallel` directive, the whole of the parallel group will be rerun.

For a test case within a `repeat` directive, a rerun will cause the MTCC to repeat only that specific test case the number of times specified by the `repeat` directive.

Like the resume option, re-run requires the existence of the scenario file that generated the old journal file.

3.3 Configuration variables

Configuration variables allow the modification of defaults for the master and slave TCCs and for test suites. The mechanism for supporting this functionality is a configuration file per test suite per mode. 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`.

There is also an additional file called `tetdist.cfg` that contains configuration variables used to specify parameters for slave systems that are equivalent to those parameters on the master system that `tcc` obtains from environment variables, or deduces from its current working directory. It may also contain transport specific variables that are required by the network interface used by dTET2. The file should only be provided on the master system.

For each mode the configuration information is obtained from one of these configuration files. This file is located on the master system by the MTCC using the following algorithm:

1. If the user named a specific configuration variable file at invocation time, that file is used;
2. otherwise, if the user named alternate execution directory has a configuration variable file in it, that file is used;
3. otherwise, if there is a configuration variable file in the test suite root directory, that file is used.

On slave systems, the configuration files are assumed to be located in the test suite root directory.

If the user sets specific variables at MTCC invocation time, those settings override the ones in the configuration variable file. When a scenario contains a `remote` directive the variables contained in the master configuration files for build, execute and clean are transferred to each slave system and will form the default values for those systems.

Any variable on the slave system with the same name, that is defined for the same mode, overrides a default variable from the master system.

3.3.1 Considerations for distributed test cases

A special case exists for variables on the master system prefixed with `TET_REM`*nnn*`_` where *nnn* refers to a particular slave system. Configuration variable assignments made on a slave system normally have precedence over those made on the master system; however, a variable defined on the master system whose name whose name begins with `TET_REM`*nnn*`_` defines a new variable or overwrites an existing variable on the slave system which has the same name after the `TET_REM`*nnn*`_` string has been removed.

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

```
TET_BUILD_TOOL=make
```

then, the value of `TET_BUILD_TOOL` will be set to `make` on the master system and on all the slave systems. If the following assignment is made on one of the slave systems:

```
TET_BUILD_TOOL=augmake
```

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

If the following assignments are made on the master system:

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

then the value of `TET_BUILD_TOOL` on slave system 002 is set to `augmake` and the value of `TET_BUILD_TOOL` on the master system and all the other slave systems is set to `make`.

Furthermore, the value of a `TET_REM`*nnn*`_` variable assignment made on the master 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 slave system 002 is set to `augmake` irrespective of any assignment that might be made on the slave system.

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

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

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

When defining the name of a build, exec or clean tool to be executed on a slave system, you should note that the STCC will only be able to execute it if the STCC's `PATH` environment variable is appropriately set. This value will be inherited by the STCC when it is started. In general, this will not be under user control and you cannot rely on it being set up correctly. In such cases it is recommended that you specify an absolute path name for the tool. For example,

```
TET_REM001_BUILD_TOOL=/home/user/bin/augmake
```

3.3.2 Configuration variables defined by dTET2

Configuration data is made available to the test case via the API. The mechanism for providing this information is different for the 'C' and Shell APIs. For more information, see the section entitled ''Manipulating configuration variables'' later in this guide.

Test suites may add their own configuration variables, although variable names beginning with the sequence `TET_` are reserved for future extensions of the toolkit. There are no default values for test suite defined variables. Test suite authors must ensure that in the absence of necessary configuration information their tests behave appropriately.

At present, the following configuration variables are defined:

Variable name	Default value
`TET_BUILD_FAIL_TOOL`	Undefined
`TET_BUILD_FILE`	Undefined
`TET_BUILD_TOOL`	Undefined
`TET_CLEAN_FILE`	Undefined
`TET_CLEAN_TOOL`	Undefined
`TET_EXEC_FILE`	Undefined
`TET_EXEC_IN_PLACE`	`False`
`TET_EXEC_TOOL`	Undefined
`TET_OUTPUT_CAPTURE`	`False`
`TET_PREBUILD_TOOL`	Undefined
`TET_RESCODES_FILE`	`tet_code`
`TET_SAVE_FILES`	Undefined
`TET_TRANSFER_SAVE_FILES`	`False`

In addition, the following distributed configuration variables are defined:

Variable name	Default value
TET_REM*nnn*_TET_ROOT	Undefined
TET_REM*nnn*_TET_TSROOT	Undefined
TET_REM*nnn*_TET_EXECUTE	Undefined
`TET_XTI_TPI`	`/dev/tcp`
`TET_XTI_MODE`	`tcp`
`TET_LOCALHOST`	Undefined

Note that a default of ''Undefined'' means that the variable is not provided unless it is in a test suite or user provided configuration variable file.

These variables are described below:

1TET_BUILD_FAIL_TOOL names a utility that will be executed if the build tool indicates that a test case build fails.

1TET_BUILD_FILE names the file of instructions for the build tool.

1TET_BUILD_TOOL names the build tool.

1TET_CLEAN_FILE names the file of instructions for the clean tool.

1TET_CLEAN_TOOL names the clean tool.

1TET_EXEC_FILE names the file of instructions for the exec tool.

1TET_EXEC_IN_PLACE controls where the test cases are executed (hence this variable applies only to execute mode). When this variable is set to `True`, the TCCs execute each test case ‘‘in place’’ in the directory containing the test case executable; when it is set to `False`, the TCCs first copy the files to a temporary directory before executing them.

This is useful if tests need to retain certain permissions or if their location in the directory hierarchy is important.

For parallel execution of test cases, this flag is ignored. Test cases executing in parallel as a result of the `:parallel:` directive are always executed in temporary directories.

1TET_EXEC_TOOL names the exec tool. Normally this is left undefined, in which case the test case itself is executed. If `TET_EXEC_TOOL` is defined, the named program is executed with `TET_EXEC_FILE` and the test case name as additional arguments preceding any test case arguments.

1TET_OUTPUT_CAPTURE

specifies whether or not test cases use the dTET2 API. If this variable is set to `False`, `tcc` assumes that the test case will use dTET2 API functions to print information lines and register test purpose results. If the variable is set to `True`, `tcc` assumes that the test case writes information lines to `stdout` and/or `stderr` and it arranges to capture this output and transfer it to the journal file when the test case ends. A result code based on the exit value of the test case is then established in which an exit code of `0` means `TET_PASS` and any other value means `TET_FAIL`.

It is not possible to have a distributed test case whose parts consist of a mixture of API-conforming and non API-conforming elements. Hence the value of this variable must be the same for all systems participating in the test.

1TET_PREBUILD_TOOL This variable specifies the pre-build tool to be executed on the master system before executing the `TET_BUILD_TOOL` on each system participating in the test. If it is not defined, the pre-build stage is not executed.

1TET_RESCODES_FILE This variable indicates the name of the file which contains test suite specific result code definitions. The default for this variable is `tet_code` in the root directory of the test suite. Result codes supplied in this file are combined with the dTET2 supplied result codes.

1TET_REM*nnn*_TET_ROOT

The values of these variables specify the locations of **tet root** directories on slave systems. One of these variable assignments must be made for each slave system that may participate in remote or distributed testing. *nnn* indicates the system ID of the system to which the variable is to be propagated.

1TET_REM*nnn*_TET_TSROOT

The values of these variables specify the locations of test suite root directories on slave systems. One of these variable assignments must be made for each slave system that may participate in remote or distributed testing. *nnn* indicates the system ID of the system to which the variable is to be propagated.

1TET_REM*nnn*_TET_EXECUTE

The values of these variables specify the locations of alternate execution directories on slave systems. The use of these variables is optional but, if they appear, they perform the equivalent functions on slave systems to that performed by the value of the `TET_EXECUTE` environment variable on the master system. *nnn* indicates the system ID of the system to which the variable is to be propagated.

1TET_SAVE_FILES

This is a comma-separated list of filenames. If files matching one of these names are found in the execution directory hierarchy after test case execution, they are transferred to the corresponding place in the saved files hierarchy.

1TET_TRANSFER_SAVE_FILES

The setting of this variable determines whether the files defined by `TET_SAVE_FILES` are to be saved on the local system or transferred to the master system and saved there. The way in which files are saved is described in the section entitled ‘‘Saved files processing’’ in the dTET2 Installation and User Guide. Note that this variable is only meaningful on the master system.

The following distributed configuration variables may be specified on the master system when the XTI network interface is used as the underlying transport mode for dTET2:

1TET_XTI_TPI

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

1TET_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.

1TET_LOCALHOST The value of this variable specifies the Internet address of the master system when the XTI network interface is used and the underlying transport provider is TCP/IP. This address is presented in dot notation and must be an address that can be used to access the master system from other machines (i.e., it should not be the address of the loopback interface). All four fields in the address must be specified.

These variables are ignored when the underlying transport interface is not XTI.

3.4 Communication variables

Each TCC must be able to communicate with the other tools it executes (build tool, clean tool, exec tool, and the test cases). The TCCs do this via communication variables. These variables are environment variables, so environment variables starting with `TET_` are reserved. The communication variables currently defined include:

1TET_ACTIVITY the number of activities performed thus far by the TCC. Activities include executions of build tool, clean tool, exec tool, and test cases.

1TET_CODE the path name of the current result codes definition file.

1TET_COM_VAR The path name of the file that contains those variables specified on the MTCC command line.

1TET_CONFIG the path name of the current configuration variable file.

1TET_DIST The path name of the distribution configuration file. This file contains the `TET_EXECUTE`, `TET_ROOT` and `TET_TSROOT` variables for the master system and all the remote systems participating in the distributed test cases. The file also contains any variables that the MTCC uses to establish a connection with the remote system.

1TET_EXECUTE the path name of the root of the execution directory hierarchy.

1TET_ROOT the path name of the dTET2 root directory.

1TET_TIARGS A string of transport independent arguments to be used by the distributed features of the master or slave TCMs. This variable is automatically defined by dTET2 and should not be specified or altered by users.

1TET_TSARGS A string of transport-specific arguments to be used by the distributed features of the master or slave TCMs. This variable is automatically defined by dTET2 and should not be specified or altered by users.

3.5 Journal entries

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

Because a test purpose may have generated other processes that may have written into the execution results file, the MTCC must ensure that the information in the execution results file is ordered correctly when it is transferred into the journal. This ordering is performed using the following algorithm:

1. While the message type is not test purpose start, lines are copied from the combined execution results file to the journal.
2. Once a test purpose start message is found, it is transferred to the journal. Then all lines between that and the test purpose result message (or end of file) are ordered by:
   a. Looking at the first untransferred line in the range. If the line is not a test purpose information line, it is transferred and step (a) is repeated. If it is a test purpose information line, it is transferred and the system ID, context and block identifiers are remembered.
   b. The MTCC then looks through the range for additional test purpose information lines with the same system ID, context and block identifiers. Those are transferred after sorting them in ascending order by sequence number.
   c. When the end of the range is encountered, the MTCC returns to step (a). This process is repeated until all lines in the range are transferred.
3. The test purpose result line is transferred to the journal and the MTCC starts again from step 1. These steps are repeated until the execution result file is exhausted. If there is a missing test purpose result, the MTCC supplies a result of `NORESULT` for that test purpose.

The mechanism used by dTET2 for re-ordering lines attempts to retain blocks of information in the time sequence of their generation. The dTET2 ordering algorithm is based on the use of system identifiers, context, and block identifiers to ensure that journal file entries written from multiple systems can be distinguished and re-ordered. Before writing the journal entry, the system ID is prepended to the context ID. Hence the first three characters of the context field can be used to match the system designation that initiated the journal entry.

3.6 Locking

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

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

- A file `tet_lock` is created in the source directory, using an atomic operation which will fail if a file or directory of that name already exists. If the file cannot be created the locking operation fails.

- If an alternate execution directory hierarchy is in use, a file `tet_lock` is created in the execution directory in the same manner. If the file cannot be created, the lock is removed from the source directory and the locking operation fails.

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

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

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

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

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

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

3.7 Exception conditions

The following exception conditions exist:

Initialisation errors

If the MTCC or STCC does not locate a user specified or required input file upon test session startup, the MTCC will abort and inform the user of the error.

Failure to build, execute or clean

If any specified tools (e.g., build tool, clean tool, exec tool) are not available, the MTCC will generate an end message for that tool with a status of −1 and discontinue processing of the scenario.

At the commencement of processing, the MTCC will log on to each of the STCCs and a failure to complete this task successfully will cause the MTCC to abort after informing any STCC that has already established connection. In this context, ‘‘log on’’ means making a connection and then exchanging system and process identifiers that identify the connected peer.

Each of the slave systems will inform the master system of any error during the build, execute or clean modes. The master system will consider the failure on any slave system in the same manner as for the master itself. This requires that the STCC indicate to the MTCC the reason for the failure.

If, for a specified test case, the build tool on any system returns an abnormal termination, then the MTCC will execute the optional build fail tool.

If in execute mode the MTCC cannot find a test case or the test case is not executable (or one of these is reported by a slave TCC), the MTCC will indicate a test case completion status of −1 and continue processing the scenario.

Test session interruption

This function is requested by using the normal system interruption facilities. Upon receipt of the system interrupt, the MTCC will order the shutdown of the currently executing test case. Test cases written to the dTET2 API will record a TCM message in the execution results file indicating that a termination request was received and terminate. Once the test case execution has stopped, the execution results file is transferred to the journal.

This functionality is implemented on the master system only. The ability to shutdown processes executing on the slave systems depends on the state of the processes on those slaves, and may not be possible until the next synchronisation point, or until a timeout occurs. There is no method of interrupting a slave system and relaying this information back to the master system and from there to each of the other slaves.

Data file corruption

Upon detection of an invalid data file format, the MTCC will abort the test session and issue a diagnostic containing the path name to the journal file and to the user.

Timeouts

A user may specify a timeout period when the MTCC is invoked. Build, execution or cleanup operations which take longer than this timeout (on a per test case basis) will be interrupted by the MTCC. In such a case, the MTCC retrieves any execution results and transfers them to the journal. It then places a message in the journal indicating a timeout and sets the completion status of the test case to −2.

Locking failure

When a locking operation fails and the user has specified a timeout period, each TCC periodically re-trys the operation until the timeout period has elapsed. If the user has not specified a timeout period, or the timeout period elapses, the TCC places a message in the journal indicating a locking failure and sets the completion status of the test case, build tool, clean tool or exec tool to −3.

4. Test suite files

4.1 Installation

The installation procedure for a test suite can be very complex or very simple. dTET2 does not specifically provide any tools to aid in the installation of test suites because such tools are usually unique to particular test suites.

It is essential that thorough installation documentation be provided by the test suite author. In particular, test suite specific configuration variables or other special information should be provided. In most cases it is expected that the installation tool will ask the user to answer some questions, and set test suite configuration variables according to these answers.

4.2 dTET2 file formats

This section describes the format of dTET2's portable data files. These data files, provided by dTET2, test suite authors and users, are portable to all implementations of dTET2.

4.2.1 Configuration variable files

There is a single configuration file per executing test scenario per execution mode. This configuration file is found, by default, in the dTET2 root directory on the master system and in the root directory of the appropriate test suite on slave systems. By default this file is called `tetbuild.cfg`, `tetexec.cfg` or `tetclean.cfg` for each of build, execute, and clean modes, respectively. The names and locations of these files can be changed through options to the `tcc` command. See the chapter entitled ‘‘Test execution’’ elsewhere in this guide for further details.

There is also an additional file called `tetdist.cfg` that contains configuration variables used to specify parameters for slave systems that are equivalent to those parameters on the master system that `tcc` obtains from environment variables, or deduces from its current working directory. It also contains transport specific variables that may be required by the network interface used by dTET2. This file need only be provided on the master system.

In a configuration variable file, variables are defined with lines of the following form:

name=*value*

The *name* may consist of any sequence of alphanumeric characters and the underscore character which does not start with a numeric character. The *value* may consist of any sequence of characters. Blank lines and lines starting with a # are ignored.

Note that names starting with the sequence `TET_` are reserved.

4.2.2 Scenario files

4.2.2.1 Description

Scenario files assign user visible names to one or more invocable components. The default scenario file is called `tet_scen` and is located in the root directory of the test suite on the master system. This file typically defines a scenario named `all`, which should cause execution of every invocable component in the suite. Test cases and **include** files are referenced in scenarios via path names. In build and clean modes, and in execute mode when there is no alternate execution directory, test case path names are relative to the test suite root directory, and may only reference files within the test suite hierarchy. In execute mode when there is an alternate execution directory, test case path names are relative to that directory and may only reference files within the alternate execution directory hierarchy. Include file path names are always relative to the test suite root directory.

The scenario file is made up of multiple data elements. Each data element consists of a name and one or more instructions. Scenario file data elements may continue beyond the end of a line. Continuation of a data element is indicated by beginning subsequent lines beyond column 1. Note that comments (indicated by a # in column 1) do not indicate the end of a data element.

A name consists of alphanumeric characters, and may be up to 31 characters in length.

4.2.2.2 Scenario file data elements

The following subsections describe instructions that may appear in scenario file data elements. Where a *filename* is specified in these descriptions, it names a file that contains test case names, one per line.

4.2.2.2.1 Invocable component execution

These specify the test cases and invocable components to be executed. Multiple invocable components within a single test case may be specified in a single instruction. Invocable component names consist of a leading slash, the path name of a test case (see above), and optional comma separated invocable component names within braces ({,}); for example:

```
/tests/readtests/test1{1,2,3}
```

4.2.2.2.2 Journal message output

This instruction causes a message to be placed into the journal with the type **scenario information**. These lines could be used by a specific report treatment filter to establish check points in a report. Journal messages must be surrounded by double quotes (") and may not contain double quotes; for example:

```
"This is a Journal Message"
```

4.2.2.2.3 Invocable component inclusion

Instructions of this type reference a set of invocable components in another file. The included invocable components are executed synchronously. This facility can be used to reduce the complexity of scenario files. Include directives consist of the colon-delimited term `include` and the path name to include; for example:

```
:include:/tests/readtests/testlist
```

Files referenced by include directives consist of entries. Each is either a journal message (see above) or the path name of a test case with optional invocable components; for example:

```
/tests/readtests/test1{1,3}
```

4.2.2.2.4 Remote system designation

The format of the `remote` directive is as follows:

`:remote,`nnn_1`[,`nnn_2`,...]:[`*filename*`]`

The comma-separated list of remote system designations $nnn_1, nnn_2, \ldots$ defines the remote systems on which the test is executed and consists of a list of one or more numeric **system IDs** as defined in the `systems` file (see below). When the *filename* is omitted, the `remote` directive applies to all subsequent test definitions in the scenario file until an `endremote` directive is encountered.

The special designation `000` is used to refer to the master system. The master system must always be included for any distributed test case. The master system must be omitted when a series of separate, non-distributed, test cases are being executed on one or more remote systems.

4.2.2.2.5 Test case repetition

The `repeat` directive is used to define test cases which are to be repeated. The format of the `repeat` directive is as follows:

`:repeat,`*n*`:`[*filename*]

where *n* is the number of times the test cases are to be repeated. When the *filename* is omitted, the `repeat` directive applies to all subsequent test definitions in the scenario file until an `endrepeat` directive is encountered.

4.2.2.2.6 Concurrent test case processing

The `parallel` directive is used to define test cases which are to be run in parallel. The format of the `parallel` directive is as follows:

`:parallel:`[*filename*]

When the *filename* is omitted, the `parallel` directive applies to all subsequent test definitions in the scenario file until an `endparallel` directive is encountered.

4.2.2.2.7 Valid directive combinations

Not all combinations of directives are valid. The following table describes the legal combinations that can be specified:

Inner directive	**Outer directive**		
	`repeat`	`parallel`	`remote`
`repeat`	Valid	Error	Error
`parallel`	Valid	Error	Error
`remote`	Valid	Valid	Error

When used, the appropriate `end*` directive must occur in the same file as the matching `parallel`, `remote` or `repeat` directive.

4.2.2.3 Example test scenarios

The following subsections contain some example test scenarios.

4.2.2.3.1 Example 1

```
# Example scenario file"

readtest
        /tests/readtests/test1{1,3}

writetest
        /tests/writetests/test1{1,2,3}

all
        /tests/readtests/test1
        "Read tests completed"
        /tests/writetests/test1
        "Write tests completed"
```

In this example, three scenarios are defined called `readtest`, `writetest` and `all`. Since no `remote` directives are included, the scenario specifies a non-distributed test suite involving the execution of test cases on the master system. With a scenario file of this type, dTET2 behaves in the same way as does TET 1.10.

4.2.2.3.2 Example 2

```
all
        :parallel:
        /tests/block/tc
        /tests/rec/tc
        /tests/byte/tc
        :endparallel:
```

This defines a set of three tests that are to be processed concurrently. The master system is not referenced so, like the first example, each test is a non-distributed test case. The scenario could also be described using:

```
all
        :parallel:/tests/par_file
```

where `tests/par_file` is the name of a file containing the details of the three test cases.

4.2.2.3.3 Example 3

```
all
        :remote,000,001,002:/tests/tc_file
```

This specifies that the distributed test cases listed in the file `/tests/tc_file` will be processed on systems `001` and `002`, as well as on the master system.

4.2.2.3.4 Example 4

```
all
        :remote,000,001,002:
        /tests/block/tc
        /tests/rec/tc
        /tests/byte/tc
        :endremote:
```

Each distributed test case in the list will be sequentially processed. The processing of each test case will take place concurrently on the three systems specified.

4.2.2.3.5 Example 5

```
all
        :parallel:
        :remote,000,001:
        /tests/rec/tc
        :endremote:
        :remote,000,001,002:
        /tests/rec/tc
        :endremote:
        :endparallel:
```

In this example, the distributed test case `tests/rec/tc` is to be run with itself on the master system and on system `001`. At the same time, the test case is also run on the master system and on systems `001` and `002`. The implementation of `tests/rec/tc` must ensure that concurrent execution is supported. In particular, if `TET_SAVE_FILES` is set, the implementation should separate those saved files resulting from each instance of the concurrent test case.

For a build, execute and clean of this scenario, the test case `test/rec/tc` will only be built and cleaned once on each system.

4.2.3 Result codes

4.2.3.1 Description

A mapping mechanism for the MTCC is provided to interpret results from test suites. This mapping is contained in a result codes file. Upon termination of a test case, the MTCC determines the action required for each result code and writes an entry in the journal. The API library ensures that the test purposes only generate dTET2 or test suite specified results.

dTET2 provides a default set of result codes; see the ‘‘Results Code Definitions’’ section below. Additional result codes may be defined on a per test suite basis.

4.2.3.2 Results code definitions

The toolkit and test suites both supply result code definitions. Test suite supplied codes are stored in files called (by default) `tet_code` in the test suite root directory on the master system. Blank lines and lines starting with a # are ignored. Other lines in this file contain up to three blank separated fields, defined as follows:

1. The result code. This is a non-negative decimal integer between 0 and 127, inclusive. Result codes from 0 to 31 (inclusive) are reserved for use by dTET2. The remainder are available for specification by the test suite author.

2. The name of this result. This is a field delimited by double quotes which can contain any sequence of alphanumeric characters and blanks.

3. The action to take when this result is encountered. This is an indication of what the TCC and TCM should do when the result is returned by a test purpose. Legal values are `Continue` and `Abort`. The default value for this field is `Continue`.

4.2.3.3 Example results code file

The following is an example result codes definition file. It contains the result codes provided by dTET2.

```
0   "PASS"          Continue
1   "FAIL"          Continue
2   "UNRESOLVED"    Continue
3   "NOTINUSE"      Continue
4   "UNSUPPORTED"   Continue
5   "UNTESTED"      Continue
6   "UNINITIATED"   Continue
7   "NORESULT"      Continue
```

4.2.4 System definitions

4.2.4.1 Description

A single file is used to define the mapping of system identifier to host name in dTET2. This file is located in the *tet-root* directory on each system. It is not required when executing a non distributed test case (that is, the scenario file does not contain a `remote` directive). The name of the file is `systems`.

In the systems definition file, variables are defined with lines of the following form:

sysid host

The *host* descriptor should be a value that can be used by the implementation to establish a connection with the appropriate system.

For implementations of dTET2 that use the X/Open Transport Interface (XTI), an additional variable must be defined for each system. This variable contains the protocol-specific address string that is to be passed to the transport provider by XTI and identifies the address of a slave TCC. Thus, when the XTI transport interface is used, the format of an entry in the `systems` file is as follows:

sysid host address-string

The value of the address string is entered as a series of 2-digit hexadecimal values. Each byte in the string is represented by a pair of characters; for example, the decimal number 13 must be specified as `0d`. The precise format of this address is defined by the transport provider. For OSI connection oriented and TCP transport providers, the addresses are defined in *X/Open Portability Guide, Issue 3, Networking Services*.

The address variable is not required for dTET2 implementations that use transport interfaces other than XTI.

4.2.4.2 Example `systems` files

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

4.2.4.2.1 Example 1

This example is for a system on which dTET2 has been built to use the socket network interface.

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

000     ozone
001     neon
002     argon
003     ozone
```

4.2.4.2.2 Example 2

This example is for a system on which dTET2 has been built to use the XTI network interface.

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

000     ozone           000204010a01020000000000000000000
001     neon            000204010a01030000000000000000000
002     argon           000204010a01040000000000000000000
003     ozone           000204010a01020000000000000000000
```

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

5. Test case support

5.1 Introduction

dTET2 supplies functionality to support the development of test cases. This chapter describes that functionality for a Test Case Manager (TCM) or ‘‘wrapper’’ that provides a suitable environment for the execution of invocable components as requested by the MTCC. In addition, interfaces are described that that an application (test case) can use to:

- Control the execution of invocable components and test purposes.
- Output information and results to the execution results file.
- Cancel test purposes.
- Manipulate configuration variables.
- Generate and execute processes.
- Synchronise test purposes at agreed points in execution (distributed test cases only).
- Execute and control remote process execution (distributed test cases only).

These interfaces are referred to as the Application Programming Interface or API.

It is expected that several different language bindings will be made available with dTET2. This implementation currently provides bindings as follows:

- The C Language (ISO 9899) on ISO 9945–1 Systems.
- The Shell Language (*X/Open Portability Guide Issue 3 Volume 1* — `sh`).

These bindings are described later in this chapter.

5.2 The Test Case Manager

Through the TCMs on both master and slave systems, developers gain support in doing the following:

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

When TCMs process a non-distributed test case, each TCM fulfills the role of a master TCM and processes the test case without reference to any other TCMs that may be running at the same time. However, when TCMs process a distributed test case, the TCM running on the master system fulfills the role of a master TCM and the TCM running on the slave systems each fulfill the role of slave TCMs. The set of TCMs that process a distributed test case keep in step by synchronising with each other at certain pre-defined points during their execution. Refer to the chapter entitled ‘‘Test case synchronisation’’ in the dTET2 Installation and User Guide for details of synchronisation between TCMs.

The general flow of control for test cases written to the dTET2 API is as follows:

1. Each TCM arranges for access to its respective configuration information.
2. A master TCM outputs a Test Case Manager start message to the combined execution results file.
3. Each TCM builds a list of test purposes to be executed from the list of requested invocable components. If no invocable components were requested, or if the special invocable component `all` is requested, each TCM builds this list from all of the invocable components in the test case.
4. Each TCM arranges for the processing of asynchronous events.
5. When processing a distributed test case, all the TCMs synchronise with each other before executing their respective start-up procedures (if any).
6. Each TCM commences parallel execution of each invocable component in turn. A master TCM outputs an invocable component start entry to the combined execution results file as each invocable component is started.
7. All the TCMs synchronise at the commencement of each test purpose. During this synchronisation process the TCMs ensure that they are executing a common test purpose. A master TCM outputs a test purpose start message to the combined execution results file.

   During the test purpose any test case information lines from the TCMs are entered into the combined execution results file.
8. If an event occurs which interrupts the processing of a test purpose, the interrupted process immediately proceeds to the end of the test purpose and outputs a test result of UNRESOLVED. If the event is a termination message from the MTCC, the TCM reports receipt of the message to the execution results file, executes the specified clean-up procedure (if any) and exits; otherwise it continues by processing the next test purpose. When processing a distributed test purpose, an interrupted TCM informs the other TCMs of the event at the next pre-defined synchronisation point.
9. If a TCM is about to execute a test purpose that has been marked as cancelled, it instead reports the test purpose as UNINITIATED and continues to process the next test purpose (if any). If a test purpose has been marked as cancelled in one part of a distributed test case, the TCM informs all the others of the cancellation during the automatic synchronisation at the start of the test purpose.
10. All the TCMs synchronise at the end of a distributed test purpose. The individual results from each test purpose part are arbitrated and a final test result is output to the combined execution results file. At the end of a non-distributed test purpose, each TCM outputs a result to the journal file.
11. Once all of the test purposes in an invocable component have been executed, a master TCM outputs an Invocable Component End message to the journal file and each TCM moves on to process the next invocable component.
12. After all test purposes for all requested invocable components in a distributed test case have been executed, all the TCMs synchronise with each other.
13. After all test purposes for all requested invocable components have been executed, each TCM executes the specified clean-up procedure (if any) and exits normally.
14. If for any reason, a particular TCM fails to execute the requested set of invocable components or the specified start-up or clean-up procedure, the relevant API will exit

abnormally. The remaining components of the distributed test case will fail to synchronise at the next synchronisation point and will all terminate.

5.3 C language binding

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

Applications written to this language binding attach themselves to it through the following files:

- *tet-root*`/lib/dtet2/libapi.a` contains the support routines for test purposes.
- *tet-root*`/lib/dtet2/tcm.o` contains the routine `main()` and associated support routines for the sequencing and control of invocable components and test purposes.
- *tet-root*`/lib/dtet2/tcmchild.o` contains a `main()` routine which can be used by test suites when building processes which test purposes will launch using the `tet_exec()` interface.
- *tet-root*`/lib/dtet2/tcmrem.o` contains a `main()` routine which can be used by test suites when building processes which test purposes will launch using the `tet_remexec()` interface.
- *tet-root*`/inc/dtet2/tet_api.h` contains a definition of `struct tet_testlist`, values for use as arguments to `tet_result()` (i.e., `TET_PASS`, `TET_FAIL`, `TET_UNRESOLVED`, `TET_NOTINUSE`, `TET_UNSUPPORTED`, `TET_UNTESTED`, `TET_UNINITIATED` and `TET_NORESULT`) plus declarations and prototypes for all the ‘C’ API interfaces.

Each of these files should be accessed by test suites via their build tool in a way which is appropriate given the available ‘C’ language translation system. Test suite authors are advised to allow easy specification of alternate pathnames for these files (possibly through dTET2 configuration variables), thus improving the flexibility of their suites.

Note that test cases built to this API require the TCCs to execute. This is because the amount of effort required to establish an environment in which test cases could execute without the TCCs is substantial. This applies especially to the requirement for test purpose synchronisation and result arbitration.

If the communication variables normally set by the MTCC are unset when the test case is executed, `TET_ACTIVITY` defaults to 0, `TET_CODE` to `tet_code` and `TET_CONFIG` to none. If the file `tet_code` does not exist in the current directory, then the default set of result codes are used. If the test case requires configuration variables or additional result codes, those communication variables should be set accordingly.

5.3.1 Test case structure and management functions

These functions support functionality used in initialising and cleaning up test cases, and in selecting invocable components and test purposes (described in the chapter entitled ‘‘Writing a C language API-conforming test suite’’ elsewhere in this guide).

Synopsis

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

struct tet_testlist tet_testlist[];

void (*tet_startup)(void);

void (*tet_cleanup)(void);

int tet_thistest;

int tet_nosigreset;

char *tet_pname;
```

Description

The `tet_testlist[]` array declares, in the `testfunc` element of the structure, the pointer to the function that is associated with each test purpose and, in the `icref` element of the structure, the relationship of the test purpose to an invocable component. The `tet_testlist[]` array is terminated by a structure with the `testfunc` element set to `NULL`. No other element of the array will use the value `NULL` for this element.

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

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

The function pointers `tet_startup` and `tet_cleanup` are set to the functions to be used for test case specific start up and clean up procedures respectively. The start up procedure is executed before the first requested invocable component and the clean up procedure is executed on completion of the last requested invocable component. These routines are executed irrespective of which invocable components are requested. Note that if either of these pointers is set to `NULL`, the TCM will not attempt to call the respective function.

The TCM is provided as the `main()` routine to the test case program and contains an external declaration of the `tet_startup` and `tet_cleanup` function pointers and of the `tet_testlist[]` array.

The `tet_thistest` variable contains the sequence number (starting at 1) of the element in the `tet_testlist[]` array that is associated with the currently executing test purpose. During execution of the start up and clean up functions, `tet_thistest` is set to zero.

The `tet_nosigreset` variable controls whether the TCM reinstates signal handlers for unexpected signals before each test purpose. The default value of zero means that signal handlers will be reinstated before each test purpose, to ensure that unexpected signals do not go unnoticed if an earlier test purpose installed a local handler but did not restore the original handler. If `tet_nosigreset` is set to a non-zero value in the start-up function called via `(*tet_startup)()`, then signal handlers will be left in place between test purposes. In test cases where stray signals constitute a test failure, it is recommended that `tet_nosigreset` is left with its default value of zero. This is because, even if test purposes contain code to restore the signal handling, this code will not be executed if an unexpected signal arrives and the TCM skips to the start of the next test purpose.

The `tet_pname` variable contains the process name as given on the test case command line.

5.3.2 Insulating from the test environment

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

1TET_SIG_IGN defines (by comma separated number) the set of signals that are to be ignored during test purpose execution. Any signal that is not set to be ignored or to be left (see `TET_SIG_LEAVE` below) with its current disposition, will be caught when raised and the result of the test purpose will be set to `UNRESOLVED` because of the receipt of an unexpected signal. A test purpose may undertake its own signal handling as required for the execution of that test purpose. The disposition of signals will be reset after the test purpose has completed, unless the global variable `tet_nosigreset` is non-zero. The TCM needs to know how many signals the implementation supports in order to set up catching functions for these signals.

1TET_SIG_LEAVE defines (by number) the set of signals that are to be left unchanged during test execution. In most cases this will mean that the signal takes its default action. However, the user can change the disposition of the signal (to ignore) before executing the TCC if this signal is to remain ignored during the execution of the test purposes.

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

5.3.3 Making journal entries

These functions support functionality used in creating journal entries.

Synopsis

```
void tet_setcontext(void);

void tet_setblock(void);

void tet_infoline(char *data);

void tet_result(int result);
```

Description

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

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

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

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

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

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

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

5.3.4 Canceling test purposes

These functions support functionality used to cancel test purposes.

Synopsis

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

char *tet_reason(int testno);
```

Description

The function `tet_delete()` marks the test purpose specified by `testno` as canceled on the local system and will output `reason` as the reason for cancellation on the information line which is output whenever the TCM attempts to execute this test purpose. The argument `testno` is the sequence number (starting at 1) of the corresponding element in the `tet_testlist[]` array. If the requested `testno` does not exist, no action is taken. If the requested `testno` is already marked as canceled, the reason is changed to `reason` and the test purpose remains marked as canceled. If the `reason` is set to `(char *) NULL` then the requested `testno` is marked as active; this allows previously canceled test purposes to be re-activated.

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

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

The function `tet_reason()` returns a pointer to a string which contains the reason why the test purpose specified by `testno` has been canceled on the local system. If this test purpose does not exist or is not marked as canceled on the local system, a value of `(char *) NULL` is returned. It is not possible to use `tet_reason()` in a distributed test case to determine whether or not a remote test purpose part has been cancelled.

5.3.5 Manipulating configuration variables

The functions in this section support functionality used for manipulating configuration variables.

Synopsis

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

Description

The function `tet_getvar()` retrieves the setting of the configuration variable `name` and returns a pointer to that setting. This pointer will remain valid for the life of the process, regardless of subsequent calls to `tet_getvar()`.

Note that if a variable has no setting, `tet_getvar()` returns a pointer to an empty string. If a requested variable is undefined, `tet_getvar()` returns a `NULL` pointer.

5.3.6 Generating and executing processes

These functions support forking new child processes.

Synopsis

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

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

extern pid_t tet_child;
```

Description

The `tet_fork()` function creates a new process which is a copy of the calling process and modifies the signal disposition in the newly created process such that any signals that were being caught in the parent process are set to their default values in the child process. The child process will then commence execution of the routine `childproc()` and, upon completion of this routine, will terminate with an exit code that indicates to the parent the correctness of execution of this routine. If the routine `parentproc()` is not set to `NULL`, this routine will be executed by the parent process before the exit code of the child process is waited for.

On completion of the optional `parentproc()` routine, the exit code returned by the child process will be examined by masking off the bits which are set in `validresults`. If the result is zero, `tet_fork()` assumes that this was a legal (or expected) termination code. If not, it assumes that the child process completed with an unexpected result and an error has occurred. This unexpected result is reported to the execution results file. The `tet_fork()` function will return −1 if the result of the child process was invalid, or the valid result code if the result of the child process was one of the valid results. When `tet_fork()` returns −1 it reports the nature of the error using `tet_infoline()` and sets the test purpose result code to `UNRESOLVED` by calling `tet_result()`. If `waittime` is not set to zero, the parent process will ensure that the child process does not continue to execute for more than `waittime` seconds after the completion of the routine `parentproc()`.

The `tet_exec()` function may be called from a `childproc()` routine of a child process generated by a call to `tet_fork()`. The `tet_exec()` function will pass the argument data as specified by `argv[]` and the environment data specified by `envp` to the process specified by `file`. The usage of the `tet_exec()` is equivalent to that of the ISO 9945–1 `execve()` function, except that the API adds arguments and environment data that are to be interpreted by the driver of the executed `file`. Also, the new process should be built with `tcmchild.o` if that process is expected to make use of API calls. If `tet_exec()` is called without first calling `tet_fork()`, the results are undefined. This is because the `tet_fork()` function makes calls to `tet_setcontext()` in the child and `tet_setblock()` in the parent to distinguish output from the child and from the parent before, during and after execution of the `parentproc()` routine.

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

5.3.7 Executed process functions

These functions are to be used by processes executed through the `tet_exec()` and `tet_remexec()` functions.

Synopsis

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

void tet_exit(int status);

void tet_logoff(void);

int tet_thistest;

char *tet_pname;
```

Description

The function `tet_main()`, supplied by the test suite developer, is called by the `main()` function of the dTET2-supplied child process controllers `tcmchild.o` and `tcmrem.o`. Prior to calling `tet_main()`, `tcmchild.o` and `tcmrem.o` both set the `tet_thistest` variable to the number associated with the test purpose in the process that called `tet_exec()` or `tet_remexec()`. This value should not be changed by the executed process.

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

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

The function `tet_exit()` should be used instead of `exit()` by child processes that are started by calls to `tet_exec()` or `tet_remexec()`. This function logs off all dTET2 servers, then calls `exit()` with the specified `status` as argument. `tet_exit()` should only be called from the child process that is started by `tet_exec()` or `tet_remexec()` and not by any of its children.

The function `tet_logoff()` may be called by child processes that are started by calls to `tet_exec()` or `tet_remexec()`, which do not need to make any further dTET2 API calls and are not able to call `tet_exit()` at process termination time (e.g., if one of the flavours of `exec()` is about to be called in the child process). `tet_logoff()` should only be called once from the child process. The results are undefined if a process or any of its descendents makes any dTET2 API calls after `tet_logoff()` is called.

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

5.3.8 Test case synchronisation

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

Synopsis

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

Description

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

When `tet_sync()` is called from a process executing on a slave system, this indicates that the process wishes to synchronise with the master system as well as the other systems listed in the zero-terminated array of system IDs pointed to by `syncnames`. If `syncnames` is `NULL`, this is interpreted to mean an empty list of slave system IDs.

When `tet_sync()` is called from a process executing on the master system, this indicates that the process wishes to synchronise with the systems listed in the zero-terminated array of slave system IDs pointed to by `syncnames`. `syncnames` cannot be `NULL` or point to an empty array in this case since the master system must synchronise with at least one slave system.

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

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

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

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

- More than `waittime` seconds elapse between synchronisation requests from participating systems.
- A related synchronisation request times out on one of the other participating systems.
- The user-supplied function in a test case on one of the other participating systems returns control to its TCM before synchronising.

- The sync point specified by `syncptno` has already occurred.
- The calling process is running on the master system and `syncnames` is `NULL` or points to an empty system ID list.
- A system ID appears more than once in the array pointed to by `syncnames`.
- An invalid parameter is specified in the call.
- The API encounters a problem while processing the request.

When a call to `tet_sync()` is unsuccessful, the API places an entry in the journal file indicating the cause of the failure. If the call was unsuccessful because one or more of the participating systems failed to synchronise, or the related process timed out or terminated before the specified sync point occurred, this message identifies the systems that failed to synchronise successfully.

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

For an overview of dTET2 synchronisation and a description of how to interpret `tet_sync()` journal messages, see the chapter entitled ‘‘Test case synchronisation’’ in the dTET2 Installation and User Guide.

5.3.9 Remote system designations

These functions allow a dTET2 test purpose to retrieve information about system designations.

Synopsis

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

int tet_remgetsys (void);
```

Description

The `tet_remgetlist()` function returns the number of slave systems which are participating in a distributed test case. If there is at least one slave system available, a pointer to a zero-terminated array containing the names of the available slave systems is returned indirectly through `*sysnames`.

The `tet_remgetsys()` function returns the system ID of the system on which the calling process is executing.

5.3.10 Remote process control

These functions allow a part of a distributed test case running on one system to generate a remote process on another system.

Synopsis

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

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

int tet_remkill(int remoteid);
```

Description

The `tet_remexec()` function may be called from a distributed test case. The calling process will wait until the requested process has been started and has synchronised with it.

Upon successful synchronisation the call to `tet_remexec()` returns the `remoteid` of the remote executed process as a value greater than zero. If the call to `tet_remexec()` fails, a value of −1 is returned.

The `sysname` argument is the system ID of one of the other systems that is participating in the current distributed test case and the corresponding STCC is requested to initiate the process specified by `file`. The location of `file` is relative to the remote system's `TET_EXECUTE` directory if set, otherwise, it is relative to *tet-root* on the remote system. Since the request is performed by an STCC, it is not necessary for a process to call `tet_fork()` before calling `tet_remexec()`.

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

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

The call to `tet_remexec()` returns −1 and sets `errno` to EINVAL if `sysname` does not refer to a known remote system.

The call to `tet_remexec()` returns −1 and sets `errno` to ENOEXEC if `file` cannot be executed on the remote system, or if synchronisation with the remote process was not successful.

The call to `tet_remexec()` returns −1 and sets `errno` to EFAULT if the `file` or `argv` parameters are invalid.

The call to `tet_remexec()` returns −1 and sets `errno` to EIO if the connection with the remote system is broken.

The `tet_remwait()` function waits for the termination of a remote process initiated by `tet_remexec()`. The `remoteid` argument is the remote execution identifier returned from a successful call to `tet_remexec()`.

The call to `tet_remwait()` provides the exit status of the remote process in the integer pointed to by `statloc` and returns zero if the call has completed successfully. The exit status value returned indirectly through `*statloc` represents the value returned by the `wait()` system call on the remote system. Thus, when interpreting a signal number extracted from `*statloc` on the local system, the test suite author should bear in mind that it is possible for the meanings of signal numbers on the remote system to be different from meanings defined (e.g., in

`<signal.h>`) on the local system.

If the call to `tet_remwait()` fails to complete after `waittime` seconds or fails for any other reason a value of −1 is returned and `*statloc` is not updated. When `waittime` is set to zero, the call to `tet_remwait()` will return immediately, either with an error if the requested process has not yet terminated or with the exit status if the requested process has already terminated.

The call to `tet_remwait()` returns −1 and sets `errno` to EINVAL if `remoteid` does not refer to a process initiated from a call to `tet_remexec()`.

The call to `tet_remwait()` returns −1 and sets `errno` to ECHILD if `remoteid` refers to a process which is already the subject of a successful call to `tet_remwait()`.

The call to `tet_remwait()` returns −1 and sets `errno` to EAGAIN if `timeout` elapses.

The call to `tet_remwait()` returns −1 and sets `errno` to EINTR if the call is interrupted.

The call to `tet_remwait()` returns −1 and sets `errno` to EIO if the connection to the remote system is broken.

The `tet_remkill()` function causes the STCC which controls the remote process designated by `remoteid` to terminate the process. The `tet_remkill()` call returns immediately without awaiting confirmation that the process has terminated. (This information can be obtained from a subsequent call to `tet_remwait()` if required).

The call to `tet_remkill()` returns −1 and sets `errno` to EINVAL if `remoteid` does not refer to a process initiated from a call to `tet_remexec()`.

The call to `tet_remkill()` returns −1 and sets `errno` to EIO if the connection to the remote system is broken.

5.4 Shell language bindings

The synopses here are described in accordance with the command shell, `sh`, as defined in the *X/Open Portability Guide Issue 3 Volume 1*. See the chapter entitled ‘‘Writing a Shell language API-conforming test suite’’ elsewhere in this guide for an example of how to write a Shell language based test suite.

Support for this language binding is provided through Shell language source files as follows:

- *tet-root*`/lib/xpg3sh/tcm.sh` contains the support routines for the sequencing and control of invocable components and test purposes (the Shell TCM).
- *tet-root*`/lib/xpg3sh/tetapi.sh` contains the support routines for test purposes (the Shell API).

These files must be ‘‘sourced’’ into an executing Shell language test purpose using the `.` (dot) shell built-in command. Sourcing the Shell TCM also automatically sources the Shell API.

Note that test cases built to this API do not require the MTCC to execute. With careful setup of the communication variables a Shell test case can be invoked from the command line.

Distributed testing is not supported by the Shell API.

5.4.1 Test case structure and management functions

These functions support functionality used in initialising and cleaning up test cases, and in selecting invocable components and test purposes (described in the chapter entitled ‘‘Writing a Shell language API-conforming test suite’’ elsewhere in this guide).

Synopsis

`iclist="`*blank-separated list of invocable component names*`"`

`ic1="`*blank-separated list of test purpose names*`"`
`ic2="`*blank-separated list of test purpose names*`"`
. . .

`tet_startup=`*startup-procedure*

`tet_cleanup=`*cleanup-procedure*

`$tet_thistest`

Description

The start up routine, clean up routine and each of the test purposes should be implemented by the test author as either shell functions or as separate executable shell scripts. These shell functions or scripts will be called by the shell TCM according to the requested set of invocable components. The `iclist` definition is provided by the test suite author, and contains a blank separated list of invocable component names. These invocable component names are formed by prefixing each invocable component number with the letters `ic`.

When an invocable component is requested by the TCC, the shell TCM executes each name in the associated list of test purposes. Each of the test purposes will be executed in a sub-shell with the appropriate signal handling being applied to the sub-shell.

The TCM does not perform any explicit error checking on the contents of a list of test purposes. It is the responsibility of the test author to ensure that the names reference shell functions or executable shell scripts.

The shell variables `tet_startup` and `tet_cleanup` are set to refer to the shell function or script to be used for test case specific start up and clean up procedures, respectively. The start up procedure is executed before the first requested invocable component and the clean up procedure is executed on completion of the last requested invocable component. These routines are executed irrespective of which invocable components are requested. If no start up or clean up is required, the `tet_startup` and `tet_cleanup` variables may be left unset or set to an empty string.

The TCM and API are provided as shell scripts which must be sourced by the test suite author immediately after the `tet_startup`, `tet_cleanup` and `iclist` variables, each of the `ic`*n* variables, and any shell functions used by the test case have been defined. The shell script is sourced by use of the `.` (dot) shell built-in command. Note that if a test purpose is written as a separate shell script, that script must source the shell API in order to have access to API support routines.

The `tet_thistest` shell variable contains the **name** of the currently executing test purpose, as specified in the `ic`*n* variable.

5.4.2 Insulating from the test environment

The following configuration variables are used by the shell TCM to help determine which events should be handled for the test case, and which should be passed through.

1TET_SIG_IGN defines (by comma separated number) the set of signals that are to be ignored during test purpose execution. Any signal that is not set to be ignored or to be left (see `TET_SIG_LEAVE` below) with its current disposition, will be caught when raised and the result of the test purpose will be set to `UNRESOLVED` because of the receipt of an unexpected signal. A test purpose may undertake its own signal handling as required for the execution of that test purpose; the disposition of signals will be reset after the test purpose has completed. The API needs to know how many signals the implementation supports in order to set up trap statements for these signals.

1TET_SIG_LEAVE defines (by number) the set of signals that are to be left unchanged during test execution. In most cases this will mean that the signal takes its default action. However, the user can change the disposition of the signal (to ignore) before executing the TCC if this signal is to remain ignored during the execution of the test purposes.

The implementation does not allow a standard set of signals to be set to be ignored or left unchanged, as this may pervert test results.

5.4.3 Making journal entries

The functions in this section implement functionality used for creating journal entries.

Synopsis

```
tet_setcontext

tet_setblock

tet_infoline data ...

tet_result result
```

Description

The `tet_setcontext` shell function sets the current context to the value of the process ID of the calling shell. The current context is stored in the environment variable `TET_CONTEXT` which is marked for export. This allows the context to be passed to subsequent processes by using this environment variable. The `tet_setcontext` function should be executed by any application which executes a background subshell and which wishes to write entries from both processes. The `tet_setcontext` function must be executed from the child process, not from the parent. The parent should call `tet_setblock` as appropriate to distinguish its output before, during and after execution of the child.

The `tet_setblock` shell function increments the current block ID. The value of the current block ID is reset to one at the start of every test purpose and after a call to `tet_setcontext` which altered the current context. The sequence ID of the next entry is set to one at the start of each new block. The current block ID is stored in the shell variable `TET_BLOCK` which is marked for export.

The `tet_infoline` shell function outputs an information line to the execution results file. The sequence number is incremented by one after the line is output. If the current context and the current block ID have not been set, the call to `tet_infoline` causes the current context to be set using the value of the calling process ID and the current block ID to be set to one. Note that `tet_infoline` does not process backslash escapes like the shell `echo` built-in command. If more than one argument is passed to `tet_infoline`, the arguments will be output separated by a space character.

The `tet_result` shell function sets the result code that will be output at the end of the test purpose. The `result` argument specifies the **name** of the result that is to be output. This result is output to the execution results file by the TCM upon test purpose completion. This ensures that all informational messages are written out before the test purpose result, and that there is one (and only one) result generated per test purpose.

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

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

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

5.4.4 Canceling test purposes

These functions support functionality used for canceling test purposes.

Synopsis

```
tet_delete testname reason

tet_reason testname
```

Description

The shell function `tet_delete` marks the test purpose specified by `testname` as canceled and will output `reason` as the reason for cancellation on the information line which is output whenever the test driver attempts to execute this test purpose. The argument `testname` matches the name which is used to call this test purpose. If the requested `testname` does not match the name of a test purpose, no action is taken. If the requested `testname` is already marked as canceled the reason is changed to `reason` and the test purpose remains marked as canceled. If `reason` is an empty string then the requested `testname` is marked as active; this allows previously canceled test purposes to be re-activated.

The shell function `tet_reason` outputs, to the standard output device, the reason why the test purpose specified by `testname` has been canceled and exits with an exit code of 0. If this test purpose is not marked as canceled, or does not match the name of a test purpose, there is no output to the standard output device, and the function exits with an exit code of 1.

5.4.5 Manipulating configuration variables

There is no explicit shell interface to support this functionality. The API will ensure that the configuration information is available to the test purposes as shell variables marked `readonly`. Each of these shell variables can be accessed using the normal shell mechanisms.

5.4.6 Generation and execution of processes

There is no explicit shell interface to support this functionality. The API ensures that the configuration information and the `tet_thistest` shell variable are available as shell variables. The ability to use parentheses to generate a subshell environment allows these variables to be inherited when a subshell is generated. The only facilities that are not provided in the shell are the ability to timeout a subshell process and the examination of the exit code from the subshell. The shell provides facilities to accomplish these tasks in a relatively straightforward manner and this is considered to be an issue for the application programmer rather than for the API.

5.4.7 Executed process support

Shell scripts which are executed by a test case written to this API should source the Shell API to include the necessary support routines using the `.` (dot) shell built-in command. Note that this will not provide TCM functions (like signal handling and test purpose sequencing). Executed processes which need this type of support should be test cases in their own right.

6. Test reporting and journaling

6.1 Making journal entries

The dTET2 API journaling facility provides a mechanism for outputting data to the execution results file. The API ensures that each entry in this file is written atomically and that there is sequencing information applied to the entry (to allow the MTCC to reorder data that is produced from two or more concurrently executing processes started by a single test purpose). The MTCC ensures that simultaneous execution of test cases are isolated from one another.

In order to allow for the correct sequencing of information three attributes are defined:

- System identifier.
- The current context.
- The current block.

The current block is a subdivision of the current context and provides a means of ensuring contiguity, after resequencing, of a block of data that needed to traverse several entries. The need to traverse several entries may be caused by the limitations on the atomicity imposed by the implementation, or may be purely a matter of convenience for the test suite author.

The current context is initialised during test case start-up and can be changed only after a new process is generated. This allows the author to choose whether a number of concurrently executing test purposes should have the same context or different ones.

The system identification is used to distinguish entries written from test case parts on the multiple systems participating in distributed test cases.

The current block is initialised to one by the start-up routines at the commencement of each test purpose. The test author can increment the current block at any point during the output of entries in order to distinguish one block of data from another. Each individual entry within a block will be sequenced starting at one. Use of the journaling support facilities allows data from concurrently executing test purposes to be ordered correctly by the TCC.

6.1.1 Entries from the API

As mentioned above, the TCM handles the sequencing of test purposes as a part of executing invocable components. The sequencing mechanism outputs invocable component start and end information and test purpose start information to the execution results file. The test author is responsible for outputting test information and test results to the execution results file.

All of the data for an entry is transferred atomically to the execution results file. It is the responsibility of the test suite author to remain within the limitations imposed by the implementation for a single atomic write operation. dTET2 guarantees atomicity of writes up to 512 bytes.

If a test purpose executes another process that is built to the dTET2 API, and that executable is expected to generate journal messages, the test purpose must use the dTET2 API to communicate the current message context to the executed process.

6.1.2 Entries from test purposes

The API provides functionality for delivering informational messages and results from test purposes to the execution results file. These messages are in addition to those specified above, which are provided automatically by the TCMs. The content of informational messages is limited only by the limit imposed upon the total length of a journal line. It is expected that test cases will use this mechanism to deliver special messages to the journal or for additional reporting sequences that can be analysed by test suite specific report treatment filters. Test purposes also deliver results to the execution results file. These results are checked by the API to ensure that they have been defined by dTET2 or by the test suite. In the event of an invalid result, the TCM delivers a message to the execution results file and sets the result to `NORESULT`. The result actions are also checked by the API. If a test purpose specifies a result for which the action is `Abort`, then the TCM will immediately stop processing test purposes and exit.

Note that if a test purpose neglects to generate a result via the API, the TCM will supply a result of `NORESULT` for that test purpose.

6.2 Journal files

6.2.1 Description

Result files are written by test cases, build tools and clean tools when run with output capture mode disabled. These result files are then transferred into the journal file by the MTCC. The format of lines in these files is identical.

6.2.2 Journal line parameters

The total length of a journal line must not exceed 512 bytes. Each journal line is made up of a message type, the parameters for that message, and a message area (the format of which is unconstrained). Each message may have zero or more parameters associated with it. These parameters (strings or integers represented by no more than ten decimal digits), are blank separated and contained between vertical bars. Possible parameters include:

- The TCC activity number (*activity*).

  This number is incremented each time an activity performed by the MTCC. Each build, execute or clean-up of a test case is considered an individual activity.

- The invocable component number (*ICnumber*).
- An invocable component count (*ICcount*).

  This is the number of invocable components executed in each test case (expected or actual as specified).

- The test purpose number (*TPnumber*).

  This number uniquely identifies the test purpose within a test case.

- The test purpose count (*TPcount*).

  This is the number of test purposes that make up an invocable component.

- The message context (*context*).

  This field represents the process that generated the journal line. It consists of a three character system ID, followed by the process ID of the the process that initiated the entry.

- The message block number (*block*).

  This number is set to one at the start of each test purpose or new context, and is incremented each time the test purpose requests it. This number, along with the process identifier and message number (below) is used by the TCC to order the data in an execution result file prior to transferring that file into the journal.

- The message sequence number (*sequence*).

  This number is set to one at the start of each block, and is incremented each time a message is written to the result file.

- The current time (*time*).

  Times are given using the notation *HH*`:`*MM*`:`*SS* with a 24 hour clock.

- The current date (*date*).

  Dates are given using the notation *YYYYMMDD*; for example, 19910610 for 10th June 1991.

- A test case name (*testcase*).

  This is the test case name as given in the scenario file.

- A path name (*pathname*).

  The full path name of a file.

- The `tcc` execution mode (*mode*); possible values for this parameter are as follows:

  `0` Build
  `1` Execute
  `2` Clean-up

- A completion status (*status*).

  This is the value returned to the TCC by the operating system after execution of another process. Values less than zero are reserved for use by dTET2.

6.2.3 Journal line descriptions

In the list below each message type is listed, followed by a line indicating the number associated with that message and the required parameters, followed by a description of the message and its uses.

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.

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.

Master System Configuration Start

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

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

Slave 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.

Operator 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||`*text*

The scenario file `parallel` directive start marker.

Parallel End

`620||`*text*

The scenario file `parallel` directive end marker.

Repeat Start

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

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

Repeat End

`720||`*text*

The scenario file `repeat` 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 remote system IDs specified by the scenario file `remote` directive.

Remote End

`820||`*text*

The scenario file `remote` directive end marker.

Test Case Controller End

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

When `tcc` completes sequencing of the scenario, it generates a TCC end message. This message indicates the time execution completed.

7. Writing a C language API-conforming test suite

7.1 Introduction

This chapter describes a sample test suite that conforms to dTET2's C language binding of the API. The source code for the test suite can be found in the appendix entitled ‘‘C language API test suite source files’’ at the end of this guide.

This sample test suite is designed to illustrate how a non-distributed test suite can be structured under dTET2, as well as how individual test cases and their test purposes relate to each other and to the API. The test suite has been deliberately kept simple and realistic. For example, one test purpose compares the returned error code against an expected error code of a failed system call, while another test purpose in the same test case checks the successful execution of the system call.

Small segments of code from the test suite appear in the following sections to help illustrate specific points. Refer to the appropriate section in the appendix entitled ‘‘C language API test suite source files’’ at the end of this guide to see the code in its entirety.

7.2 Defining a test suite

Test suites reside in subdirectories of *tet-root*. As explained in the chapter entitled ‘‘Testing structure’’ earlier in this guide, the name of the subdirectory and the test suite are the same.

The following figure shows the component files of the sample test suite, called `C-API`:

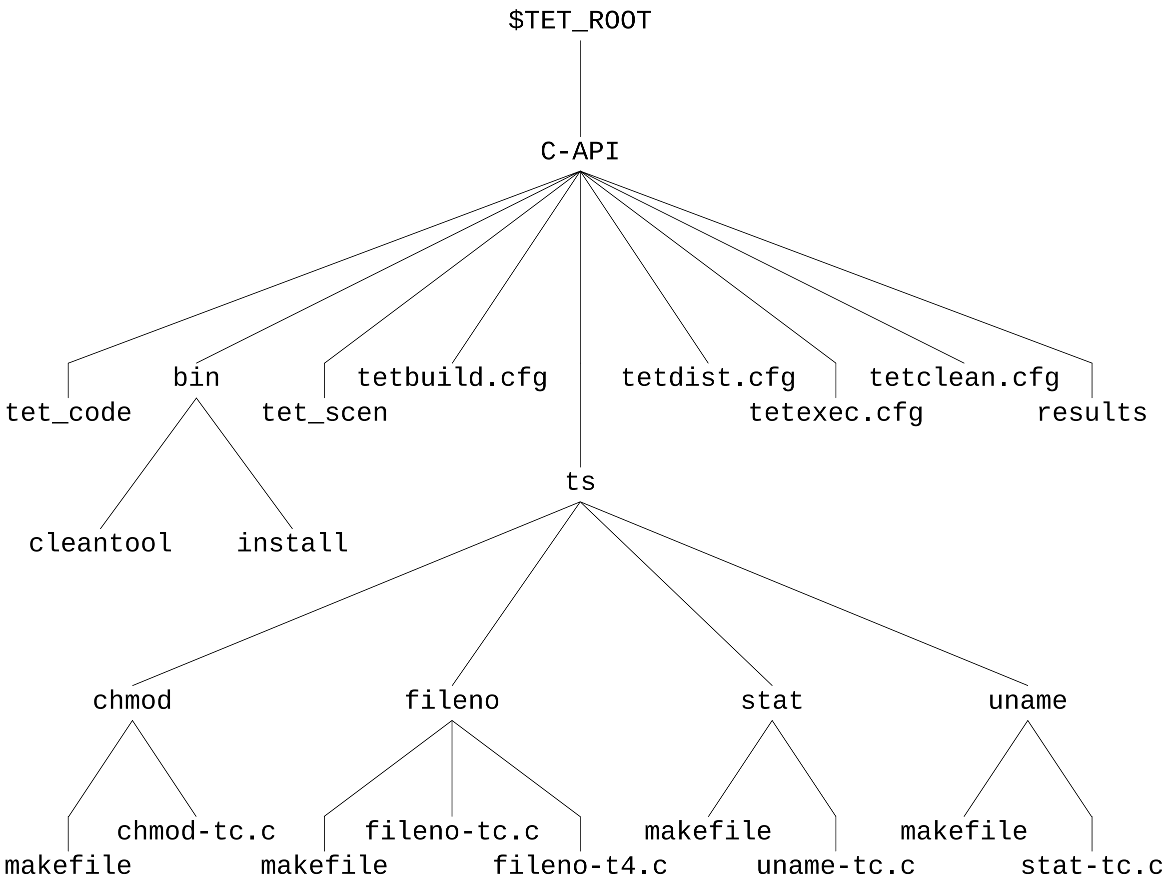

Figure 4. Directory structure for the example ‘C’ language test suite

The make-up of this test suite is similar to the demonstration test suite as defined for the master system and contains the following files:

- An install script and clean tool in the `bin` directory.
- Configuration files for test build, execution, and cleanup.
- A control file, `tet_scen`.
- A result codes file, `tet_code`.
- Several test cases in a directory structure under the directory `ts`.
- A results directory.

In this test suite, the `tetdist.cfg` file is only required for versions of dTET2 that use the XTI network interface. In that case it should be used to set values for the `TET_XTI_TPI`, `TET_XTI_MODE` and `TET_LOCALHOST` configuration variables. The `systems` file is also not required for this test suite because there are no slave systems defined in the test configuration.

The control file, `tet_scen`, lists the components of the test suite; and its contents determine the scenarios that can be used in running the test suite. The control file, `tet_scen`, for the `C-API` test suite contains the following lines:

```
#       chmod, fileno, stat, uname test suite.

all
        "Starting Full Test Suite"
        /ts/chmod/chmod-tc
        /ts/fileno/fileno-tc
        /ts/stat/stat-tc
        /ts/uname/uname-tc
        "Completed Full Test Suite"

chmod
        "Starting chmod Test Case"
        /ts/chmod/chmod-tc
        "Finished chmod Test Case"

fileno
        "Starting fileno Test Case"
        /ts/fileno/fileno-tc
        "Finished fileno Test Case"

stat
        "Starting stat Test Case"
        /ts/stat/stat-tc
        "Finished stat Test Case"

uname
        "Starting uname Test Case"
        /ts/uname/uname-tc
        "Finished uname Test Case"

# EOF
```

The control file lists five scenarios for the test suite: `all` (required), `chmod`, `fileno`, `stat` and `uname`. Since the test suite is composed of four test cases, one for the `chmod()` system call, one for the `fileno()` system call, one for the `stat()` system call, and one for the `uname()` system call, the control file has been written to allow each test case to be handled as a separate scenario, or for the whole test suite to be run at once with the `all` scenario.

The lines enclosed in double quotes (") are optional information lines that get passed into the journal file. The lines that begin with a slash or stroke character (/) name the executable test cases associated with each scenario. Note that, even though these lines begin with a slash character, the location of the test cases is interpreted relative to the local directory (the root directory for the test suite). In this instance, the test cases are in a subdirectory named `ts`.

The clean tool is used to remove unwanted files after the build of each test case. It is invoked in the source directory of the test case. In this case it is set to `exec make clean` to remove unwanted object files as defined in each `makefile`.

7.3 Defining common test case functions and variables

Since most test suites lend themselves to lots of code redundancy, making an effort to group together common functions and variables can greatly simplify the writing and debugging of a test suite. With the `C-API` test suite (which is very small), no common functions and variables other than the standard ones in `tetapi.h` were created.

One additional result code was invented, however, which would normally be defined in a test suite specific header file. But because it is only used within one test case in this very small test suite, it is instead defined within `uname-tc.c` as follows:

```
#undef TET_INSPECT      /* must undefine because TET_ is reserved prefix */
#define TET_INSPECT 33  /* this would normally be in a test suite header */
```

7.4 Initialising test cases

Every test case requires some minimum initialisation of functions and variables. The `fileno-tc` test case provides a good illustration of how this initialisation can be handled.

```
/* fileno-tc.c : test case for fileno() interface */

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

#include <tet_api.h>

extern char **environ;

static void cleanup();
static void tp1(), tp2(), tp3(), tp4(), ch4();

/* Initialise TCM data structures */
void (*tet_startup)() = NULL;
void (*tet_cleanup)() = cleanup;
struct tet_testlist tet_testlist[] = {
    { tp1, 1 },
    { tp2, 2 },
    { tp3, 3 },
    { tp4, 4 },
    { NULL, 0 }
};

/* Test Case Wide Declarations */
static char msg[256];                  /* buffer for info lines */
```

After the `#include` statements, several functions are declared. As described in the chapter entitled ‘‘Test case support’’ earlier in this guide, dTET2 provides the option of naming both a startup and a cleanup function. The named startup function will be called before the first test purpose is executed; and the cleanup function will be called after all test purposes have been executed. In this test case, only the cleanup function is named. The cleanup function `cleanup()` removes files created during the course of the test case.

The `stat-tc` test case includes a more substantial cleanup function, as well as a startup function. It requires that a file be created before the first test purpose, so this is handled by the startup function; this same file, as well as another file and a directory created during the tests, is then removed in the cleanup function. See the appendix entitled ‘‘C language API test suite source files’’ at the end of this guide for a complete code listing of the `stat-tc` test case.

The `fileno-tc` test case includes four test purposes, contained in the functions `tp1()`, `tp2()`, `tp3()` and `tp4()`. First the functions are declared (including an extra function which is a child process started by `tp4()`), as shown above. Then they are listed in the `tet_testlist` array with the invocable component to which they belong. In this case, each test purpose can be executed individually, so they are assigned to separate invocable components. If, say, `tp2()` depended on prior execution of `tp1()`, then they would be assigned the same IC number. After the array is set, any test case wide declarations are made. This commonly includes a buffer to use for constructing information lines to be output with `tet_infoline()`.

7.5 Controlling and recording test case execution results

Identifying and executing highly specific tests is central to any test case. Each test purpose in a test case typically targets one specific test that is loosely or strongly related to the other test purposes contained in the test case. The central purpose of each of these test purposes is to relay information about the execution of the test for the tester to examine later. This relaying of information can take the form of informational messages describing the test being executed, fatal or non-fatal errors that were encountered, and specific test execution results, such as PASS or FAIL.

The `chmod-tc` test case contains test purposes as follows:

1tp1 A successful `chmod` of a file, expecting a return code of 0.

1tp2 A failed `chmod` of a non-existent file, expecting a return code of −1 and `errno` set to `ENOENT`.

1tp3 A failed `chmod` of a file that contains a non-directory path component, expecting a return code of −1 and `errno` set to `ENOTDIR`.

Functions `tp1()` and `tp2()` are shown here and are described below.

```
static void
tp1()           /* successful chmod of file: return 0 */
{
    int ret, err;
    mode_t mode;

    tet_infoline("SUCCESSFUL CHMOD OF FILE");

    /* change mode of file created in startup function */

    errno = 0;
    if ((ret=chmod(tfile, (mode_t)0)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "chmod(\"%s\", 0) returned %d, expected 0",
```

```
                tfile, ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
            return;
        }

        /* check mode was changed correctly */

        if (stat(tfile, &buf) == -1)
        {
            (void) sprintf(msg,
                "stat(\"%s\", buf) failed - errno %d", tfile, errno);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }

        mode = buf.st_mode & O_ACCMODE;
        if (mode != 0)
        {
            (void) sprintf(msg, "chmod(\"%s\", 0) set mode to 0%lo, expected 0",
                tfile, (long)mode);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }

    static void
    tp2()       /* chmod of non-existent file: return -1, errno ENOENT */
    {
        int ret, err;

        tet_infoline("CHMOD OF NON-EXISTENT FILE");

        /* ensure file does not exist */

        if (stat("chmod.2", &buf) != -1 && unlink("chmod.2") == -1)
        {
            tet_infoline("could not unlink chmod.2");
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* check return value and errno set by call */

        errno = 0;
        ret = chmod("chmod.2", (mode_t)0);
```

```
        if (ret != -1 || errno != ENOENT)
        {
            err = errno;
            if (ret != -1)
            {
                (void) sprintf(msg,
                    "chmod(\"chmod.2\", 0) returned %d, expected -1", ret);
                tet_infoline(msg);
            }

            if (err != ENOENT)
            {
                (void) sprintf(msg,
                    "chmod(\"chmod.2\", 0) set errno to %d, expected %d (ENOENT)",
                    err, ENOENT);
                tet_infoline(msg);
            }

            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }
```

The comments for the code should clarify what is happening on each line. However, it is important to note that a lot of useful diagnostics have been written right into the tests. If any of the system calls fail, whether it is the one being specifically tested or one that the test relies on, that failure will be reported. Also, the tests begin the same, with a message about the test's purpose; and they end the same, with a pass/fail result being reported.

This sort of consistency yields two important benefits:

- Test purposes will be easier to write when they follow some sort of template.
- Test purposes will be easier to debug and evaluate when diagnostic information is built in from the very start.

7.6 Results that must be verified by the user

Some test cases may require user verification of information generated by a test case. An example of this can be found in the `uname-tc` test case when system specific information is being reported.

```
    static void
    tp1()         /* successful uname: return 0 */
    {
        int ret, err;
        struct utsname name;

        tet_infoline("UNAME OUTPUT FOR MANUAL CHECK");

        /* The test cannot determine automatically whether the information
           returned by uname() is correct.  It therefore outputs the
           information with an INSPECT result code for checking manually. */
```

```
        errno = 0;
        if ((ret=uname(&name)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "uname() returned %d, expected 0", ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else
        {
            (void) sprintf(msg, "System Name:  \"%s\"", name.sysname);
            tet_infoline(msg);
            (void) sprintf(msg, "Node Name:    \"%s\"", name.nodename);
            tet_infoline(msg);
            (void) sprintf(msg, "Release:      \"%s\"", name.release);
            tet_infoline(msg);
            (void) sprintf(msg, "Version:      \"%s\"", name.version);
            tet_infoline(msg);
            (void) sprintf(msg, "Machine Type: \"%s\"", name.machine);
            tet_infoline(msg);

            tet_result(TET_INSPECT);
        }
    }
```

Since the information from `uname()` will be different on every machine, the output needs to be reported and then verified. Here the information is simply being printed out for the tester to see and check, but no attempt has been made to interact with the tester to receive verification of the information and then use that verification to set the pass/fail result. Instead, a result code of INSPECT has been used.

7.7 Child processes and subprograms

Some test purposes require the creation of a child process or execution of a subprogram. dTET2 provides several interfaces to facilitate this, as follows:

1tet_fork() an API function called by test purposes to create a child process and perform processing in parent and child concurrently.

1tet_exec()
: an API function called by child processes to execute subprograms.

1tet_main()
: a user-supplied function to be defined in subprograms executed by `tet_exec()`.

An example of their use can be found in test purpose `tp4` of the `fileno` test case:

```
static void
tp4()         /* on entry to main(), stream position of stdin, stdout and
                 stderr is same as fileno(stream) */
{
    tet_infoline("ON ENTRY TO MAIN, STREAM POSITION OF STDIN, \
        STDOUT AND STDERR");

    /* fork and execute subprogram, so that unique file positions can be
       set up on entry to main() in subprogram */

    (void) tet_fork(ch4, TET_NULLFP, 30, 0);
}

static void
ch4()
{
    int fd, ret;
    static char *args[] = { "./fileno-t4", NULL };

    /* set up file positions to be inherited by stdin/stdout/stderr
       in subprogram */

    for (fd = 0; fd < 3; fd++)
    {
        (void) close(fd);
        if ((ret=open("fileno.4", O_RDWR|O_CREAT, S_IRWXU)) != fd)
        {
            (void) sprintf(msg, "open() returned %d, expected %d", ret, fd);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }
        if (lseek(fd, (off_t)(123 + 45*fd), SEEK_SET) == -1)
        {
            (void) sprintf(msg, "lseek() failed - errno %d", errno);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }
    }

    /* execute subprogram to carry out remainder of test */

    (void) tet_exec(args[0], args, environ);

    (void) sprintf(msg, "tet_exec(\"%s\", args, env) failed - errno %d",
        args[0], errno);
    tet_infoline(msg);
    tet_result(TET_UNRESOLVED);
}
```

All the testing is done in the child, so the function `tp4()` simply calls `tet_fork()` and ignores the return value. If it needed to do any processing after the call to `tet_fork()`, it should check that the return value was one of the expected child exit codes before continuing.

The arguments to `tet_fork()` are as follows:

- A function to be executed in the child.
- A function to be executed in the parent. In this case no parent processing is required, so the null function pointer `TET_NULLFP` (defined in `tet_api.h`) is used.
- A timeout period in seconds.
- A bitwise OR of the valid child exit codes. In this case the only valid exit code is zero.

The file `fileno-t4.c` contains the definition of `tet_main()`, as follows:

```
int
tet_main(argc, argv)
int argc;
char **argv;
{
    long ret, pos;
    int fd, err, fail = 0;
    static FILE *streams[] =  {  stdin,   stdout,   stderr };
    static char *strnames[] = { "stdin", "stdout", "stderr" };

    /* check file positions of streams are same as set up in parent */

    for (fd = 0; fd < 3; fd++)
    {
        pos = 123 + 45*fd; /* must match lseek() in parent */
        errno = 0;
        if ((ret = ftell(streams[fd])) != pos)
        {
            err = errno;
            (void) sprintf(msg, "ftell(%s) returned %ld, expected %ld",
                strnames[fd], ret, pos);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            fail = 1;
        }
    }

    if (fail == 0)
        tet_result(TET_PASS);
    else
        tet_result(TET_FAIL);

    return 0;
}
```

7.8 Cleaning up test cases

Since test cases often change and/or create data, it is important to cleanup this data before exiting the test case. As explained earlier, one way to do this is to specify a cleanup function with dTET2's `tet_cleanup` utility. The cleanup function named in the `stat-tc` test case provides a good example.

```
static void
cleanup()
{
    /* remove file created by start-up */
    (void) unlink(tfile);

    /* remove files created by test purposes, in case they don't run
       to completion */
    (void) rmdir("stat.d");
    (void) unlink("stat.p");
}
```

The `cleanup` function is called when all the test purposes have finished executing. As shown, it simply removes the files and directory that were created during the test.

8. Writing a Shell language API-conforming test suite

8.1 Introduction

This chapter describes a sample test suite that conforms to dTET2's shell language binding of the API. The source code for the test suite can be found in the appendix entitled ‘‘Shell API test suite source files’’ at the end of this guide.

This sample test suite, like the one in the chapter entitled ‘‘Writing a C language API-conforming test suite’’, is designed to illustrate how a test suite can be structured under dTET2, as well as how individual test cases, and their test purposes, relate to each other and to the API. Like the `C-API` test suite, this test suite has been deliberately kept simple and realistic. However, instead of system calls being tested, the equivalent user-level commands are tested. Sample tests include checking a returned error code and error message against an expected error code and expected error message and printing out system specific information for verification by the tester.

Note that no support for distributed test cases is presently provided by the shell API. It is possible to execute test cases on a local system or on one or more remote systems, but no synchronisation between test parts on multiple systems is possible. The `systems` file is only required if you wish to execute test cases on remote systems. `tetdist.cfg` is also required for remote execution.

Small segments of code from the test suite appear in the following sections to help illustrate specific points. Refer to the appropriate section in the appendix entitled ‘‘Shell API test suite source files’’ at the end of this guide to see the code in its entirety.

8.2 Defining a test suite

As explained in the chapter entitled ‘‘Writing a C language API-conforming test suite’’, test suites reside in subdirectories of *tet-root*. The name of the subdirectory and the test suite are the same.

The following figure shows the component files of the sample test suite, called `SHELL-API`:

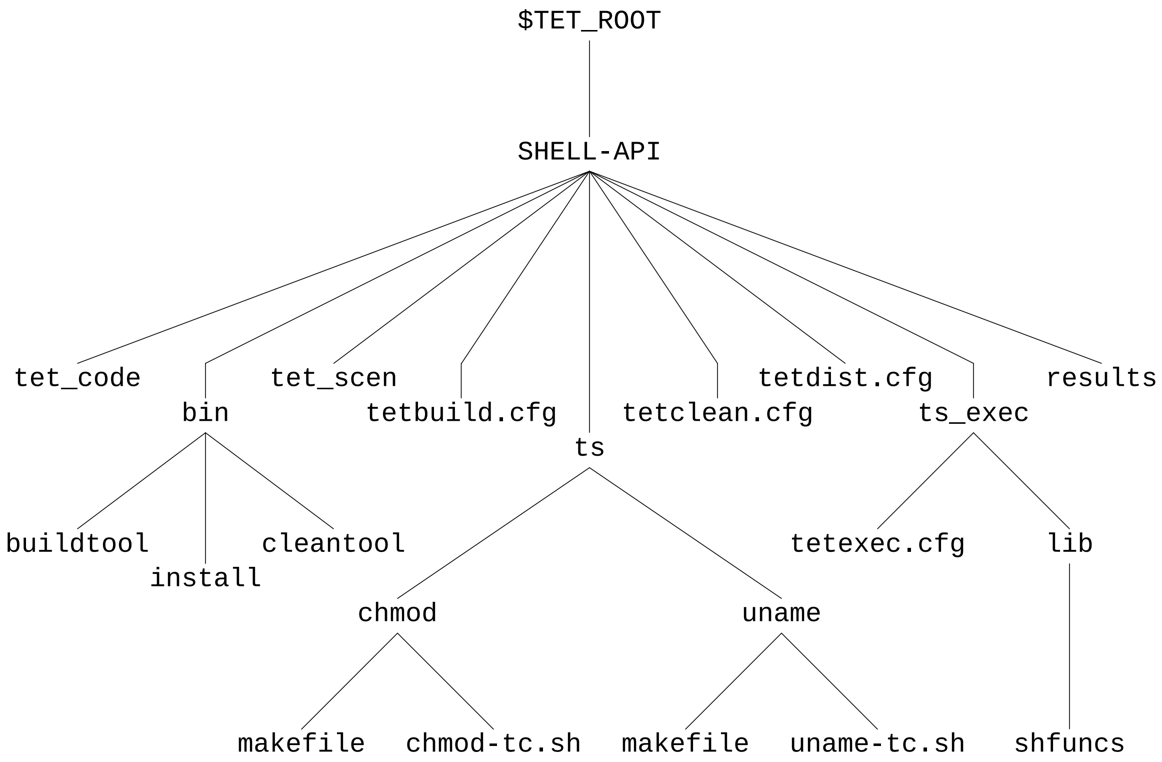

Figure 5. Directory structure for the example Shell language test suite

The make-up of this test suite is similar to the `C-API` test suite and contains the following files:

- An install script, build tool and clean tool in the `bin` directory.
- Some configuration files for test build, execution, and cleanup
- A control file, `tet_scen`.
- A result codes file, `tet_code`.
- Several test cases in a directory structure under the directory `ts`.
- An alternate execution directory `ts_exec`.
- A results directory.

The control file, `tet_scen`, is similar to the control file for the `C-API` test suite. See the chapter entitled ‘‘Writing a C language API-conforming test suite’’ for a description of the control file and how its structure relates to the scenarios that can be run.

The installation utility `install` creates the directory structure under the alternate execution directory to match the structure under the `ts` directory. For the purpose of this example the location of the alternate execution directory is fixed as `$TET_ROOT/SHELL-API/ts_exec` but in general it would be obtained from the user and could be located anywhere.

The build tool is used to build each test case. It is invoked in the source directory of the test case and installs the relevant files under the alternate execution directory. It does this by calling `make`, setting `TET_EXECUTE` to the correct value on the `make` command line in order to override the default value in each `makefile`.

The clean tool is used to remove the installed files from under the alternate execution directory. It works in the same way as the build tool except it executes a `make clean` instead of just `make`.

8.3 Defining common test case functions and variables

Just as with the `C-API` test suite, it makes good sense to minimise code redundancy by grouping together common functions and variables. In the process of writing the `SHELL-API` test suite, several common functions were created. This code was collected into a file named `shfuncs`, which is in the `lib` subdirectory of the `ts_exec` directory. The `shfuncs` file is sourced into each of the two test cases using the shell built-in `.` (dot) command. The required dTET2 Shell API file, `tcm.sh` (which in turn sources in the other required dTET2 Shell API file `tetapi.sh`) is also sourced into each of the two test cases. It is important to note the point at which these files are sourced in. Since the dTET2 API files read definitions and begin execution when they are sourced in, they must be sourced in as the very last part of each test case. Therefore, the last line of each of the test cases sources in `tcm.sh`.

`shfuncs` contains the following functions:

```
# shfuncs : test suite common shell functions

tpstart() # write test purpose banner and initialise variables
{
    tet_infoline "$*"
    FAIL=N
}

tpresult() # give test purpose result
{
    # $1 is result code to give if FAIL=N (default PASS)
    if [ $FAIL = N ]
    then
        tet_result ${1-PASS}
    else
        tet_result FAIL
    fi
}

check_exit() # execute command (saving output) and check exit code
{
    # $1 is command, $2 is expected exit code (0 or "N" for non-zero)
    eval "$1" > out.stdout 2> out.stderr
    CODE=$?
    if [ $2 = 0 -a $CODE -ne 0 ]
    then
        tet_infoline "Command ($1) gave exit code $CODE, expected 0"
        FAIL=Y
    elif [ $2 != 0 -a $CODE -eq 0 ]
    then
        tet_infoline "Command ($1) gave exit code $CODE, expected non-zero"
```

```
            FAIL=Y
        fi
    }

    check_nostdout() # check that nothing went to stdout
    {
        if [ -s out.stdout ]
        then
            tet_infoline "Unexpected output written to stdout, as shown below:"
            infofile out.stdout stdout:
            FAIL=Y
        fi
    }

    check_nostderr() # check that nothing went to stderr
    {
        if [ -s out.stderr ]
        then
            tet_infoline "Unexpected output written to stderr, as shown below:"
            infofile out.stderr stderr:
            FAIL=Y
        fi
    }

    check_stderr() # check that stderr matches expected error
    {
        # $1 is file containing regexp for expected error
        # if no argument supplied, just check out.stderr is not empty

        case $1 in
        "")
            if [ ! -s out.stderr ]
            then
                tet_infoline "Expected output to stderr, but none written"
                FAIL=Y
            fi
            ;;
        *)
            expfile="$1"
            OK=Y
            exec 4<&0 0< "$expfile" 3< out.stderr
            while read expline
            do
                if read line <&3
                then
                    if expr "$line" : "$expline" > /dev/null
                    then
                        :
                    else
                        OK=N
                        break
                    fi
                else
                    OK=N
                fi
```

```
            done
            exec 0<&4 3<&- 4<&-
            if [ $OK = N ]
            then
                tet_infoline "Incorrect output written to stderr, as shown below"
                infofile "$expfile" "expected stderr:"
                infofile out.stderr "received stderr:"
                FAIL=Y
            fi
            ;;
        esac
    }

    infofile() # write file to journal using tet_infoline
    {
        # $1 is file name, $2 is prefix for tet_infoline

        prefix=$2
        while read line
        do
            tet_infoline "$prefix$line"
        done < $1
    }
```

Since these functions perform commonly required tasks, they are better defined once rather than twice. Also, should they ever need to be changed, this means changing only one file.

Executing the test for each test purpose in some common, controlled way can make writing the tests and checking their results much easier. The function `check_exit` was written to:

— execute a command in a given argument (`$1`);

— capture both standard error and standard output in separate files (in case one or both need to be checked);

— record the exit code in a variable called `$CODE`;

— output a message to the journal if an unexpected exit code is found.

This function is coded as follows:

```
    check_exit() # execute command (saving output) and check exit code
    {
        # $1 is command, $2 is expected exit code (0 or "N" for non-zero)
        eval "$1" > out.stdout 2> out.stderr
        CODE=$?
        if [ $2 = 0 -a $CODE -ne 0 ]
        then
            tet_infoline "Command ($1) gave exit code $CODE, expected 0"
            FAIL=Y
        elif [ $2 != 0 -a $CODE -eq 0 ]
        then
            tet_infoline "Command ($1) gave exit code $CODE, expected non-zero"
            FAIL=Y
        fi
    }
```

8.4 Initialising test cases

Every test case requires some minimum initialisation of functions and variables. The `uname-tc` test case provides a good illustration of how this initialisation can be handled.

```
:
# uname-tc.sh : test case for uname command

tet_startup=""                          # no startup function
tet_cleanup="cleanup"                   # cleanup function
iclist="ic1 ic2"                        # list invocable components
ic1="tp1"                               # functions for ic1
ic2="tp2"                               # functions for ic2
```

As described in the chapter entitled ''Test case support'' elsewhere in this guide, dTET2 provides the option of naming both a startup and cleanup function. The named startup function will be called before the first test purpose is executed; and the cleanup function will be called after all test purposes have been executed. Here, only a cleanup function is named, by setting `tet_cleanup` equal to the name of the function that will be used.

A `cleanup` function is used by both of the test cases.

```
cleanup() # clean-up function
{
     rm -f out.stdout out.stderr out.experr
}
```

It simply removes the files containing the actual standard output, actual standard error and expected standard error for the test.

The `iclist` variable must contain a space-separated list of the invocable components contained in the test case. This list must be in the form shown above, meaning: `ic1`, `ic2`, and so on. No other names can be used. The next lines define the correspondence between invocable components (`ic`*n*) and the test purpose(s) that they contain. In this test case, each test purpose can be executed individually, so they are assigned to separate invocable components. If, say, `tp2` depended on the prior execution of `tp1`, then the definitions would be:

```
iclist=ic1                              # list invocable components
ic1="tp1 tp2"                           # functions for ic1
```

8.5 Controlling and recording test case execution results

As shown above, a lot of effort has been taken to report on the processing of each test case, and even on the individual test purposes. The `chmod-tc` test case, presented below, shows how information about the processing of a test case can be handled.

The `chmod-tc` test case contains test purposes as follows:

1tp1 successful `chmod` of a file with an expected exit code of 0.

1tp2 failed `chmod` of a non-existent file with an expected exit code of non-zero

1tp3 failed `chmod` due to invalid syntax with an expected exit code of non-zero.

Function `tp1` is shown here and is described below.

```
tp1() # simple chmod of file - successful: exit 0
{
    tpstart "SIMPLE CHMOD OF FILE: EXIT 0"

    echo x > chmod.1 2> out.stderr      # create file
    if [ ! -f chmod.1 ]
    then
        tet_infoline "Could not create test file: chmod.1"
        tet_infoline `cat out.stderr`
        tet_result UNRESOLVED
        return
    fi

    check_exit "chmod 777 chmod.1" 0    # check exit value

    MODE=`ls -l chmod.1 |cut -d" " -f1` # get and check mode of file
    if [ X"$MODE" != X"-rwxrwxrwx" ]
    then
        tet_infoline "chmod 777 set mode to $MODE, expected -rwxrwxrwx"
        FAIL=Y
    fi

    check_nostdout                      # should be no stdout
    check_nostderr                      # should be no stderr

    tpresult                            # set result code
}
```

The comments for the code should clarify what is happening on each line. Like the `C-API` test cases, this test purpose begins with reporting information about what the test has been designed to check and ends with setting the test result. In this case, this is done by the function `tpresult`, where the test status variable `$FAIL` is tested and reported on. If the file that is needed for the test cannot be created, the test outputs diagnostics to the journal and returns a result of `UNRESOLVED`. Note that in addition to checking the exit code, the file itself is checked to make sure that the mode set by `chmod` was actually set. Also, since a successful execution of this command means that nothing is written to standard error or standard output, functions contained in `shfuncs` are used to make sure that no data was output by the command.

In the test purpose, `tp2`, much of the same function calls are used, even though in this test `chmod` is expected to fail.

```
tp2() # chmod of non-existent file : exit non-zero
{
    tpstart "CHMOD OF NON-EXISTENT FILE: EXIT NON-ZERO"

    # ensure test file does not exist
    rm -f chmod.2 2> out.stderr
    if [ -f chmod.2 ]
    then
        tet_infoline "Could not remove test file: chmod.2"
        tet_infoline `cat out.stderr`
        tet_result UNRESOLVED
        return
    fi
```

```
        check_exit "chmod 777 chmod.2" N    # check exit value

        check_nostdout                      # should be no stdout
        check_stderr                        # check error message

        tpresult                            # set result code
    }
```

Again, the description of the test is reported, the exit code is checked against that expected by `check_exit`, `tp_result` is called to report the test result, and `check_nostdout` is used to make sure no data was sent to standard output. In this case, however, an error message should be produced; so the expected message is captured in a file in order that it can be later compared with the error message received. This is done through the function `check_stderr` which is defined in `shfuncs` and is shown here.

```
    check_stderr() # check that stderr matches expected error
    {
        # $1 is file containing regexp for expected error
        # if no argument supplied, just check out.stderr is not empty

        case $1 in
        "")
            if [ ! -s out.stderr ]
            then
                tet_infoline "Expected output to stderr, but none written"
                FAIL=Y
            fi
            ;;
        *)
            expfile="$1"
            OK=Y
            exec 4<&0 0< "$expfile" 3< out.stderr
            while read expline
            do
                if read line <&3
                then
                    if expr "$line" : "$expline" > /dev/null
                    then
                        :
                    else
                        OK=N
                        break
                    fi
                else
                    OK=N
                fi
            done
            exec 0<&4 3<&- 4<&-
            if [ $OK = N ]
            then
                tet_infoline "Incorrect output written to stderr, as shown below"
                infofile "$expfile" "expected stderr:"
                infofile out.stderr "received stderr:"
                FAIL=Y
            fi
```

```
            ;;
        esac
    }
```

The two files, one containing the expected error output in regular expression form, and the other containing the received error output, are compared line-by-line and, if they are identical, nothing is done. However, if they differ, it is important to know how they differ; therefore, both files are printed for the tester to evaluate later and the status of the test purpose is set to show a failed result.

As shown here, a lot of useful diagnostics have been written right into the tests. If any of the commands fail, whether it is the one being specifically tested or one that the test relies on, that failure will be reported. Also, each test case and test purpose begins with information reported in a consistent format; and they end the same, with a pass/fail (or other) result being reported.

As with the `C-API` test suite, this sort of consistency yields two important benefits:

- Test purposes will be easier to write when they follow some sort of template.
- Test purposes will be easier to debug and evaluate when diagnostic information is built in from the very start.

8.6 Results that must be verified by the user

Some test cases may require user verification of information generated by a test case. An example of this can be found in the `uname-tc` test case when system specific information is being reported.

```
tp1() # simple uname of file - successful: exit 0
{
    tpstart "UNAME OUTPUT FOR MANUAL CHECK"

    check_exit "uname -a" 0            # check exit value

    infofile out.stdout                # send output to journal

    check_nostderr                     # should be no stderr

    tpresult INSPECT                   # set result code
}
```

Since the output from `uname` will be different on every machine, this information needs to be reported and then verified. Here the information is being printed out for the tester to see and check; the test purpose result is INSPECT to indicate that the tester must inspect the output in the journal.

8.7 Cleaning up test cases

Since test cases often change and/or create data, it is important to cleanup this data before exiting the test case. As explained earlier, one way to do this is to specify a cleanup function with dTET2's `tet_cleanup` utility. The `cleanup` function is the most practical place to specify the removal of temporary files.

9. A simple distributed test suite

9.1 Introduction

This chapter describes the dTET2 demonstration test suite. The demonstration consists of a simple test that executes on two systems. This test suite is useful in that it helps to delineate the basic components of a distributed test suite in its simplest form.

More substantial examples of non-distributed test cases, are presented elsewhere in this guide. The chapter entitled ‘‘Writing a C language API-conforming test suite’’ describes a ‘C’ language API-conforming test suite with various tests for the `chmod`, `fileno`, `stat` and `uname` system calls. This is not a distributed test case; it expects to execute on the master system only and shows how dTET2 can be used to execute test cases in a similar fashion to TET 1.10. The chapter entitled ‘‘Writing a Shell language API-conforming test suite’’ describes a Shell language API-conforming test suite with various tests for the `chmod` and `uname` user level commands. This is also a non-distributed test case. No support for distributed test cases is provided with the Shell API.

9.2 Test suite files

The following figure shows the file structure of the demonstration test suite on the master system. Exactly the same structure is replicated on the slave system except that the `tetdist.cfg`, `tet_code` and `tet_scen` files are not present. It is not necessary for the value of *tet-root* to be the same on each system because configuration variables are available to define it separately for each system.

Each of these files is described below.

The following figure shows the component files of the example distributed test suite:

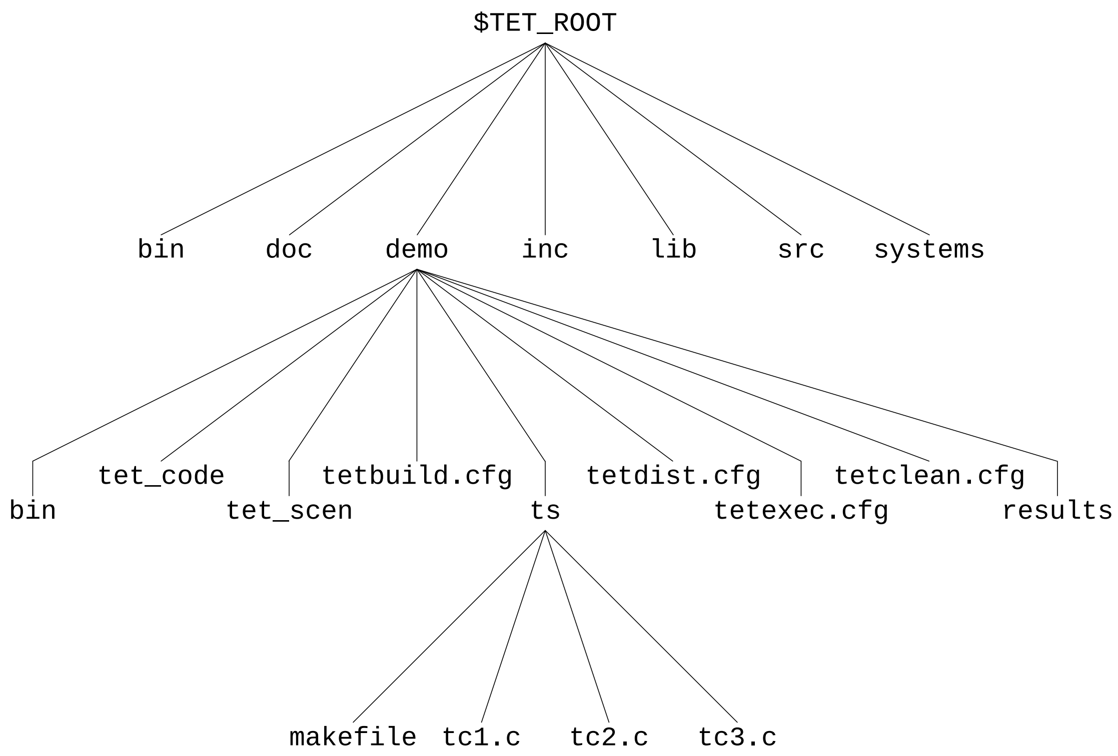

Figure 6. Directory structure for the demonstration test suite

NOTE: For more detailed descriptions about the code shown below, such as the functions listed in the test cases, see the chapter entitled ‘‘Writing a C language API-conforming test suite’’ elsewhere in this guide.

9.2.1 The `systems` file

This file contains the mappings that assign system identifiers to host names. The file must be located in the *tet-root* directory on each system participating in the test. In the demonstration it contains the following lines. In general, it will need to be customised for each installation.

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

If you are using a version of dTET2 that uses XTI as the network transport interface you will need to add an additional field to the second line in this file. The extra field contains the address of the STCC (tccd daemon) on the slave system. The format of this address is defined in the section entitled ‘‘System definitions’’ elsewhere in this guide.

If you are using a version of dTET2 that uses Berkeley Sockets for the transport interface you only need to replace the names `master` and `slave` with system names suitable for your installation. In particular, ensure that these host names are in the host database on both the master and the slave system (often in the file `/etc/hosts`).

9.2.2 The tet_scen file

The tet_scen file contains the test suite's scenario, or control, definitions. The tet_scen file contains the following lines:

```
#       Demonstration test suite.
all
        "starting scenario"
        :remote,000,001:
        /ts/tc1
        /ts/tc2
        "next is the last test case"
        /ts/tc3
        :endremote:
        "done"
```

This file controls the execution sequence of the test suite. The lines in double quotation marks are comments that are echoed into the journal file (see the journal file below). The other lines list the individual test cases, relative to the test suite root directory; their order indicates the order in which they are executed.

The :remote,000,001: and :endremote: keywords indicate that the three test cases are to be executed on the two systems designated 000 and 001.

9.2.3 The tetbuild.cfg file

The tetbuild.cfg file contains the test suite's build parameters. The tetbuild.cfg file contains the following lines:

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

This file sets three configuration variables specific to dTET2:

1TET_OUTPUT_CAPTURE is used here to specify that all output for standard output and standard error should be captured by the journal file, rather than being sent to the terminal (or elsewhere if you have re-directed standard output and/or standard error). It also means that the exit status of each build is interpreted as a result code in which an exit value of 0 means TET_PASS and any other value means TET_FAIL.

1TET_BUILD_TOOL specifies the make utility as the command to use for building the test cases.

1TET_BUILD_FILE specifies the file of instructions to feed to the build tool. Note that this is relative to the directory in which the test case source resides.

Therefore, when dTET2 goes to build the test cases the following command will be executed:

make -f makefile *test-case*

9.2.4 The `tetclean.cfg` file

The `tetclean.cfg` file contains the test suite's clean parameters. The `tetclean.cfg` file contains the following lines:

```
TET_OUTPUT_CAPTURE=True
TET_CLEAN_TOOL=rm
TET_CLEAN_FILE=-f
```

The variables here are similar to the variables shown above. Again, output to standard output and standard error are set to be captured by the dTET2 journal file. When the test cases are cleaned up, the following command will be executed:

`rm -f` *test-case*

9.2.5 The `tetexec.cfg` file

The `tetexec.cfg` file contains the test suite's execution parameters. The `tetexec.cfg` file contains the following lines:

```
TET_OUTPUT_CAPTURE=False
```

Setting the `TET_OUTPUT_CAPTURE` variable to `False` means that execution statements such as `printf` and `echo` will not be collected in the journal file but will instead be printed to standard output and standard error. Test results are communicated to the MTCC via a results file created by the API.

9.2.6 The `tetdist.cfg` file

This file is only required on the master system, and must always be located in the test suite root directory. It contains variable assignments that specify parameters for slave systems that are equivalent to those parameters on the master system that `tcc` obtains from environment variables or deduces from the current working directory. It also defines parameters for the network interface used by distributed test cases. In the demonstration it contains values for the `TET_ROOT` and `TET_TSROOT` variables that define the **tet root** and **test suite root** directories on the slave system. In the example, it contains the lines:

```
TET_REM001_TET_ROOT=/home/dtet2/dtet2
TET_REM001_TET_TSROOT=/home/dtet2/dtet2/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
```

`TET_REM001_` is a prefix used to define variables on the master system that overwrite variables on a slave system (in this case, a slave with system designation 001). The name of the variable to be overwritten is the the same as the name on the master, after removing the prefix.

The last three variables are only required for versions of dTET2 that use XTI as the network transport interface. They are defined in the section entitled ''Configuration variables'' elsewhere in this guide. You must ensure that the variables are set appropriately for your installation.

9.2.7 The `makefile` file

The `makefile` is used to build the test cases during build mode, as was specified in the `tetbuild.cfg` file. The `makefile` contains the following lines:

```
LIBDIR  = ../../lib/dtet2
INCDIR  = ../../inc/dtet2
CC      = cc
CFLAGS  = -I$(INCDIR)

all:    tc1 tc2 tc3

tc1:    tc1.c $(INCDIR)/tet_api.h
        $(CC) $(CFLAGS) -o tc1 tc1.c $(LIBDIR)/tcm.o $(LIBDIR)/libapi.a

tc2:    tc2.c $(INCDIR)/tet_api.h
        $(CC) $(CFLAGS) -o tc2 tc2.c $(LIBDIR)/tcm.o $(LIBDIR)/libapi.a

tc3:    tc3.c $(INCDIR)/tet_api.h
        $(CC) $(CFLAGS) -o tc3 tc3.c $(LIBDIR)/tcm.o $(LIBDIR)/libapi.a
```

This shows a typical makefile, listing the individual test cases and how they are built. The makefile is the same on the master and slave systems.

If you are using a version of dTET2 that is built with the XTI transport interface, you will probably need to append `–lxti` to the compiler `$(CC)` command line. It is possible that other changes will be required, depending on your system configuration.

9.2.8 The `tc1.c` file

`tc1.c` is the source file for the first test case to be executed. It is compiled during build mode, as shown above. `tc1.c` contains the following lines, on both the master and slave systems:

```
#include <stdio.h>
#include <tet_api.h>

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;
void tp1();

struct tet_testlist tet_testlist[] = { {tp1,1}, {NULL,0} };

void tp1()
{
        tet_infoline("This is the first test case (tc1)");
        tet_result(TET_PASS);
}
```

All that is happening in this test case is that the `tet_infoline` argument is being sent to the journal file and the `tet_result` argument of `TET_PASS` is being sent to the journal file, indicating that the test case passed. In this journal file you can see that both the master and slave parts of the test case generate this message. Also, since both produce a result of `TET_PASS` this is the overall result for the test.

9.2.9 The `tc2.c` file

`tc2.c` is the source file for the second test case to be executed. It is compiled during build mode, as shown above, and on the master system contains the following lines:

```
#include <stdio.h>
#include <tet_api.h>

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;
void tp1();

struct tet_testlist tet_testlist[] = { {tp1,1}, {NULL,0} };

void tp1()
{
        tet_infoline("This is the second test case (tc2, master)");
        tet_result(TET_PASS);
}
```

On the slave system, it contains the following lines:

```
#include <stdio.h>
#include <tet_api.h>

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;
void tp1();

struct tet_testlist tet_testlist[] = { {tp1,1}, {NULL,0} };

void tp1()
{
        tet_infoline("This is the second test case (tc2, slave)");
        tet_result(TET_FAIL);
}
```

The only difference from the first example is that the test case part executing on the slave issues a result of `TET_FAIL` instead of `TET_PASS`. This example illustrates the principle that each part of a distributed test purpose is expected to contribute to the overall result for that test purpose. dTET2 uses an arbitration process to resolve differences. In this example the overall result resolves to `TET_FAIL`.

9.2.10 The `tc3.c` file

`tc3.c` is the source file for the third test case to be executed. It is compiled during build mode, as shown above, and on the master system contains the following lines:

```
#include <stdlib.h>
#include <tet_api.h>

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;
void tp1();

struct tet_testlist tet_testlist[] = { {tp1,1}, {NULL,0} };

void tp1()
{
```

```
        int sysid;
        int sync[2];

        tet_infoline("This is the third test case (tc3, master)");

        tet_infoline("sync with slave system (sysid:001)");

        sync[0] = 1;
        sync[1] = 0;
        if (tet_sync((long)1, sync, 20) < 0) {
                tet_infoline("sync failed");
                tet_result(TET_UNRESOLVED);
        }
        else
                tet_result(TET_PASS);
}
```

On the slave system, it contains the following lines:

```
#include <stdlib.h>
#include <tet_api.h>

void (*tet_startup)() = NULL, (*tet_cleanup)() = NULL;
void tp1();

struct tet_testlist tet_testlist[] = { {tp1,1}, {NULL,0} };

void tp1()
{
        int sysid;

        tet_infoline("This is the third test case (tc3, slave)");

        tet_infoline("sync with master system (sysid:000)");

        if (tet_sync((long)1, (int *)0, 20) < 0) {
                tet_infoline("sync failed");
                tet_result(TET_UNRESOLVED);
        }
        else
                tet_result(TET_PASS);
}
```

This example shows the use of `tet_sync()` to mark a user synchronisation point in the execution of the distributed test case. Both master and slave synchronise using a sync point number 1. If the synchronisation is successful, each side goes on to produce a result of `TET_PASS`. A timeout of `20` seconds is provided. Although this timeout is hardly required in the this example, the facility is useful in complex tests where one part of the test needs to reach a point matching a comparable point in the other. In such cases one of the parts may need to wait until both sides reach this agreed point.

In addition to `tet_sync()`, dTET2 provides automatic synchronisation between the TCMs participating in the distributed test. The TCMs synchronise before executing start-up functions (if any are defined), at the start of each invocable component, and at the start and end of each test purpose. By making use of these predefined synchronisation points you can simplify the writing of test purposes that need synchronisation services. For more information on how dTET2 synchronisation works, see the chapter entitled ‘‘Test case synchronisation’’ in the dTET2 Installation and User Guide.

9.3 Test suite execution

The following command line will build, execute, and cleanup the test suite:

```
tcc -bec demo
```

The default scenario name `all` matches the keyword `all` in the scenario file `tet_scen`. The scenario file might include other test scenario definitions to run the test cases in a different order, for example, or even to run a different set of test cases.

Executing `tcc` as shown above results in messages being written to the terminal (when `TET_OUTPUT_CAPTURE` is set to `False`) as well as messages being written to the journal file. These messages and the journal file are described below.

9.3.1 Messages written to the terminal

The following text is written to the terminal during execution:

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

9.3.2 The journal file — unformatted

The test case journal is written to a file called `journal` that is created in a directory below the `results` directory. The `results` directory is created by `tcc` in the `demo` directory if it does not already exist. The following information is written to the journal file:

```
0|2.0 13:50:19 19930920|User: tet (664) TCC Start, \
        Command Line: tcc -bec demo
20|/home/dtet2/dtet2/demo/tetbuild.cfg 0|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=True
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
30||TET_BUILD_TOOL=make
30||TET_BUILD_FILE=-f makefile
40||Config End
20|/home/dtet2/dtet2/demo/tetexec.cfg 1|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=False
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
40||Config End
20|/home/dtet2/dtet2/demo/tetclean.cfg 2|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=True
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
30||TET_CLEAN_TOOL=rm
30||TET_CLEAN_FILE=-f
40||Config End
20|/home/dtet2/dtet2/demo/tetdist.cfg 3|Config Start
```

```
30||TET_REM000_TET_ROOT=/home/dtet2/dtet2
30||TET_REM000_TET_TSROOT=/home/dtet2/dtet2/demo
30||TET_REM000_TET_EXECUTE=
30||TET_REM000_TET_TMP_DIR=/home/dtet2/dtet2/tet_tmp_dir/12199a
30||TET_REM001_TET_ROOT=/home/dtet2/dtet2
30||TET_REM001_TET_TSROOT=/home/dtet2/dtet2/demo
30||TET_REM002_TET_ROOT=/home/dtet2/dtet2
30||TET_REM002_TET_TSROOT=/home/dtet2/dtet2/demo
30||TET_XTI_TPI=/dev/tcp
30||TET_XTI_MODE=tcp
30||TET_LOCALHOST=89.200.0.7
40||Config End
20|remote_001 0|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=True
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
30||TET_BUILD_TOOL=make
30||TET_BUILD_FILE=-f makefile
40||Config End
20|remote_001 1|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=False
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
40||Config End
20|remote_001 2|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=True
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
30||TET_CLEAN_TOOL=rm
30||TET_CLEAN_FILE=-f
40||Config End
70||"starting scenario"
800|000,001|Remote Start, scenario ref 4-1
110|0 /ts/tc1 13:50:20|Build Start, scenario ref 5-1
100|0|  cc -I../../inc/dtet2 -o tc1 tc1.c ../../lib/dtet2/tcm.o \
100|0|          ../../lib/dtet2/libapi.a
100|0|
130|0 0 13:50:26|Build End
110|0 /ts/tc1 13:50:20|Build Start, scenario ref 5-1
100|0|  cc -I../../inc/dtet2 -o tc1 tc1.c ../../lib/dtet2/tcm.o \
100|0|          ../../lib/dtet2/libapi.a -lxti
130|0 0 13:50:26|Build End
10|1 /ts/tc1 13:50:26|TC Start, scenario ref 5-1
15|1 1.0b 1|TCM Start
400|1 1 1 13:50:28|IC Start
200|1 1 13:50:28|TP Start
520|1 1 00012220 1 1|This is the first test case (tc1)
520|1 1 00116558 1 1|This is the first test case (tc1)
220|1 1 0 13:50:28|PASS
410|1 1 1 13:50:28|IC End
80|1 0 13:50:30|TC End
10|1 /ts/tc1 13:50:26|TC Start, scenario ref 5-1
```

```
80|1 0 13:50:29|TC End
300|2 /ts/tc1 13:50:30|Clean Start, scenario ref 5-1
320|2 0 13:50:30|Clean End
300|2 /ts/tc1 13:50:30|Clean Start, scenario ref 5-1
320|2 0 13:50:31|Clean End
110|3 /ts/tc2 13:50:31|Build Start, scenario ref 6-1
100|3|  cc -I../../inc/dtet2 -o tc2 tc2.c ../../lib/dtet2/tcm.o \
100|3|          ../../lib/dtet2/libapi.a
130|3 0 13:50:36|Build End
110|3 /ts/tc2 13:50:31|Build Start, scenario ref 6-1
100|3|  cc -I../../inc/dtet2 -o tc2 tc2.c ../../lib/dtet2/tcm.o \
100|3|          ../../lib/dtet2/libapi.a -lxti
130|3 0 13:50:36|Build End
10|4 /ts/tc2 13:50:36|TC Start, scenario ref 6-1
15|4 1.0b 1|TCM Start
400|4 1 1 13:50:38|IC Start
200|4 1 13:50:38|TP Start
520|4 1 00012241 1 1|This is the second test case (tc2, master)
520|4 1 00116572 1 1|This is the second test case (tc2, slave)
220|4 1 1 13:50:39|FAIL
410|4 1 1 13:50:39|IC End
80|4 0 13:50:40|TC End
10|4 /ts/tc2 13:50:36|TC Start, scenario ref 6-1
80|4 0 13:50:39|TC End
300|5 /ts/tc2 13:50:40|Clean Start, scenario ref 6-1
320|5 0 13:50:41|Clean End
300|5 /ts/tc2 13:50:40|Clean Start, scenario ref 6-1
320|5 0 13:50:41|Clean End
70||"next is the last test case"
110|6 /ts/tc3 13:50:41|Build Start, scenario ref 9-1
100|6|  cc -I../../inc/dtet2 -o tc3 tc3.c ../../lib/dtet2/tcm.o \
100|6|          ../../lib/dtet2/libapi.a
130|6 0 13:50:46|Build End
110|6 /ts/tc3 13:50:41|Build Start, scenario ref 9-1
100|6|  cc -I../../inc/dtet2 -o tc3 tc3.c ../../lib/dtet2/tcm.o \
100|6|          ../../lib/dtet2/libapi.a -lxti
130|6 0 13:50:46|Build End
10|7 /ts/tc3 13:50:46|TC Start, scenario ref 9-1
15|7 1.0b 1|TCM Start
400|7 1 1 13:50:49|IC Start
200|7 1 13:50:49|TP Start
520|7 1 00116586 1 1|This is the third test case (tc3, slave)
520|7 1 00116586 1 2|sync with master (sysid: 000)
520|7 1 00012262 1 1|This is the third test case (tc3, master)
520|7 1 00012262 1 2|sync with slave (sysid: 001)
220|7 1 0 13:50:49|PASS
410|7 1 1 13:50:49|IC End
80|7 0 13:50:50|TC End
10|7 /ts/tc3 13:50:47|TC Start, scenario ref 9-1
80|7 0 13:50:50|TC End
300|8 /ts/tc3 13:50:50|Clean Start, scenario ref 9-1
320|8 0 13:50:52|Clean End
300|8 /ts/tc3 13:50:50|Clean Start, scenario ref 9-1
320|8 0 13:50:52|Clean End
820||Remote End
```

```
70||"done"
900|13:50:52|TCC End
```

Even without understanding the specific codes in the `journal` file, the information clearly indicates how `tcc` sequenced through the control file, building, executing, and cleaning up the three test cases.

9.3.3 The journal file — formatted

More presentable results can be created from the `journal` file by using a filter to extract the pertinent information. The following script uses `awk` to print the third field of specific lines in the journal file:

```
# simple report filter
awk '
BEGIN { FS = "|" }
{
        if ( $1 == 0 || $1 == 10 || $1 == 70 || $1 == 80 || \
                $1 == 220 || $1 == 520 || $1 == 900 )
        {
                line = $3
                for ( i = 4 ; i <= NF ; i++ )
                        line = line "|" $i

                print line
        }
}'
```

The script shown above was named `filter` and was placed in the `bin` directory for the test suite. It was used to generate the formatted report shown below.

```
cat demo/results/0001bec/journal | demo/bin/filter > demo.report
```

The `demo.report` file contains the following lines:

```
User: tet (664) TCC Start, Command Line: tcc -bec demo
"starting scenario"
TC Start, scenario ref 5-1
This is the first test case (tc1,slave)
This is the first test case (tc1,master)
PASS
TC End
TC Start, scenario ref 5-1
TC End
TC Start, scenario ref 6-1
This is the second test case (tc2, master)
This is the second test case (tc2, slave)
FAIL
TC End
TC Start, scenario ref 6-1
TC End
TC Start, scenario ref 7-1
This is the third test case (tc3, master)
sync with slave (sysid: 001)
This is the third test case (tc3, slave)
sync with master (sysid: 000)
PASS
```

```
TC End
TC Start, scenario ref 7-1
TC End
"done"
TCC End
```

APPENDICES

A. Example C language API test suite source files

A.1 Introduction

This appendix contains listings for the files that comprise the example ‘C’ language test suite presented in the chapter entitled ‘‘Writing a C language API-conforming test suite’’.

A.2 `tet_code`

```
# TET reserved codes
0 "PASS"
1 "FAIL"
2 "UNRESOLVED"
3 "NOTINUSE"
4 "UNSUPPORTED"
5 "UNTESTED"
6 "UNINITIATED"
7 "NORESULT"

# Test suite additional codes
33 "INSPECT"
```

A.3 `install`

```
echo This is the C-API test suite install tool.
```

A.4 `cleantool`

```
exec make clean
```

A.5 `tet_scen`

```
#       chmod, fileno, stat, uname test suite.

all
        "Starting Full Test Suite"
        /ts/chmod/chmod-tc
        /ts/fileno/fileno-tc
        /ts/stat/stat-tc
        /ts/uname/uname-tc
        "Completed Full Test Suite"

chmod
        "Starting chmod Test Case"
        /ts/chmod/chmod-tc
        "Finished chmod Test Case"

fileno
        "Starting fileno Test Case"
        /ts/fileno/fileno-tc
        "Finished fileno Test Case"

stat
        "Starting stat Test Case"
        /ts/stat/stat-tc
```

```
        "Finished stat Test Case"

uname
        "Starting uname Test Case"
        /ts/uname/uname-tc
        "Finished uname Test Case"

# EOF
```

A.6 tetbuild.cfg

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

A.7 tetexec.cfg

```
TET_OUTPUT_CAPTURE=False

# The name of a character device file (or "unsup" if not supported)
CHARDEV=/dev/null

# The name of a block device file (or "unsup" if not supported)
BLOCKDEV=unsup
```

A.8 tetclean.cfg

```
TET_OUTPUT_CAPTURE=True
TET_CLEAN_TOOL=cleantool
TET_CLEAN_FILE=
```

A.9 Makefile for chmod-tc.c

```
TET_ROOT = ../../..
LIBDIR   = $(TET_ROOT)/lib/dtet2
INCDIR   = $(TET_ROOT)/inc/dtet2
CC       = cc
CFLAGS   = -I$(INCDIR) -D_POSIX_SOURCE

chmod-tc:       chmod-tc.c $(INCDIR)/tet_api.h
                $(CC) $(CFLAGS) -o chmod-tc chmod-tc.c $(LIBDIR)/tcm.o \
                        $(LIBDIR)/libapi.a
                -rm -f chmod-tc.o

clean:
                rm -f chmod-tc chmod-tc.o

lint:
                lint $(CFLAGS) chmod-tc.c -ltcm
```

A.10 chmod-tc.c

```
/* chmod-tc.c : test case for chmod() interface */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <tet_api.h>

static void tp1(), tp2(), tp3();
static void startup(), cleanup();

/* Initialize TCM data structures */
void (*tet_startup)() = startup;
void (*tet_cleanup)() = cleanup;
struct tet_testlist tet_testlist[] = {
    { tp1, 1 },
    { tp2, 2 },
    { tp3, 3 },
    { NULL, 0 }
};

/* Test Case Wide Declarations */
static char *tfile = "chmod.1";         /* test file name */
static char *tndir = "chmod.1/chmod.1"; /* path with non-directory in prefix */
static struct stat buf;                 /* buffer for stat(ing) file */
static char msg[256];                   /* buffer for info lines */

static void
startup()
{
    int fd;
    static char *reason = "Failed to create test file in startup";

    if ((fd=creat(tfile, S_IRWXU)) < 0)
    {
        (void) sprintf(msg,
            "creat(\"%s\", S_IRWXU) failed in startup - errno %d",
            tfile, errno);
        tet_infoline(msg);

        /* Prevent tests which use this file from executing */
        tet_delete(1, reason);
        tet_delete(3, reason);
    }
    else
        (void) close(fd);
}

static void
cleanup()
{
    /* remove file created by start-up */
    (void) unlink(tfile);
}
```

```
static void
tp1()         /* successful chmod of file: return 0 */
{
    int ret, err;
    mode_t mode;

    tet_infoline("SUCCESSFUL CHMOD OF FILE");

    /* change mode of file created in startup function */

    errno = 0;
    if ((ret=chmod(tfile, (mode_t)0)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "chmod(\"%s\", 0) returned %d, expected 0",
            tfile, ret);
        tet_infoline(msg);
        if (err != 0)
        {
            (void) sprintf(msg, "errno was set to %d", err);
            tet_infoline(msg);
        }
        tet_result(TET_FAIL);
        return;
    }

    /* check mode was changed correctly */

    if (stat(tfile, &buf) == -1)
    {
        (void) sprintf(msg,
            "stat(\"%s\", buf) failed - errno %d", tfile, errno);
        tet_infoline(msg);
        tet_result(TET_UNRESOLVED);
        return;
    }

    mode = buf.st_mode & O_ACCMODE;
    if (mode != 0)
    {
        (void) sprintf(msg, "chmod(\"%s\", 0) set mode to 0%lo, expected 0",
            tfile, (long)mode);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else
        tet_result(TET_PASS);
}

static void
tp2()       /* chmod of non-existent file: return -1, errno ENOENT */
{
    int ret, err;

    tet_infoline("CHMOD OF NON-EXISTENT FILE");

    /* ensure file does not exist */

    if (stat("chmod.2", &buf) != -1 && unlink("chmod.2") == -1)
```

```
        {
            tet_infoline("could not unlink chmod.2");
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* check return value and errno set by call */

        errno = 0;
        ret = chmod("chmod.2", (mode_t)0);

        if (ret != -1 || errno != ENOENT)
        {
            err = errno;
            if (ret != -1)
            {
                (void) sprintf(msg,
                    "chmod(\"chmod.2\", 0) returned %d, expected -1", ret);
                tet_infoline(msg);
            }

            if (err != ENOENT)
            {
                (void) sprintf(msg,
                    "chmod(\"chmod.2\", 0) set errno to %d, \
                            expected %d (ENOENT)", err, ENOENT);
                tet_infoline(msg);
            }

            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }

    static void
    tp3()       /* non-directory path component: return -1, errno ENOTDIR */
    {
        int ret, err;

        tet_infoline("CHMOD OF NON-DIRECTORY PATH PREFIX COMPONENT");

        /* tndir is a pathname containing a plain file (created by the
           startup function) in the prefix */

        errno = 0;
        ret = chmod(tndir, (mode_t)0);

        /* check return value and errno set by call */

        if (ret != -1 || errno != ENOTDIR)
        {
            err = errno;
            if (ret != -1)
            {
                (void) sprintf(msg,
                    "chmod(\"%s\", 0) returned %d, expected -1", tndir, ret);
                tet_infoline(msg);
            }
```

```
            if (err != ENOTDIR)
            {
                (void) sprintf(msg,
                    "chmod(\"%s\", 0) set errno to %d, expected %d (ENOTDIR)",
                    tndir, err, ENOTDIR);
                tet_infoline(msg);
            }

            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }
```

A.11 Makefile for `fileno-tc.c`

```
TET_ROOT = ../../..
LIBDIR   = $(TET_ROOT)/lib/dtet2
INCDIR   = $(TET_ROOT)/inc/dtet2
CC       = cc
CFLAGS   = -I$(INCDIR) -D_POSIX_SOURCE

fileno-tc:      fileno-t4 fileno-tc.c $(INCDIR)/tet_api.h
                $(CC) $(CFLAGS) -o fileno-tc fileno-tc.c $(LIBDIR)/tcm.o \
                        $(LIBDIR)/libapi.a
                -rm -f fileno-tc.o

fileno-t4:      fileno-t4.c $(INCDIR)/tet_api.h
                $(CC) $(CFLAGS) -o fileno-t4 fileno-t4.c \
                        $(LIBDIR)/tcmchild.o $(LIBDIR)/libapi.a
                -rm -f fileno-t4.o

clean:
                rm -f fileno-tc fileno-tc.o fileno-t4 fileno-t4.o

lint:
                lint $(CFLAGS) fileno-tc.c -ltcm
                lint $(CFLAGS) fileno-t4.c -ltcmc
```

A.12 `fileno-tc.c`

```
/* fileno-tc.c : test case for fileno() interface */

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

#include <tet_api.h>

extern char **environ;

static void cleanup();
static void tp1(), tp2(), tp3(), tp4(), ch4();

/* Initialize TCM data structures */
```

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

/* Test Case Wide Declarations */
static char msg[256];                   /* buffer for info lines */

static void
cleanup()
{
    (void) unlink("fileno.1");
    (void) unlink("fileno.4");
}

static void
tp1()         /* successful fileno: return fd associated with stream */
{
    FILE *fp;
    struct stat buf1, buf2;

    tet_infoline("FD RETURNED BY FILENO REFERS TO FILE OPEN ON STREAM");

    /* open stream to test file */

    if ((fp=fopen("fileno.1", "w")) == NULL)
    {
        (void) sprintf(msg, "fopen(\"fileno.1\", \"w\") failed - errno %d",
            errno);
        tet_infoline(msg);
        tet_result(TET_UNRESOLVED);
        return;
    }

    /* check device and inode numbers from file descriptor associated
       with the stream match those from the file itself */

    if (stat("fileno.1", &buf1) == -1)
    {
        (void) sprintf(msg, "stat(\"fileno.1\", buf1) failed - errno %d",
            errno);
        tet_infoline(msg);
        tet_result(TET_UNRESOLVED);
        return;
    }

    if (fstat(fileno(fp), &buf2) == -1)
    {
        (void) sprintf(msg, "fstat(fileno(fp), buf2) failed - errno %d",
            errno);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else if (buf1.st_ino != buf2.st_ino || buf1.st_dev != buf2.st_dev)
```

```
        {
            tet_infoline("fileno(fp) does not refer to same file as fp");
            (void) sprintf(msg, "st_dev, st_ino of file: 0x%lx, %ld",
                (long)buf1.st_dev, (long)buf1.st_ino);
            tet_infoline(msg);
            (void) sprintf(msg, "st_dev, st_ino of fileno(fp): 0x%lx, %ld",
                (long)buf2.st_dev, (long)buf2.st_ino);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);

        (void) fclose(fp);
    }

    static void
    tp2()         /* fileno on stdin/stdout/stderr: return 0/1/2 */
    {
        int fd, fail = 0;

        tet_infoline("FILENO ON STDIN/STDOUT/STDERR");

        /* check return value of fileno() for stdin/stdout/stderr */
        /* this code relies on the fact that the TCM does not interfere
           with these streams */

        if ((fd = fileno(stdin)) != 0)
        {
            (void) sprintf(msg, "fileno(stdin) returned %d, expected 0", fd);
            tet_infoline(msg);
            tet_result(TET_FAIL);
            fail = 1;
        }

        if ((fd = fileno(stdout)) != 1)
        {
            (void) sprintf(msg, "fileno(stdout) returned %d, expected 1", fd);
            tet_infoline(msg);
            tet_result(TET_FAIL);
            fail = 1;
        }

        if ((fd = fileno(stderr)) != 2)
        {
            (void) sprintf(msg, "fileno(stderr) returned %d, expected 2", fd);
            tet_infoline(msg);
            tet_result(TET_FAIL);
            fail = 1;
        }

        if (fail == 0)
            tet_result(TET_PASS);
    }

    static void
    tp3()         /* on entry to main(), stdin is readable, stdout and stderr
                     are writable */
    {
```

```
int flags, fail = 0;

tet_infoline("ON ENTRY TO MAIN, STDIN IS READABLE, STDOUT AND STDERR \
    ARE WRITABLE");
/* this code relies on the fact that the TCM does not interfere
   with these streams */

/* check file descriptor associated with stdin is readable */

if ((flags = fcntl(fileno(stdin), F_GETFL)) == -1)
{
    (void) sprintf(msg, "fcntl(fileno(stdin), F_GETFL) failed - errno %d",
        errno);
    tet_infoline(msg);
    tet_result(TET_UNRESOLVED);
    return;
}

flags &= O_ACCMODE;
if (flags != O_RDONLY && flags != O_RDWR)
{
    tet_infoline("stdin is not readable");
    fail = 1;
}

/* check file descriptor associated with stdout is writable */

if ((flags = fcntl(fileno(stdout), F_GETFL)) == -1)
{
    (void) sprintf(msg, "fcntl(fileno(stdout), F_GETFL) failed - errno %d",
        errno);
    tet_infoline(msg);
    tet_result(TET_UNRESOLVED);
    return;
}

flags &= O_ACCMODE;
if (flags != O_WRONLY && flags != O_RDWR)
{
    tet_infoline("stdout is not writable");
    fail = 1;
}

/* check file descriptor associated with stderr is writable */

if ((flags = fcntl(fileno(stderr), F_GETFL)) == -1)
{
    (void) sprintf(msg, "fcntl(fileno(stderr), F_GETFL) failed - errno %d",
        errno);
    tet_infoline(msg);
    tet_result(TET_UNRESOLVED);
    return;
}

flags &= O_ACCMODE;
if (flags != O_WRONLY && flags != O_RDWR)
{
    tet_infoline("stderr is not writable");
    fail = 1;
```

```
    }

    if (fail == 0)
        tet_result(TET_PASS);
    else
        tet_result(TET_FAIL);
}

static void
tp4()           /* on entry to main(), stream position of stdin, stdout and
                   stderr is same as fileno(stream) */
{
    tet_infoline("ON ENTRY TO MAIN, STREAM POSITION OF STDIN, STDOUT \
        AND STDERR");

    /* fork and execute subprogram, so that unique file positions can be
       set up on entry to main() in subprogram */

    (void) tet_fork(ch4, TET_NULLFP, 30, 0);
}

static void
ch4()
{
    int fd, ret;
    static char *args[] = { "./fileno-t4", NULL };

    /* set up file positions to be inherited by stdin/stdout/stderr
       in subprogram */

    for (fd = 0; fd < 3; fd++)
    {
        (void) close(fd);
        if ((ret=open("fileno.4", O_RDWR|O_CREAT, S_IRWXU)) != fd)
        {
            (void) sprintf(msg, "open() returned %d, expected %d", ret, fd);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }
        if (lseek(fd, (off_t)(123 + 45*fd), SEEK_SET) == -1)
        {
            (void) sprintf(msg, "lseek() failed - errno %d", errno);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }
    }

    /* execute subprogram to carry out remainder of test */

    (void) tet_exec(args[0], args, environ);

    (void) sprintf(msg, "tet_exec(\"%s\", args, env) failed - errno %d",
        args[0], errno);
    tet_infoline(msg);
    tet_result(TET_UNRESOLVED);
}
```

A.13 fileno-t4.c

```
/* fileno-t4.c : child program of test purpose 4 for fileno() */

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#include <tet_api.h>

static char msg[256];                   /* buffer for info lines */

/* ARGSUSED */

int
tet_main(argc, argv)
int argc;
char **argv;
{
    long ret, pos;
    int fd, err, fail = 0;
    static FILE *streams[] =  {  stdin,   stdout,   stderr };
    static char *strnames[] = { "stdin", "stdout", "stderr" };

    /* check file positions of streams are same as set up in parent */

    for (fd = 0; fd < 3; fd++)
    {
        pos = 123 + 45*fd; /* must match lseek() in parent */
        errno = 0;
        if ((ret = ftell(streams[fd])) != pos)
        {
            err = errno;
            (void) sprintf(msg, "ftell(%s) returned %ld, expected %ld",
                strnames[fd], ret, pos);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            fail = 1;
        }
    }

    if (fail == 0)
        tet_result(TET_PASS);
    else
        tet_result(TET_FAIL);

    return 0;
}
```

A.14 Makefile for `stat-tc.c`

```
TET_ROOT = ../../..
LIBDIR   = $(TET_ROOT)/lib/dtet2
INCDIR   = $(TET_ROOT)/inc/dtet2
CC       = cc
CFLAGS   = -I$(INCDIR) -D_POSIX_SOURCE

stat-tc:        stat-tc.c $(INCDIR)/tet_api.h
                $(CC) $(CFLAGS) -o stat-tc stat-tc.c $(LIBDIR)/tcm.o \
                        $(LIBDIR)/libapi.a
                -rm -f stat-tc.o

clean:
                rm -f stat-tc stat-tc.o

lint:
                lint $(CFLAGS) stat-tc.c -ltcm
```

A.15 `stat-tc.c`

```
/* stat-tc.c : test case for stat() interface */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <tet_api.h>

static void tp1(), tp2(), tp3(), tp4(), tp5(), tp6(), tp7();
static void startup(), cleanup();

/* Initialize TCM data structures */
void (*tet_startup)() = startup;
void (*tet_cleanup)() = cleanup;
struct tet_testlist tet_testlist[] = {
    { tp1, 1 },
    { tp2, 2 },
    { tp3, 3 },
    { tp4, 4 },
    { tp5, 5 },
    { tp6, 6 },
    { tp7, 7 },
    { NULL, 0 }
};

/* Test Case Wide Declarations */
static char *tfile = "stat.1";          /* test file name */
static char *tndir = "stat.1/stat.1";   /* path with non-directory in prefix */
static struct stat buf;                 /* buffer for stat(ing) file */
static char msg[256];                   /* buffer for info lines */

static void
startup()
{
    int fd;
    static char *reason = "Failed to create test file in startup";
```

```
    if ((fd=creat(tfile, S_IRWXU)) < 0)
    {
        (void) sprintf(msg,
            "creat(\"%s\", S_IRWXU) failed in startup - errno %d",
            tfile, errno);
        tet_infoline(msg);

        /* Prevent tests which use this file from executing */
        tet_delete(1, reason);
        tet_delete(7, reason);
    }
    else
        (void) close(fd);
}

static void
cleanup()
{
    /* remove file created by start-up */
    (void) unlink(tfile);

    /* remove files created by test purposes, in case they don't run
       to completion */
    (void) rmdir("stat.d");
    (void) unlink("stat.p");
}

static void
tp1()           /* successful stat of plain file: return 0 */
{
    int ret, err;

    tet_infoline("SUCCESSFUL STAT OF PLAIN FILE");

    /* stat file created in startup function and check mode indicates
       a plain file */

    errno = 0;
    if ((ret=stat(tfile, &buf)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
            tfile, ret);
        tet_infoline(msg);
        if (err != 0)
        {
            (void) sprintf(msg, "errno was set to %d", err);
            tet_infoline(msg);
        }
        tet_result(TET_FAIL);
    }
    else if (!S_ISREG(buf.st_mode))
    {
        tet_infoline("S_ISREG(st_mode) was not true for plain file");
        (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
```

```
    else
        tet_result(TET_PASS);
}

static void
tp2()         /* successful stat of directory: return 0 */
{
    int ret, err;
    char *tdir = "stat.d";

    tet_infoline("SUCCESSFUL STAT OF DIRECTORY");

    /* create a test directory */

    if (mkdir(tdir, S_IRWXU) == -1)
    {
        (void) sprintf(msg,
            "mkdir(\"%s\", S_IRWXU) failed in startup - errno %d",
            tdir, errno);
        tet_infoline(msg);
        tet_result(TET_UNRESOLVED);
        return;
    }

    /* stat the directory and check mode indicates a directory */

    errno = 0;
    if ((ret=stat(tdir, &buf)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
            tdir, ret);
        tet_infoline(msg);
        if (err != 0)
        {
            (void) sprintf(msg, "errno was set to %d", err);
            tet_infoline(msg);
        }
        tet_result(TET_FAIL);
    }
    else if (!S_ISDIR(buf.st_mode))
    {
        tet_infoline("S_ISDIR(st_mode) was not true for directory");
        (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else
        tet_result(TET_PASS);

    (void) rmdir(tdir);
}

static void
tp3()         /* successful stat of FIFO file: return 0 */
{
    int ret, err;
    char *tfifo = "stat.p";
```

```
        tet_infoline("SUCCESSFUL STAT OF FIFO");

        /* create a test FIFO */

        if (mkfifo(tfifo, S_IRWXU) == -1)
        {
            (void) sprintf(msg,
                "mkfifo(\"%s\", S_IRWXU) failed in startup - errno %d",
                tfifo, errno);
            tet_infoline(msg);
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* stat the FIFO and check mode indicates a FIFO */

        errno = 0;
        if ((ret=stat(tfifo, &buf)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
                tfifo, ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else if (!S_ISFIFO(buf.st_mode))
        {
            tet_infoline("S_ISFIFO(st_mode) was not true for FIFO file");
            (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);

        (void) unlink(tfifo);
    }

    static void
    tp4()         /* successful stat of character device file: return 0 */
    {
        int ret, err;
        char *chardev;

        tet_infoline("SUCCESSFUL STAT OF CHARACTER DEVICE FILE");

        /* obtain device name from execution configuration parameter */

        chardev = tet_getvar("CHARDEV");
        if (chardev == NULL || *chardev == '\0')
        {
            tet_infoline("parameter CHARDEV is not set");
            tet_result(TET_UNRESOLVED);
            return;
```

```
        }

        /* check if parameter indicates character devices are not supported */

        if (strcmp(chardev, "unsup") == 0)
        {
            tet_infoline("parameter CHARDEV is set to \"unsup\"");
            tet_result(TET_UNSUPPORTED);
            return;
        }

        /* stat the device and check mode indicates a character device */

        errno = 0;
        if ((ret=stat(chardev, &buf)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
                chardev, ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else if (!S_ISCHR(buf.st_mode))
        {
            (void) sprintf(msg, "S_ISCHR(st_mode) was not true for \"%s\"",
                chardev);
            tet_infoline(msg);
            (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }

    static void
    tp5()         /* successful stat of block device file: return 0 */
    {
        int ret, err;
        char *blockdev;

        tet_infoline("SUCCESSFUL STAT OF BLOCK DEVICE FILE");

        /* obtain device name from execution configuration parameter */

        blockdev = tet_getvar("BLOCKDEV");
        if (blockdev == NULL || *blockdev == '\0')
        {
            tet_infoline("parameter BLOCKDEV is not set");
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* check if parameter indicates block devices are not supported */
```

```
        if (strcmp(blockdev, "unsup") == 0)
        {
            tet_infoline("parameter BLOCKDEV is set to \"unsup\"");
            tet_result(TET_UNSUPPORTED);
            return;
        }

        /* stat the device and check mode indicates a block device */

        errno = 0;
        if ((ret=stat(blockdev, &buf)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
                blockdev, ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else if (!S_ISBLK(buf.st_mode))
        {
            (void) sprintf(msg, "S_ISBLK(st_mode) was not true for \"%s\"",
                blockdev);
            tet_infoline(msg);
            (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
            tet_infoline(msg);
            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }

    static void
    tp6()       /* stat of non-existent file: return -1, errno ENOENT */
    {
        int ret, err;

        tet_infoline("STAT OF NON-EXISTENT FILE");

        /* ensure file does not exist */

        if (stat("stat.6", &buf) != -1 && unlink("stat.6") == -1)
        {
            tet_infoline("could not unlink stat.6");
            tet_result(TET_UNRESOLVED);
            return;
        }

        /* check return value and errno set by call */

        errno = 0;
        ret = stat("stat.6", &buf);

        if (ret != -1 || errno != ENOENT)
        {
```

```
            err = errno;
            if (ret != -1)
            {
                (void) sprintf(msg,
                    "stat(\"stat.6\", 0) returned %d, expected -1", ret);
                tet_infoline(msg);
            }

            if (err != ENOENT)
            {
                (void) sprintf(msg,
                    "stat(\"stat.6\", 0) set errno to %d, expected %d (ENOENT)",
                    err, ENOENT);
                tet_infoline(msg);
            }

            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
    }

    static void
    tp7()        /* non-directory path component: return -1, errno ENOTDIR */
    {
        int ret, err;

        tet_infoline("STAT OF NON-DIRECTORY PATH PREFIX COMPONENT");

        /* tndir is a pathname containing a plain file (created by the
           startup function) in the prefix */

        errno = 0;
        ret = stat(tndir, &buf);

        /* check return value and errno set by call */

        if (ret != -1 || errno != ENOTDIR)
        {
            err = errno;
            if (ret != -1)
            {
                (void) sprintf(msg,
                    "stat(\"%s\", 0) returned %d, expected -1", tndir, ret);
                tet_infoline(msg);
            }

            if (err != ENOTDIR)
            {
                (void) sprintf(msg,
                    "stat(\"%s\", 0) set errno to %d, expected %d (ENOTDIR)",
                    tndir, err, ENOTDIR);
                tet_infoline(msg);
            }

            tet_result(TET_FAIL);
        }
        else
            tet_result(TET_PASS);
```

```
}
```

A.16 Makefile for `uname-tc.c`

```
TET_ROOT = ../../..
LIBDIR   = $(TET_ROOT)/lib/dtet2
INCDIR   = $(TET_ROOT)/inc/dtet2
CC       = cc
CFLAGS   = -I$(INCDIR) -D_POSIX_SOURCE

uname-tc:       uname-tc.c $(INCDIR)/tet_api.h
                $(CC) $(CFLAGS) -o uname-tc uname-tc.c $(LIBDIR)/tcm.o \
                        $(LIBDIR)/libapi.a
                -rm -f uname-tc.o

clean:
                rm -f uname-tc uname-tc.o

lint:
                lint $(CFLAGS) uname-tc.c -ltcm
```

A.17 `uname-tc.c`

```
/* uname-tc.c : test case for uname() interface */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/utsname.h>

#include <tet_api.h>

#undef TET_INSPECT      /* must undefine because TET_ is reserved prefix */
#define TET_INSPECT 33  /* this would normally be in a test suite header */

static void tp1();

/* Initialize TCM data structures */
void (*tet_startup)() = NULL;   /* no start-up function */
void (*tet_cleanup)() = NULL;   /* no clean-up function */
struct tet_testlist tet_testlist[] = {
    { tp1, 1 },
    { NULL, 0 }
};

/* Test Case Wide Declarations */
static char msg[256];                  /* buffer for info lines */

static void
tp1()         /* successful uname: return 0 */
{
    int ret, err;
    struct utsname name;

    tet_infoline("UNAME OUTPUT FOR MANUAL CHECK");

    /* The test cannot determine automatically whether the information
       returned by uname() is correct.  It therefore outputs the
       information with an INSPECT result code for checking manually. */
```

```
        errno = 0;
        if ((ret=uname(&name)) != 0)
        {
            err = errno;
            (void) sprintf(msg, "uname() returned %d, expected 0", ret);
            tet_infoline(msg);
            if (err != 0)
            {
                (void) sprintf(msg, "errno was set to %d", err);
                tet_infoline(msg);
            }
            tet_result(TET_FAIL);
        }
        else
        {
            (void) sprintf(msg, "System Name:  \"%s\"", name.sysname);
            tet_infoline(msg);
            (void) sprintf(msg, "Node Name:    \"%s\"", name.nodename);
            tet_infoline(msg);
            (void) sprintf(msg, "Release:      \"%s\"", name.release);
            tet_infoline(msg);
            (void) sprintf(msg, "Version:      \"%s\"", name.version);
            tet_infoline(msg);
            (void) sprintf(msg, "Machine Type: \"%s\"", name.machine);
            tet_infoline(msg);

            tet_result(TET_INSPECT);
        }
    }
```

B. Example Shell API test suite source files

B.1 Introduction

This appendix contains listings for the files that comprise the example Shell test suite presented in the chapter entitled ‘‘Writing a Shell language API-conforming test suite’’.

B.2 tet_code

```
# TET reserved codes
0 "PASS"
1 "FAIL"
2 "UNRESOLVED"
3 "NOTINUSE"
4 "UNSUPPORTED"
5 "UNTESTED"
6 "UNINITIATED"
7 "NORESULT"

# Test suite additional codes
33 "INSPECT"
```

B.3 install

```
echo Installing SHELL-API test suite

cd SHELL-API || exit 1

# create alternate execution directory hierarchy
find ts -type d -print |
while read d
do
    if test ! -d ts_exec/"$d"
    then mkdir ts_exec/"$d"
    fi
done
```

B.4 buildtool

```
:
# Check TET_EXECUTE is set
if [ -z "$TET_EXECUTE" ]
then
        echo >&2 "No alternate execution directory supplied to buildtool"
        exit 1
fi

# Set TET_EXECUTE on command line to override default value in makefile
exec make TET_EXECUTE="$TET_EXECUTE"
```

B.5 cleantool

```
:
# Check TET_EXECUTE is set
if [ -z "$TET_EXECUTE" ]
then
        echo >&2 "No alternate execution directory supplied to cleantool"
        exit 1
fi

# Set TET_EXECUTE on command line to override default value in makefile
exec make TET_EXECUTE="$TET_EXECUTE" clean
```

B.6 tet_scen

```
#       chmod, uname test suite.
all
        "Starting Full Test Suite"
        /ts/chmod/chmod-tc
        /ts/uname/uname-tc
        "Completed Full Test Suite"
chmod
        "Starting chmod Test Case"
        /ts/chmod/chmod-tc
        "Finished chmod Test Case"
uname
        "Starting uname Test Case"
        /ts/uname/uname-tc
        "Finished uname Test Case"
# EOF
```

B.7 tetbuild.cfg

```
TET_OUTPUT_CAPTURE=True
TET_BUILD_TOOL=buildtool
```

B.8 tetexec.cfg

```
TET_OUTPUT_CAPTURE=False
TET_EXEC_IN_PLACE=True

# highest signal number plus 1
TET_NSIG=22
```

B.9 tetclean.cfg

```
TET_OUTPUT_CAPTURE=True
TET_CLEAN_TOOL=cleantool
```

B.10 `shfuncs` — common functions used in the Shell API test suite

```
# shfuncs : test suite common shell functions

tpstart() # write test purpose banner and initialize variables
{
    tet_infoline "$*"
    FAIL=N
}

tpresult() # give test purpose result
{
    # $1 is result code to give if FAIL=N (default PASS)
    if [ $FAIL = N ]
    then
        tet_result ${1-PASS}
    else
        tet_result FAIL
    fi
}

check_exit() # execute command (saving output) and check exit code
{
    # $1 is command, $2 is expected exit code (0 or "N" for non-zero)
    eval "$1" > out.stdout 2> out.stderr
    CODE=$?
    if [ $2 = 0 -a $CODE -ne 0 ]
    then
        tet_infoline "Command ($1) gave exit code $CODE, expected 0"
        FAIL=Y
    elif [ $2 != 0 -a $CODE -eq 0 ]
    then
        tet_infoline "Command ($1) gave exit code $CODE, expected non-zero"
        FAIL=Y
    fi
}

check_nostdout() # check that nothing went to stdout
{
    if [ -s out.stdout ]
    then
        tet_infoline "Unexpected output written to stdout, as shown below:"
        infofile out.stdout stdout:
        FAIL=Y
    fi
}

check_nostderr() # check that nothing went to stderr
{
    if [ -s out.stderr ]
    then
        tet_infoline "Unexpected output written to stderr, as shown below:"
        infofile out.stderr stderr:
        FAIL=Y
```

```
        fi
    }

    check_stderr() # check that stderr matches expected error
    {
        # $1 is file containing regexp for expected error
        # if no argument supplied, just check out.stderr is not empty

        case $1 in
        "")
            if [ ! -s out.stderr ]
            then
                tet_infoline "Expected output to stderr, but none written"
                FAIL=Y
            fi
            ;;
        *)
            expfile="$1"
            OK=Y
            exec 4<&0 0< "$expfile" 3< out.stderr
            while read expline
            do
                if read line <&3
                then
                    if expr "$line" : "$expline" > /dev/null
                    then
                        :
                    else
                        OK=N
                        break
                    fi
                else
                    OK=N
                fi
            done
            exec 0<&4 3<&- 4<&-
            if [ $OK = N ]
            then
                tet_infoline "Incorrect output written to stderr, as shown below"
                infofile "$expfile" "expected stderr:"
                infofile out.stderr "received stderr:"
                FAIL=Y
            fi
            ;;
        esac
    }

    infofile() # write file to journal using tet_infoline
    {
        # $1 is file name, $2 is prefix for tet_infoline

        prefix=$2
        while read line
        do
            tet_infoline "$prefix$line"
        done < $1
```

```
}
```

B.11 Makefile for `chmod-tc.sh`

```
TET_EXECUTE = ../../ts_exec
INSTALL_DIR = $(TET_EXECUTE)/ts/chmod

$(INSTALL_DIR)/chmod-tc: chmod-tc.sh clean
        cp chmod-tc.sh $@
        chmod 755 $@

clean:
        rm -f $(INSTALL_DIR)/chmod-tc
```

B.12 `chmod-tc.sh`

```
:
# chmod-tc.sh : test case for chmod command

tet_startup=""                          # no startup function
tet_cleanup="cleanup"                   # cleanup function
iclist="ic1 ic2 ic3"                    # list invocable components
ic1="tp1"                               # functions for ic1
ic2="tp2"                               # functions for ic2
ic3="tp3"                               # functions for ic3

tp1() # simple chmod of file - successful: exit 0
{
    tpstart "SIMPLE CHMOD OF FILE: EXIT 0"

    echo x > chmod.1 2> out.stderr      # create file
    if [ ! -f chmod.1 ]
    then
        tet_infoline "Could not create test file: chmod.1"
        tet_infoline ‘cat out.stderr‘
        tet_result UNRESOLVED
        return
    fi

    check_exit "chmod 777 chmod.1" 0    # check exit value

    MODE=‘ls -l chmod.1 |cut -d" " -f1‘ # get and check mode of file
    if [ X"$MODE" != X"-rwxrwxrwx" ]
    then
        tet_infoline "chmod 777 set mode to $MODE, expected -rwxrwxrwx"
        FAIL=Y
    fi

    check_nostdout                      # should be no stdout
    check_nostderr                      # should be no stderr

    tpresult                            # set result code
}

tp2() # chmod of non-existent file : exit non-zero
{
    tpstart "CHMOD OF NON-EXISTENT FILE: EXIT NON-ZERO"

    # ensure test file does not exist
```

```
        rm -f chmod.2 2> out.stderr
        if [ -f chmod.2 ]
        then
            tet_infoline "Could not remove test file: chmod.2"
            tet_infoline `cat out.stderr`
            tet_result UNRESOLVED
            return
        fi

        check_exit "chmod 777 chmod.2" N    # check exit value

        check_nostdout                      # should be no stdout
        check_stderr                        # check error message

        tpresult                            # set result code
    }

    tp3() # chmod with invalid syntax: exit non-zero
    {
        tpstart "CHMOD WITH INVALID SYNTAX: EXIT NON-ZERO"

        # expected error message
        echo "chmod: illegal option -- :\n.*" > out.experr

        check_exit "chmod -:" N             # check exit value

        check_nostdout                      # should be no stdout
        check_stderr out.experr             # check error message

        tpresult                            # set result code
    }

    cleanup() # clean-up function
    {
         rm -f out.stdout out.stderr out.experr
         rm -f chmod.1
    }

    # source common shell functions
    . $TET_EXECUTE/lib/shfuncs

    # execute shell test case manager - must be last line
    . $TET_ROOT/lib/xpg3sh/tcm.sh
```

B.13 Makefile for `uname-tc.sh`

```
TET_EXECUTE = ../../ts_exec
INSTALL_DIR = $(TET_EXECUTE)/ts/uname

$(INSTALL_DIR)/uname-tc: uname-tc.sh clean
        cp uname-tc.sh $@
        chmod 755 $@

clean:
        rm -f $(INSTALL_DIR)/uname-tc
```

B.14 uname-tc.sh

```
:
# uname-tc.sh : test case for uname command

tet_startup=""                          # no startup function
tet_cleanup="cleanup"                   # cleanup function
iclist="ic1 ic2"                        # list invocable components
ic1="tp1"                               # functions for ic1
ic2="tp2"                               # functions for ic2

tp1() # simple uname of file - successful: exit 0
{
    tpstart "UNAME OUTPUT FOR MANUAL CHECK"

    check_exit "uname -a" 0             # check exit value

    infofile out.stdout                 # send output to journal

    check_nostderr                      # should be no stderr

    tpresult INSPECT                    # set result code
}

tp2() # uname with invalid syntax: exit non-zero
{
    tpstart "UNAME WITH INVALID SYNTAX: EXIT NON-ZERO"

    # expected error message
    echo "uname: illegal option -- :\n.*" > out.experr

    check_exit "uname -:" N             # check exit value

    check_nostdout                      # should be no stdout
    check_stderr out.experr             # check error message

    tpresult                            # set result code
}

cleanup() # clean-up function
{
     rm -f out.stdout out.stderr out.experr
}

# source common shell functions
. $TET_EXECUTE/lib/shfuncs

# execute shell test case manager - must be last line
. $TET_ROOT/lib/xpg3sh/tcm.sh
```

CONTENTS

LIST OF FIGURES