Test Environment Toolkit

Implementation for POSIX.1 Systems in ‘C’ and XPG3 Shell
by OSF, UNIX International, and X/Open

Programmer’s Guide

TET Workshop
Revision: 1.7 (92/05/07)

!!!!! DISCARD THIS PAGE !!!!!

1. Introduction

1.1 Overview

1.1.1 Intended Audience

This guide has been designed for developers of test suites, system testers, and sophisticated end users who have had some exposure to the concepts of system conformance testing.

1.1.2 Product Definition

This document describes the Test Environment Toolkit (TET), a set of tools for the development and execution of system and unit tests. The goal behind creating TET 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

The TET project started in September 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 organizations agreed to develop and make freely available an implementation written to that specification; additionally, the three organizations 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 now expects that a substantial portion of the test development community will begin using TET for the development of conformance testing software.

1.1.4 General Testing Model and Methodology

The project developed specifications and an implementation for the following:

- A test case controller.
- A test suite installation utility.
- A test report generator utility.
- Various programmatic interfaces to be used in the development of test cases.

The project has tried to allow for future extensibility and flexibility of TET 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 TET. 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, TET imposes a minimum amount of structure on test cases (see chapters 2 and 4 for test suite structure information).

1.1.5 Organization

This manual is organized as follows:

- Chapter 1 describes the manual's organization and documentation conventions. It also provides some history regarding TET, a discussion of testing methodology, and a feature list.
- Chapter 2 describes the basic structure of test cases and test suites. It also presents a walk-through of the demo test suite provided with TET.
- Chapter 3 explains test execution, focusing on the Test Case Controller (TCC) and its implementation, the `tcc` command. It describes the TCC's various modes of operation and the parameters for configuring TET to meet the needs of different environments.
- Chapter 4 describes the formats of files requred to be supplied by test suites.
- Chapter 5 explains how Test Cases are managed, and describes the C and Shell language APIs
- Chapter 6 describes how reporting and journaling is performed by TET.
- Chapter 7 describes a sample test suite that conforms to TET's C language binding of the API.
- Chapter 8 describes a sample test suite that conforms to TET's shell language binding of the API.
- Appendix A contains manual pages for the following commands: `rpt`, `tet_ins`, and `tcc`.
- Appendix B contains a listing of source files that compose the sample C language test suite described in chapter 7.
- Appendix C contains a listing of source files that compose the sample shell language test suite described in chapter 8.

1.1.6 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.
`constant width`	Command names, file names, and data that you are instructed to enter, output produced by TET or by the operating system, such as error messages, program prompts, or retrieved data, and code fragments, appear in constant width 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.

1.2 Architecture

The architecture of the Test Environment Toolkit provides simplicity and portability, which helps to increase the transportability of test suites. The architecture description can be broken down into three sections:

- The Terminology, which loosely defines the elements of the TET Structure.
- The System Requirements, which define minimum system facilities necessary to support this implementation of the TET functionality.
- The TET Structure description, which shows how elements in the TET implementation link together.

1.2.1 Terminology

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 TET, 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 instructions to be carried out by the build tool when preparing test cases for execution.

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 TET API.

Clean File A clean file is a set of instructions to be carried out by the clean tool when 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 TET 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 behavior of the TCC, the build tool, and the clean tool. Configuration variables are set via configuration variable files and via the TCC user interface. TET 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.

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 An execution results file is a temporary file into which test cases, via the API, place information regarding their execution. This information is later incorporated into the journal by the TCC.

Invocable Component An invocable component is the smallest unit that the TCC can execute individually. Invocable components are made up of one or more test purposes. Usually test purposes that are closely related and for which individual execution is not necessary are grouped together within a single invocable component. In this way they are always executed together.

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

Result Code A result code is the determination made by a test purpose as to the status of the test it performed. TET 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 analyzed by the report generator and appropriate treatment filters.

Scenario File A scenario file is a file containing test scenario definitions.

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

Test Case Controller (TCC)

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

Test Case 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.

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 a 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.

1.2.2 Basic System Requirements

In order for a system to support TET's architecture, the following facilities must be available.

- ISO 9945-1 1990 conformance.
- ISO 9899 or Common Usage C (as defined in ISO 9945-1) conforming compiler.
- The command shell, `sh`, as defined in the *X/Open Portability Guide*, Issue 3 Volume 1, and some commonly-used utilities (such as `make`, `cp`, and `rm`).

1.2.3 TET Structure

The structure of TET consists of six functions (represented by boxes) and ten data files (represented by ellipses). The figure below depicts the three TCC modes, build, execute, and cleanup, and conveys a general view of the sequence of events and the inter-relationship of the files and functions. For a more concrete picture of what the various data files can look like and how they relate to each other, see the section, "A Simple Test Suite", in chapter 2.

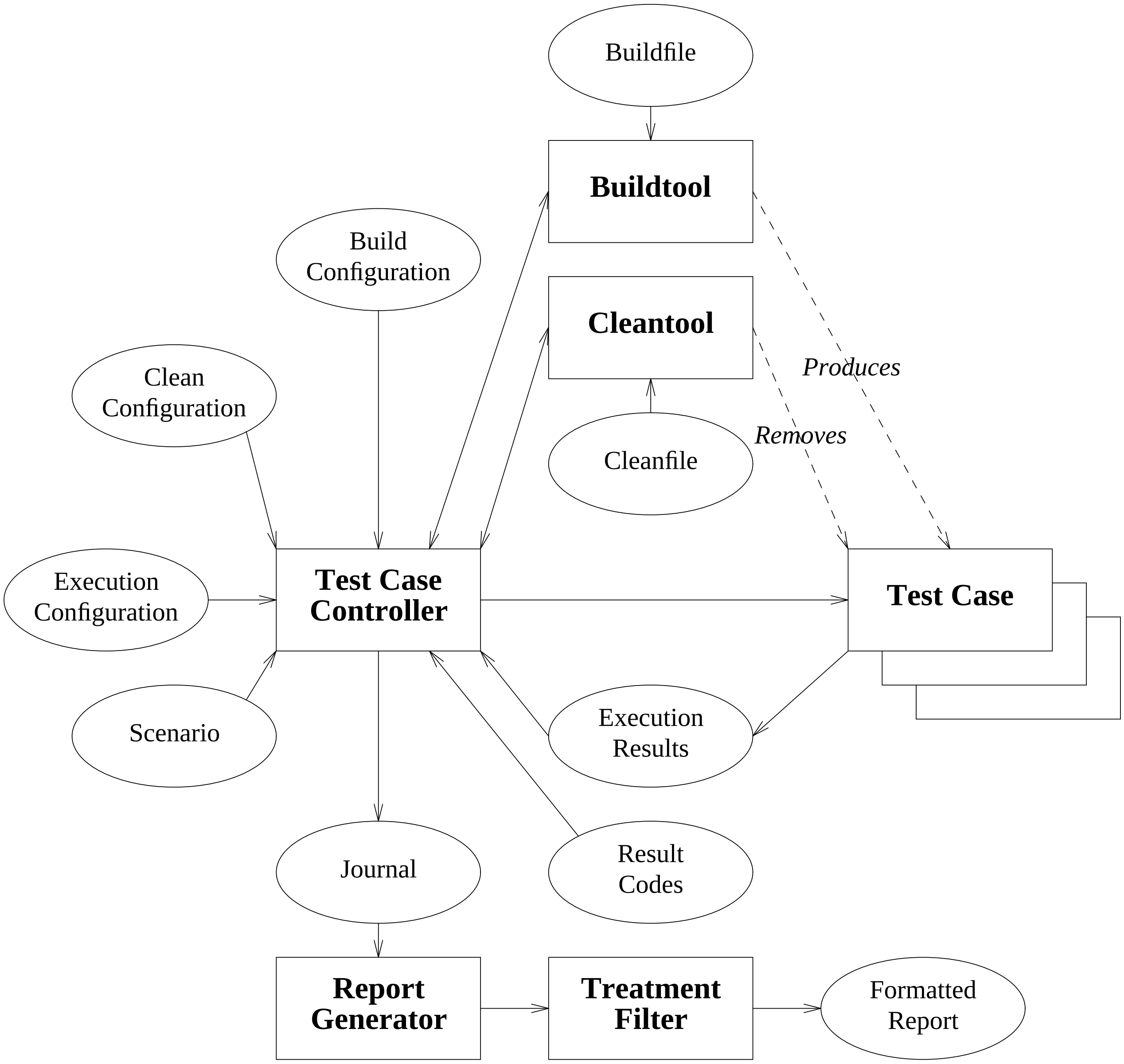

Figure 1. TET Structure

1.3 TET Features

The following list highlights the basic features of TET. Subsequent sections describe the features of individual components of TET in more detail.

- **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 TCC allows 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:** TET is capable of executing test cases that are distributed as binary executables.
- **Result Codes:** TET 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:** TET's report generation utility executes test suite or user supplied report generation filters.

1.3.1 Test Case Controller Features

The following list highlights features specific to the TCC.

- **Selective Execution:** The TCC allows users to select which invocable components are to be run. The names of invocable components must be made available to TET 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 TCC), or the test scenario (by sending a SIGQUIT to the TCC). 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 TCC allows all test cases in a test scenario to be built before they are all executed (by means of two successive invocations of the TCC). 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 TCC keeps 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 TCC are readily definable by users through test scenario creation or by selection on the tcc command line.
- **Isolation:** The TCC does 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 TCC provides cleanup of test case space, which prevents test cases from corrupting each other or inducing inter-test data collisions.

- **Black-Box Testing:** Test cases communicate with the TCC 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 TCC 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 TCC allows tests to interact with an operator.

1.3.2 API Features

- **Sequencing:** The TCM ensures that invocable components are processed in the sequence defined in the test scenario or on the command line.
- **Audit Trails:** The TCM marks 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 allows a test to be built in such a way that it can be executed without the TCC. Compilation of stand-alone tests is thereby made possible with a minimum of difficulty.
- **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 Synchronization:** The API provides facilities that assist in synchronizing multiple processes associated with a single invocable component.
- **Result Code Journaling:** The API provides functionality that records messages and result codes to an execution results file. These messages are delivered to the file 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.

1.3.3 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 TCC 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.

!!!!! DISCARD THIS PAGE !!!!!

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 TET API and are invoked by the TET TCM. This relationship is illustrated through the following picture:

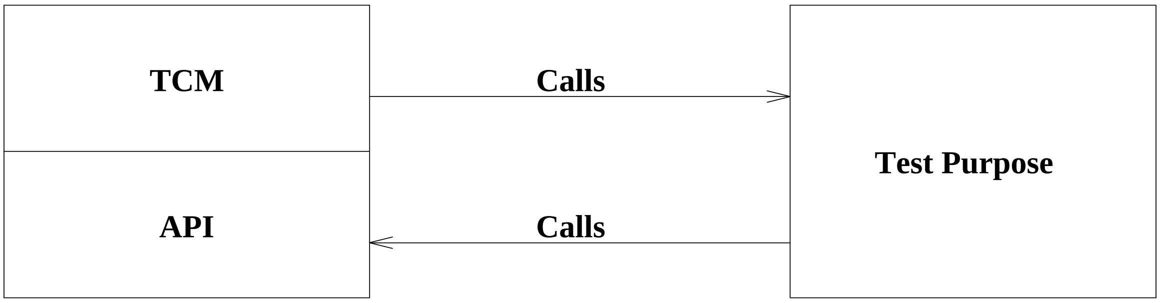

Figure 2. Test Case Interaction

The picture shows that the TCM interacts directly with the test purposes, which in turn interact directly with the API. The API performs functions such as fetching configuration variable settings and writing messages into the journal.

Test purposes within a test case are placed in groups called invocable components. An invocable component is a set of one or more test purposes which are always executed together in an application defined sequence. There is no limit to the number of test purposes that can be grouped in an invocable component; and there is no 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, chapter 3, and in chapter 5, which discuss test suite structure, test suite execution, and test suite development, respectively. Specific information about constructing test cases can be found in chapters 7 and 8, which provide example test suites in both the 'C' and Shell APIs.

2.2 Test Suite Structure

TET 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 "TET File Formats" section in chapter 4.

2.2.1 Required Files and Directories

Configuration Variable Settings
TET has several configuration variables that can be used to customize the way in which the TCC operates. Test suites may define additional configuration variables as needed. Files containing the test suite settings for these variables are used by the API and TCC when building, executing, and cleaning up 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 TCC and TET users when executing the test suite. At a minimum, a scenario named `all` must be provided. This scenario should execute all invocable components in a test suite.

Installation Utility The installation utility is expected to perform test suite specific installation tasks. One such task might be the initial setting of configuration variables. The installation utility may interact with the operator; it must be called `install` and be located in the `bin` directory of the test suite.

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 TCC.

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 TET users.

Build Tool This utility is activated when the TCC is executed with build mode specified. 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 **TET_OUTPUT_CAPTURE** to be enabled.

Clean Tool This utility is activated when the TCC is executed with clean mode specified. 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 the TCC 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 The TET report generator delivers a journal to report treatment filters. Test suite authors may wish to provide treatment filters that produce reports in specific formats. The format of the journal file is described in chapter 6.

2.3 A Simple Test Suite

The following sections describe the demo test suite delivered with the TET source code. The demo test suite is useful in that it helps to delineate the basic components of a test suite built for TET.

For more concrete examples of test cases, see chapter 7 and chapter 8. Chapter 7 describes a C language API conforming test suite with various tests for the `chmod`, `fileno`, `stat`, and `uname` system calls. Chapter 8 describes a shell language API conforming test suite with various tests for the `chmod` and `uname` user level commands.

2.3.1 Test Suite Files

The following figure shows the file structure of the demo test suite.

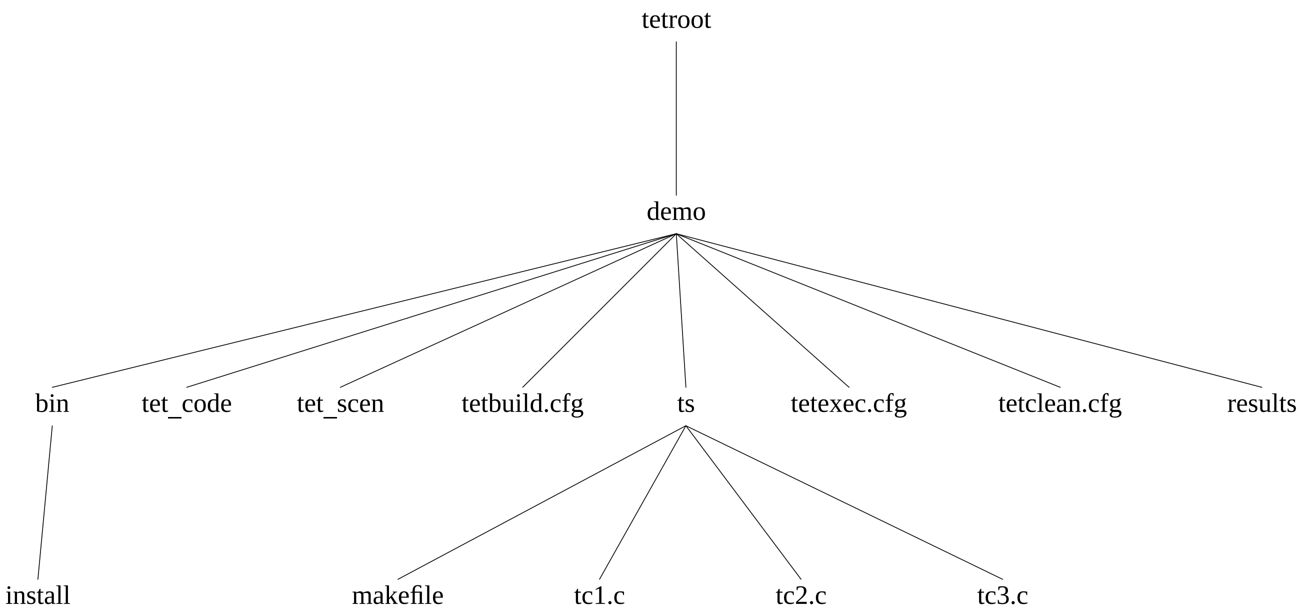

Figure 3. The demo Test Suite

Each of these files is described below.

NOTE: For more detailed descriptions about the code shown below, such as the functions listed in the test cases, see chapter 7.

2.3.1.1 The install File

The `install` file contains the test suite's installation instructions. This file is required by the installation tool, `tet_ins`, supplied with TET.

The `install` file contains the following line:

```
echo This is the demo suite install tool.
```

This file is executed when you run:

```
tet_ins demo
```

The environment variable, `$TET_ROOT`, must first be set to specify the root directory of the test suite. With `$TET_ROOT` defined as `/tetroot`, executing `tet_ins` as shown above produces the following output:

```
     Install tool file = /tetroot/demo/bin/install

This is the demo suite install tool.
```

The installation tool takes the name of the test suite, `demo`, and looks for a file called `install` in `$TET_ROOT/demo/bin`. It then executes that file, which echoes the line onto the terminal. More typical installation scripts would include commands to move files, set their modes, run makefiles, etc. These installation instructions are controlled through the install script supplied with the test suite.

2.3.1.2 The tet_code File

The `tet_code` file contains the test suite's local result code definitions. If present, the codes in this file augment those defined by TET (see section 4.2.3 for more details).

2.3.1.3 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"
        /ts/tc1
        /ts/tc2
        "next is the last test case"
        /ts/tc3
        "done"
```

This file controls the execution sequence of the test suite. The lines in double quotation marks are comments that get 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; and their order indicates the order in which they are executed.

2.3.1.4 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 -f
TET_BUILD_FILE=makefile
```

This file sets three configuration variables specific to TET:

`TET_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). This also means that the exit status of each build is taken to be a result code and acted upon accordingly.

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

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

Therefore, when TET goes to build each test case the following command is executed:

```
make -f makefile <test case>
```

2.3.1.5 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 TET journal file. When the test cases are cleaned up, the following command will be executed:

```
rm -f <test case>
```

2.3.1.6 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. The test case `tc2` includes `printf` statements that will appear on standard output when the test case is executed, as shown below. Test results are communicated to the TCC via an execution results file created by the API.

2.3.1.7 The makefile File

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

```
LIBDIR  = ../../lib/posix_c
INCDIR  = ../../inc/posix_c
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.

2.3.1.8 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:

```
#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.

2.3.1.9 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 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)");
        printf("We have not set TET_OUTPUT_CAPTURE=True ");
        printf("so all normal stdin/stdout/stderr\nfiles are available.  ");
        printf("\nIf we had set output capture,  the results logged by");
        printf(" the API would not be in the journal.\n");
        printf("But these lines would.\n");
        tet_result(TET_PASS);
}
```

This test case includes some `printf` statements that will print to standard output since `TET_OUTPUT_CAPTURE` has been set to `False.` The rest is the same as the first test case.

2.3.1.10 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 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 third test case (tc3)");
        tet_result(TET_PASS);
}
```

This test case is effectively the same as `tc1.c`.

2.3.2 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.

2.3.2.1 Messages Written to the Terminal

The following text is written to the terminal during execution.

```
Journal file is: /usr/TET/demo/results/0001bec/journal
We have not set TET_OUTPUT_CAPTURE=True so all normal stdin/stdout/stderr
files are available.
If we had set output capture, the results logged by the API would not be in the journal.
But these lines would.
```

2.3.2.2 The Journal File - Unformatted

The following information is written to the journal file. The `results` directory is created by `tcc` in the `demo` directory if one does not already exist.

```
0|1.7 11:23:16 19911001|User: fred (146) TCC Start, Command Line: tcc -bec demo
20|/usr/TET/demo/tetbuild.cfg 0|Config Start
30||TET_VERSION=1.7
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|/usr/TET/demo/tetexec.cfg 1|Config Start
30||TET_VERSION=1.7
30||TET_OUTPUT_CAPTURE=False
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
40||Config End
20|/usr/TET/demo/tetclean.cfg 2|Config Start
30||TET_VERSION=1.7
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"
110|0 /ts/tc1 11:23:17|Build Start
100|0|      cc -I../../inc/posix_c -o tc1 tc1.c ../../lib/posix_c/tcm.o ../../lib/posix_c/libapi.a
130|0 0 11:23:26|Build End
10|1 /ts/tc1 11:23:26|TC Start
15|1 1.7 1|TCM Start
```

```
400|1 1 1 11:23:29|IC Start
200|1 1 11:23:29|TP Start
520|1 1 2249 1 1|This is the first test case (tc1)
220|1 1 0 11:23:29|PASS
410|1 1 1 11:23:29|IC End
80|1 0 11:23:30|TC End
300|2 /ts/tc1 11:23:30|Clean Start
320|2 0 11:23:31|Clean End
110|3 /ts/tc2 11:23:31|Build Start
100|3|      cc -I../../inc/posix_c -o tc2 tc2.c ../../lib/posix_c/tcm.o ../../lib/posix_c/libapi.a
130|3 0 11:23:40|Build End
10|4 /ts/tc2 11:23:40|TC Start
15|4 1.7 1|TCM Start
400|4 1 1 11:23:43|IC Start
200|4 1 11:23:43|TP Start
520|4 1 2265 1 1|This is the second test case (tc2)
220|4 1 0 11:23:43|PASS
410|4 1 1 11:23:43|IC End
80|4 0 11:23:44|TC End
300|5 /ts/tc2 11:23:44|Clean Start
320|5 0 11:23:45|Clean End
70||"next is the last test case"
110|6 /ts/tc3 11:23:45|Build Start
100|6|      cc -I../../inc/posix_c -o tc3 tc3.c ../../lib/posix_c/tcm.o ../../lib/posix_c/libapi.a
130|6 0 11:23:52|Build End
10|7 /ts/tc3 11:23:52|TC Start
15|7 1.7 1|TCM Start
400|7 1 1 11:23:56|IC Start
200|7 1 11:23:56|TP Start
520|7 1 2281 1 1|This is the third test case (tc3)
220|7 1 0 11:23:56|PASS
410|7 1 1 11:23:56|IC End
80|7 0 11:23:57|TC End
300|8 /ts/tc3 11:23:57|Clean Start
320|8 0 11:23:58|Clean End
70||"done"
900|11:23:58|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.

2.3.2.3 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 with the `rpt` command, as shown here, to generate the formatted report shown below.

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

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

```
User: fred (146) TCC Start, Command Line: tcc -bec demo
"starting scenario"
TC Start
This is the first test case (tc1)
PASS
TC End
TC Start
This is the second test case (tc2)
PASS
TC End
"next is the last test case"
TC Start
This is the third test case (tc3)
PASS
TC End
"done"
TCC End
```

!!!!! DISCARD THIS PAGE !!!!!

3. Test Execution

3.1 Test Case Controller

This section describes the functionality of the Test Case Controller (TCC); the manual page for the `tcc` command line is given in appendix A.

The TCC accepts user-specific and suite-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 TCC also includes functionality to support internal mechanisms essential to the operation of TET. These include managing interaction with the TCM, the processing of results, and the removal of temporary files.

3.1.1 Modes of Operation

The TCC provides these three modes of operation:

Build translate source test cases into executables.

Execute load and execute test cases.

Clean remove unwanted files.

The behavior 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 TCC 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 TCC performs the following actions during startup:

1. If a specific journal file is named by the user, verifies that it does not exist.
2. Verifies that the test suite selected by the user is available.
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.
5. Ensures that the results directory exists.
6. Tells the user the name of the journal file being used.
7. Writes a startup message into the journal.
8. Writes to the journal file details of option and configuration variable settings for all enabled modes.

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

3.1.1.1 Build Mode

When the test suite is provided in source form, the TCC 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.

For each test case referenced in the specified test scenario, the TCC does the following:

1. Delivers a build start message to the journal.
2. Obtains exclusive locks in the source and execution directories of the test case.
3. Executes the build tool in the source directory of the test case. If output capture mode is enabled, then the arguments passed to the build tool are `TET_BUILD_FILE` and the name of the test case, otherwise no arguments are passed.
4. Transfers the build tool output and error messages to the journal file, if output capture mode is enabled. If output capture mode is disabled, it re-orders and copies the contents of the results file to the journal.
5. If output capture mode is enabled, transfers cleantool output and error messages to the journal file. If output capture mode is disabled, re-orders and copies the contents of the execution results file to the journal.
6. Removes the lock set in step 2.
7. Delivers a build end message to the journal.

If the user has specified an alternate execution directory, the 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 later section in this chapter).
- Set configuration variable values.

3.1.1.2 Execute Mode

In this mode, the TCC executes test cases in the specified scenario. For each test case referenced, the TCC does the following:

1. Delivers a test case start message to the journal.

2. Obtains a shared or exclusive lock in the execution directory of the test case. If `TET_EXEC_IN_PLACE` is false, the lock is shared, otherwise it is exclusive.
3. If `TET_EXEC_IN_PLACE` is false, it creates a temporary directory in which executions can be safely performed.
4. If `TET_EXEC_IN_PLACE` is false, it recursively copies the files in the test case execution directory to the temporary directory.
5. Removes the lock set in step 2, if `TET_EXEC_IN_PLACE` is false.
6. If `TET_EXEC_TOOL` is not defined, executes the test case in the test case execution directory if `TET_EXEC_IN_PLACE` is true, otherwise in the temporary directory, with arguments of the appropriate invocable components. If `TET_EXEC_TOOL` is defined, executes the exec tool in the test case execution directory if `TET_EXEC_IN_PLACE` is true, otherwise in the temporary directory, with arguments of `TET_EXEC_FILE`, the test case name and the appropriate invocable components. Behavior of test cases written to the TET API is detailed in chapter 7.
7. Recursively transfers files matching `TET_SAVE_FILES` to the corresponding place in the saved files directory hierarchy.
8. If output capture mode is disabled, it re-orders and copies the contents of the execution results file to the journal. If any test purpose has not generated a result, the TCC adds a result of `NORESULT`. If output capture mode is enabled, the TCC adds appropriate lines to the journal based on test case output and exit status.
9. Removes the lock set in step 2, if `TET_EXEC_IN_PLACE` is true.
10. Removes the temporary directory, if `TET_EXEC_IN_PLACE` is false.
11. 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 later section in this chapter).
- Set configuration variable values.

3.1.1.3 Clean Mode

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

For each test case referenced, the TCC does the following:

1. Delivers a clean start message to the journal.
2. Obtains exclusive locks in the source and execution directories of the test case.
3. Executes 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. Removes the lock set in step 2.
5. Delivers a clean end message to the journal.

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 later section in this chapter).
- Set configuration variable values.

3.1.2 Major Options

3.1.2.1 Resume Option

When invoked with the resume option, the TCC analyzes 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 TCC 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 which was used to generate the old journal file.

3.1.2.2 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

Modes	Build result match	Execute result match	Clean result match
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

3.1.3 Configuration Variables

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

For each mode the configuration information is obtained from a configuration file. This file is located by the TCC 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.

If the user set specific variables at TCC invocation time, those settings override the ones in the configuration variable file.

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.

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. The following configuration variables are currently defined:

Variable	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_RESCODES_FILE	tet_code
TET_SAVE_FILES	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:

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

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

`TET_BUILD_TOOL` names the build tool.

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

`TET_CLEAN_TOOL` names the clean tool.

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

`TET_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 TCC executes each test case 'in place' in the directory containing the test case executable; when it is set to `False`, the TCC first copies 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.

`TET_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.

`TET_OUTPUT_CAPTURE` enables or disables output capture. Test cases not written using the TET API may write diagnostic and error messages using system output or diagnostic facilities. When this variable is `True`, the TCC pretends that the test case is an executable that has within it a single invocable component numbered 0 with a single test purpose. The TCC generates test case, invocable component, and test purpose start messages and then executes the test case. The output from the test case is recorded as test purpose messages (with a block number of 0). When the test case terminates, the TCC supplies a test purpose result message using the completion status of the process. It then generates invocable component and test case end messages. The default for TCC operation is `False`. Note that when this mode is enabled, the result file processing normally done by the TCC is disabled.

The setting of this variable also affects build and clean modes.

In some environments it may be appropriate to perform tests which require interaction with users. The TCC does not prohibit such interactive testing using normal system provided methods. However, interactive testing will not be possible if the `TET_OUTPUT_CAPTURE` configuration variable is set to `True`.

`TET_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 TET supplied result codes.

`TET_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.

3.1.4 Communication Variables

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

`TET_CONFIG` the pathname of the current configuration variable file.

`TET_CODE` the pathname of the current result codes definition file.

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

`TET_EXECUTE` the pathname of the root of the execution directory hierarchy.

`TET_ROOT` the pathname of the TET root directory.

3.1.5 Journal Entries

The TCC manages updates to the journal file. Upon completion of a test case build, execution or clean, the results are transferred from a temporary file into the journal file. During this transfer, the TCC 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 TCC 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 result 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 context and block identifiers are remembered.
   b. The TCC then looks through the range for additional test purpose information lines with the same 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 TCC 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 TCC 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 TCC supplies a result of `NORESULT` for that test purpose.

3.1.6 Locking

The 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.

In build and clean modes the 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, the 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, the 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.

3.1.7 Exception Conditions

The following exception conditions exist:

Initialization Errors

If the TCC does not locate a user specified or required input file upon test session startup, the TCC 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 TCC will generate an end message for that tool with a status of "-1" and discontinue processing of the scenario.

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

If in execute mode the TCC cannot find a test case or the test case is not executable, the TCC 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 TCC will order the shutdown of the currently executing test case. Test cases written to the TET 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 captured and transferred to the journal.

Data File Corruption

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

Timeouts

A user of TCC may specify a timeout period. Build, execution or cleanup operations which take longer than this timeout (on a per test case basis) will be interrupted by the TCC. In such a case, the TCC 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, if the user has specified a timeout period, then the TCC periodically re-tries the operation until the timeout period has elapsed. If the user has not specified a timeout period, or the timeout period elapses, then 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".

!!!!! DISCARD THIS PAGE !!!!!

4. Test Suite Files

4.1 Installation

Since TET cannot anticipate the internal structure of test suites, it uses test suite supplied utilities to perform some tasks. One such case requires that the test suite provide utilities to assist in installing the test suite.

The installation procedure for a test suite can be very complex or very simple. Instead of providing a generic solution for installation, TET provides an interface which executes a test suite supplied installation tool. After execution of this tool (called `install` in the test suite subdirectory `bin`) the installer checks that the suite structure and the contents of some data files meet TET requirements. In the event of any discrepancy, the installer exits with a diagnostic message.

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.

4.1.1 The tet_ins Command

The test suite installer, `tet_ins`, is supplied with TET to assist with test suite installation. Its syntax is described below.

- **Synopsis**

  `tet_ins` *Suite_name*

- **Description**

  The `tet_ins` utility causes the test suite provided installation tool to be executed. It expects to find a file called `install` in `$TET_ROOT/`*Suite_name*`/bin` . After execution, all TET required data files are checked to ensure that they are syntactically reasonable. Note that this utility does not attempt to move the test suite into the file system. It is the responsibility of the user to ensure that the test suite has been loaded under the main TET directory before executing `tet_ins`.

- **Operands**

  Suite_name names the test suite to be installed.

4.2 TET File Formats

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

4.2.1 Configuration Variables Files

There is a single configuration file per executing test scenario per execution mode. This configuration file is found, by default, in the root directory of the appropriate test suite. 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 chapter 3 for further details.

In a configuration variable file, variables are defined with lines of the 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.

```
# Example configuration variable settings
TET_BUILD_TOOL=make
TET_BUILD_FILE=-f Makefile
```

Figure 4. Sample Configuration File

4.2.2 Scenario Files

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 each test suite. This file must at least define 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 pathnames. In build and clean modes, and in execute mode when there is no alternate execution directory, test case pathnames 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 pathnames are relative to that directory and may only reference files within the alternate execution directory hierarchy. Include file pathnames 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.

```
# Example scenario file
readtest
                /tests/readtests/test1{1,3}
writetest
                /tests/writetests/test1{1,2,3}
all
                /tests/readtests/test1{all}
                "Read Tests completed!"
                /tests/writetests/test1{all}
                "Write Tests completed!"
```

Figure 5. Sample Scenario File

Scenario file data elements may contain any of the following instructions:

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 pathname of a test case (see above), and optional comma separated invocable component names within braces ({,}) (e.g., `/tests/readtests/test1{1,2,3}`).

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 quotation marks (") and may not contain quotation marks (e.g. `"This is a Journal Message"`).

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 pathname to include (e.g. `:include:/tests/readtests/testlist`). Files referenced by include directives consist of entries. Each is either a journal message (see above) or the pathname of a test case with optional invocable components (e.g. `/tests/readtests/test1{1,3}`).

4.2.3 Result Codes

A mapping mechanism for the TCC is provided to interpret results from test suites. This mapping is contained in a result codes file. Upon termination of a test case, the TCC 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 TET or test suite specified results.

TET 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.

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

- 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 the TET implementation. The remainder are available for specification by the test suite author.
- The name of this result. This is a double quotation mark delimited field which can contain any sequence of alphanumeric characters and blanks.
- 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`.

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

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

!!!!! DISCARD THIS PAGE !!!!!

5. Test Case Support

TET supplies functionality to support the development of test cases. This chapter describes that functionality for a "wrapper" that provides a suitable environment for the execution of invocable components as requested by the TCC or directly from the command line. After that it describes the interfaces 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.

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

It is expected that several different language bindings will be made available with TET. This implementation currently provides two:

- 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.

5.1 Test Case Manager

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

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

The general flow of control for test cases written to TET is as follows:

1. The Test Case Manager (TCM) arranges for access to the configuration information.
2. The TCM outputs a TCM start entry to the execution results file.
3. The TCM builds from the list of requested invocable components a list of test purposes to be executed. If no invocable components were requested, or if the special invocable component `all` is requested, the TCM builds this list from all of the invocable components in the test case.
4. The TCM arranges for processing of asynchronous events as specified by the test suite.
5. The TCM executes the requested start up procedure (if any).
6. The TCM starts execution of each requested invocable component and outputs a start entry to the execution results file.
7. The TCM starts execution of each test purpose within an invocable component. As each test purpose is started a test purpose start entry is output to the execution results file.

8. If a signal arrives that interrupts the processing of a test purpose, the TCM reports the receipt of the signal to the execution results file as an information line. The TCM then outputs a test result of `UNRESOLVED` to the execution results file. If the signal is a termination instruction from the TCC then the TCM reports receipt of the message to the execution results file, executes the specified clean up procedure (if any) and exits abnormally; otherwise, it continues by processing the next test purpose.
9. If a test purpose has been marked as canceled, the TCM does not attempt to execute this test purpose. Instead, it outputs an information line detailing the reason for cancellation to the execution results file. The TCM then outputs a test result of `UNINITIATED` to the execution results file and continues by processing the next test purpose.
10. Once all of the test purposes in an invocable component have been executed, the TCM outputs an Invocable Component End message and moves on to the next requested invocable component (See step 6).
11. After all test purposes for all requested invocable components have been executed, the API executes the specified clean up procedure (if any) and exits normally.
12. If, for any reason, the test case fails to execute the requested set of invocable components or the specified start up or clean up procedure, the API will exit abnormally.

5.2 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 chapter 7 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/posix_c/libapi.a` contains the support routines for test purposes.
- `TET_ROOT/lib/posix_c/tcm.o` contains the routine `main()` and associated support routines for the sequencing and control of invocable components and test purposes.
- `TET_ROOT/lib/posix_c/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/inc/posix_c/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`) and declarations/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 TET configuration variables), thus improving the customizability of their suites.

Note that test cases built to this API do not require the TCC to execute. If the communication variables normally set by the TCC 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 is used. If the test case requires configuration variables or additional result codes, those communication variables should be set accordingly.

5.2.1 Test Case Structure and Management Functions

These functions support functionality used in initializing and cleaning up test cases, and in selecting invocable components and test purposes (described in chapter 7).

- **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 will not perform any error checking on the contents of the `tet_testlist` array. It is the test author's responsibility to ensure that the array is correctly specified.

  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 to the `tet_startup` and `tet_cleanup` function pointers and to 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.

5.2.2 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.

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

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

The implementation 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.

5.2.3 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 insures 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.

5.2.4 Canceling Test Purposes

These functions support functionality used to cancel test purposes.

- **Synopsis**

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

char *tet_reason(int test_no);
```

- **Description**

The function `tet_delete()` marks the test purpose specified by `test_no` as canceled 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 `test_no` is the sequence number (starting at 1) of the corresponding element in the `tet_testlist[]` array. If the requested `test_no` does not exist no action is taken. If the requested `test_no` 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 `test_no` 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.

The function `tet_reason()` returns a pointer to a string which contains the reason why the test purpose specified by `test_no` has been canceled. If this test purpose is not marked as canceled, or does not exist, a value of `(char *) NULL` is returned.

5.2.5 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.

5.2.6 Generating and Executing Processes

These functions support forking new child processes and synchronizing with them.

- **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 will create a new process which is a copy of the calling process and will modify the signal disposition in the newly created process such that any signals that were being caught in the parent process are now 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()` interface 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 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 `tcm_child.o` if that process is expected to generate journal lines or to synchronize with the parent. 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.

5.2.7 Executed Process Functions

These functions are to be used by processes executed through the `tet_exec` function.

- **Synopsis**

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

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 TET supplied child-process controller `tcmchild.o`. Prior to calling `tet_main()`, `tcmchild.o` will set the `tet_thistest()` variable to the number associated with the test purpose in the original test case. This value should not be changed by the executed process. Upon completion of the `tet_main()` routine the return value from this function will be returned to the process which called the `tet_fork()` function. The value returned from `tet_main()` should, in most cases, match one of the valid result values specified on the call to the `tet_fork()` function. The current context is preserved from the calling process and the current block is incremented by one before `tet_main()` is called. The `tet_pname` variable contains the process name as given in argv[0].

5.3 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 chapter 8 for an example of how to write a shell language based test suite.

Support for this language binding is provided through two shell language source files:

- `$TET_ROOT/lib/xpg3sh/tcm.sh` contains the support routines for the sequencing and control of invocable components and test purposes (the shell TCM).
- `$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 TCC to execute. With careful setup of the communication variables the test case can be invoked from the command line.

5.3.1 Test Case Structure and Management Functions

These functions support functionality used in initializing and cleaning up test cases, and in selecting invocable components and test purposes (described in chapter 8).

- **Synopsis**

```
iclist="blank separated list of invocable components"

ic1="blank separated list of test purposes"
ic2="blank separated list of test purposes"
etc...

tet_startup=start_up_function_name

tet_cleanup=clean_up_function_name

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` 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 will 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 shell variables for `tet_startup`, `tet_cleanup`, `iclist`, each of the `ic`ns, and any shell functions 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 those support routines.

The `tet_thistest` shell variable contains the name of the test purpose, as specified in the `icname` variable.

5.3.2 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.

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

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

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

5.3.3 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 command should be executed by any application which executes a background subshell and which wishes to write entries from both processes. The tet_setcontext command 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 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, then they 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. This result is output to the execution results file by the TCM upon test purpose completion. This insures 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.

5.3.4 Canceling Test Purposes

These functions support functionality used for canceling test purposes.

- **Synopsis**

```
tet_delete test_name reason

tet_reason test_name
```

- **Description**

The shell function tet_delete marks the test purpose specified by test_name 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 test_name matches the name which is used to call this test purpose. If the requested test_name does not match the name of a test purpose no action is taken. If the requested test_name 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 test_name 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 test_name 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.

5.3.5 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.

5.3.6 Generation and Execution of Processes

There is no explicit shell interface. 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.

5.3.7 Executed Process Support

Shell scripts which are executed by a test case written to this API should source (using the dot (.) shell built-in command) the shell API to include the necessary support routines. 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.

!!!!! DISCARD THIS PAGE !!!!!

6. Test Reporting/Journaling

6.1 Making Journal Entries

TET's API journaling facilities provide 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 TCC to reorder data that is produced from two or more concurrently executing processes started by a single test purpose). The TCC ensures that simultaneous execution of test cases are isolated from one another.

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

- 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 initialized 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 current block is initialized 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. TET guarantees atomicity of writes up to 512 bytes.

If a test purpose executes another process that is built to TET's API, and that executable is expected to generate journal messages, the test purpose must use TET's 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 TCM. The content of informational messages is limited only by the limit imposed upon the total length of a journal line. It is expected that test cases will use this mechanism to deliver special messages to the journal or for additional reporting sequences that can be analyzed 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 TET 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

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 TCC. The format of lines in these files is identical.

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 once for each activity performed by the TCC (each build, execute or clean-up of a test case is considered an individual activity).

- The invocable component number (ICnumber).
- 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. Its make up is implementation defined. Implementations are required to ensure that there are no repeated process identifiers during the life of a test purpose.

- The message block number (Block).

  This number is set to one at the start of each test purpose or new context, and is incremented once when 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 once for each message 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, e.g. 19910610 for 10th June 1991.

- A test case name (TestCase).

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

- A pathname (Pathname).

  The full pathname of a file.

- A mode (Mode).

  Legal modes are Build (0), Execute (1), and Clean-up (2).

- 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 by TET.

- An invocable component count (ICcount).

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

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 textual description of the message and its uses.

Test Case Controller Start `0|Version Time Date|who`

This message is generated by the TCC at the beginning of each TCC execution. The parameters include the version of the 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|IC list`

A test case start message is generated by the TCC for each test case executed in a scenario. The parameters are the sequence number of this TCC activity, the pathname 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|`

The TCM writes this message during its initialization. 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. The number of invocable components should be used by report treatment filters which audit test results to ensure that all ICs were correctly executed.

Configuration Start `20|Pathname Mode|`

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

Configuration Variable Setting `30||variable=value`

A configuration variable setting line is written to the journal by the TCC for each variable set for the current mode. The message area is of the form "variable=value".

Configuration End `40||`

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

Test Case Controller Message `50||`

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

Scenario Information `70||text`

The 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|`

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

Operator Abort `90|Time|`

The TCC generates an operator abort message when its execution is interrupted by the user.

Captured `100|Activity|text`

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

Build Start `110|Activity TestCase Time|`

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

Build End `130|Activity Status Time|`

A build end message is generated by the TCC upon completion of the build tool's 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|`

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|`

The TCM generates a result message for each test purpose executed. This result is usually supplied by the test purpose, although it is supplied by the TCM in the event a test purpose does not specify a result. Legal results are specified in "Result Code Definitions" in chapter 4. 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|`

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

Clean End `320|Activity Status Time|`

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

Invocable Component Start `400|Activity ICnumber TPcount Time|`

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|`

The TCC 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/API encounters a problem, it will be written to the journal as a TCM message. The parameters indicate the TCC activity counter. The message area contains a brief description of the problem.

Test Case Information `520|Activity TPnumber Context Block Sequence|text`

When a test purpose 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.

Test Case Controller End `900|Time|`

When the TCC completes processing of the scenario, it generates a TCC end message. This message indicates the time that execution completed.

6.3 The rpt Command

The report generator, `rpt`, opens a named journal file, reads its contents, and delivers it to a named report treatment filter.

Note: journal files are not modified by the report generator.

- **Synopsis**

  `rpt` *Journal_file Report_filter*

- **Description**

  The report generator takes data in *Journal_file* and passes it through a *Report_filter* for processing.

- **Operands**

  Journal_file A file generated by the `tcc` command. The format for this file is defined in the previous section.

  Report_filter An executable program that will interpret the *Journal_file* and generate a report. Report filters are provided by test suite authors. They can range from simple `awk` programs (as was used in chapter 2 on the demo journal file), to sophisticated text and graphics processors.

!!!!! DISCARD THIS PAGE !!!!!

7. Writing a C Language API Conforming Test Suite

7.1 Introduction

This chapter describes a sample test suite that conforms to TET's C language binding of the API. The source code for the test suite can be found in appendix B.

This sample test suite is designed to illustrate how a test suite can be structured under TET, 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 appendix B to see the code in its entirety.

7.2 Defining a Test Suite

Test suites reside in subdirectories of `$TET_ROOT`. As explained in chapter 2, 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`:

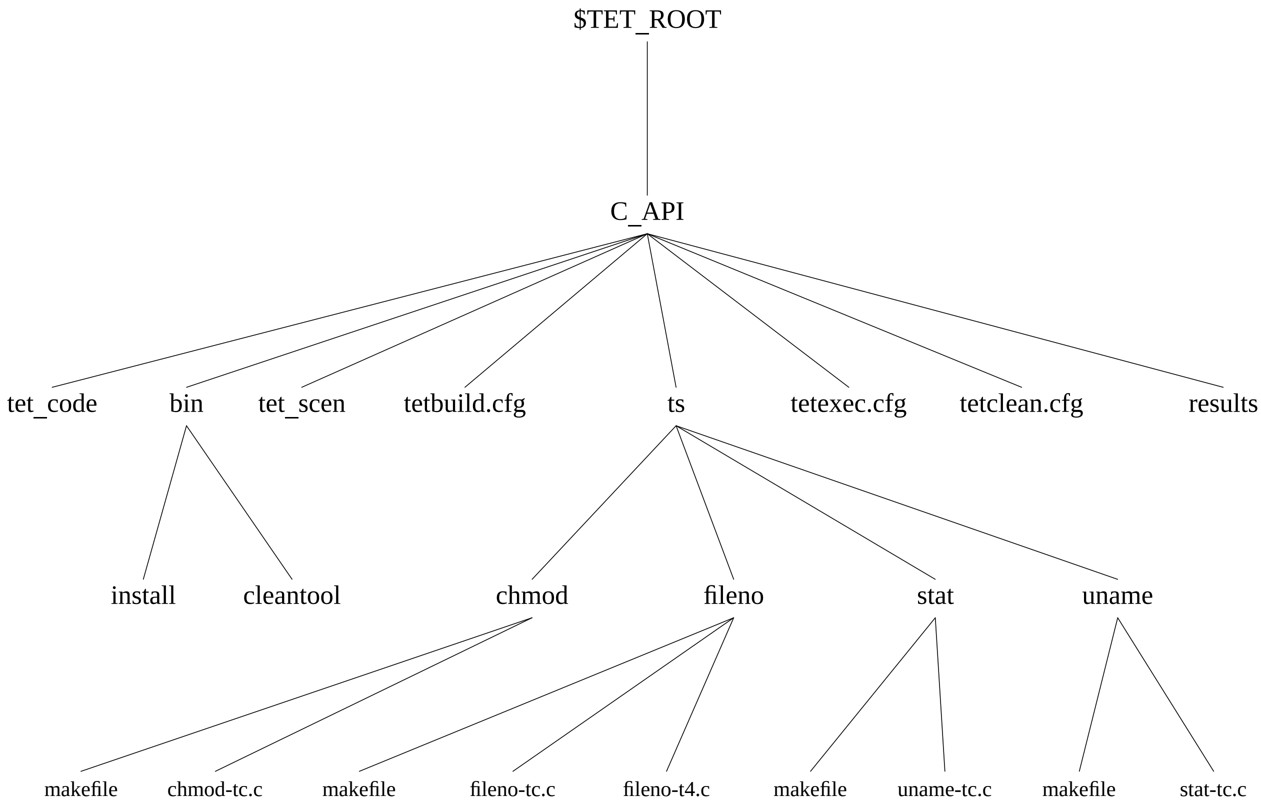

Figure 6. C-API Test Suite Files

The make-up of this test suite is similar to the `demo` test suite: an install script and cleantool 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`; and a results directory.

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 following figure shows the contents of the control file, `tet_scen`, for the `C-API` test suite.

```
#       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
```

Figure 7. The C-API Control File

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 quotation marks 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, their location is 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 cleantool 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 Initializing Test Cases

Every test case requires some minimum initialization of functions and variables. The `fileno-tc` test case provides a good illustration of how this initialization 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();

/* 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 */
```

After the `#include` statements, several functions are declared. As described in chapter 5, TET 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. 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 appendix B 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 of 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 three test purposes:

- `tp1`: successful chmod of a file, expecting a return code of 0.
- `tp2`: failed chmod of a non-existent file, expecting a return code of -1 and `errno` set to `ENOENT`.
- `tp3`: 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.

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.

7.5.1 Child Processes and Subprograms

Some test purposes require the creation of a child process or execution of a subprogram. The Toolkit provides three interfaces to facilitate this:

- `tet_fork()`

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

- `tet_exec()`

  an API function called by child processes to execute subprograms.

- `tet_main()`

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

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

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

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

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

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

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

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

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

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

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

```
    tet_result(TET_UNRESOLVED);
}
```

All the testing is done in the child, so the function `tp4()` simply calls `tet_fork()` and ignores the return value. If it needed to do any processing after the call to `tet_fork()`, it should check that the return value was one of the expected child exit codes before continuing. The arguments to `tet_fork()` are: 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 a null function pointer `TET_NULLFP`, defined in `tet_api.h`, is used); a timeout period in seconds; and, 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()`:

```
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;
}
```

7.6 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 TET'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.

!!!!! DISCARD THIS PAGE !!!!!

8. Writing a Shell Language API Conforming Test Suite

8.1 Introduction

This chapter describes a sample test suite that conforms to TET's shell language binding of the API. The source code for the test suite can be found in appendix C.

This sample test suite, like the one in chapter 7, is designed to illustrate how a test suite can be structured under TET, 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.

Small segments of code from the test suite appear in the following sections to help illustrate specific points. Refer to the appropriate section in appendix C to see the code in its entirety.

8.2 Defining a Test Suite

As explained in chapter 7, 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`:

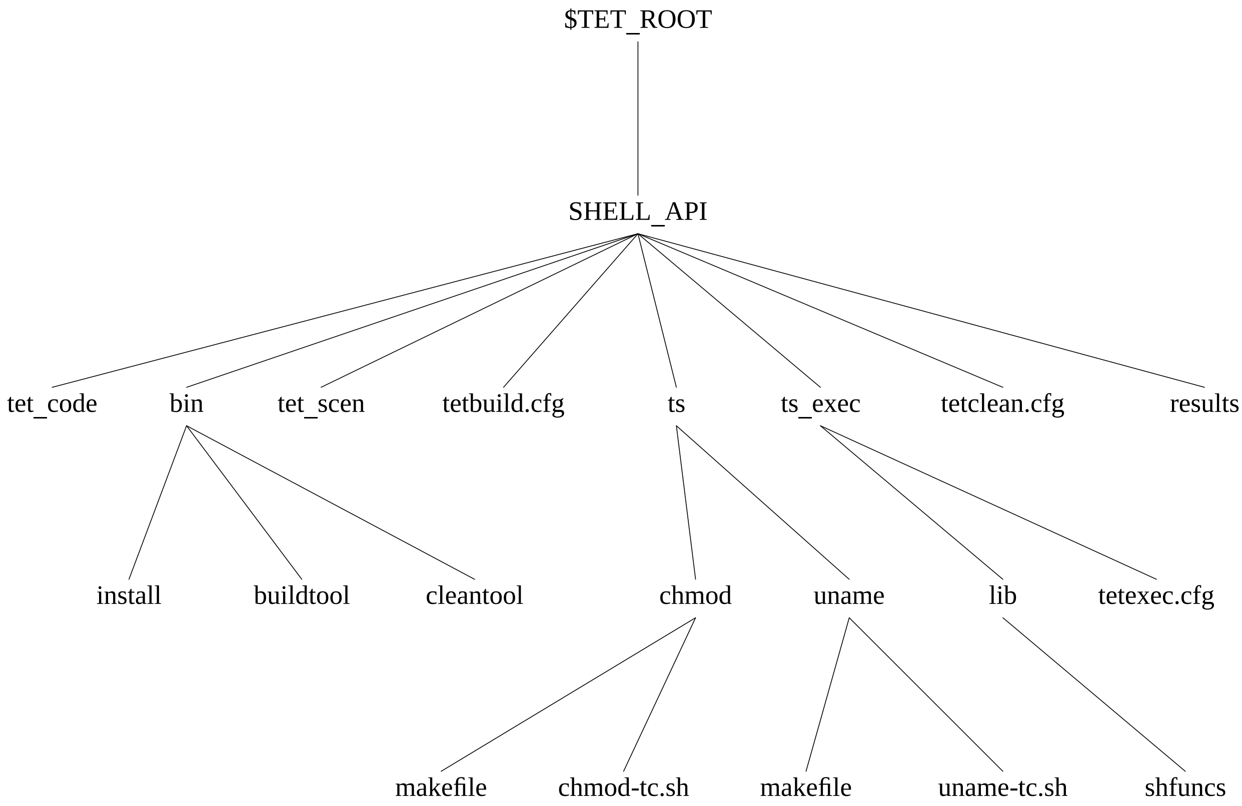

Figure 8. SHELL-API Test Suite Files

The make-up of this test suite is similar to the `C-API` test suite: an install script, buildtool and cleantool 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`; an alternate execution directory `ts_exec`; and a results directory.

The control file, `tet_scen`, is similar to the control file for the `C-API` test suite. See chapter 7 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/ts_exec` but in general it would be obtained from the user and could be located anywhere.

The buildtool 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 cleantool is used to remove the installed files from under the alternate execution directory. It works in the same way as the buildtool 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 minimize 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 convention of dot (.). The required TET shell API file, `tcm.sh` (which in turn sources in the other required TET shell API file `tetapi.sh`) is also sourced into each of the two test cases. It is important to note when these files are sourced in. Since the TET 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 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
}
```

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 following simple function, `check_exit`, was written to execute a command in a given argument (`$1`) , and to capture both standard error and standard output in separate files (in case one or both need to be checked), and to record the exit code in a variable called `$CODE`, and to output a message to the journal if an unexpected exit code is found.

```
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 Initializing Test Cases

Every test case requires some minimum initialization of functions and variables. The `uname-tc` test case provides a good illustration of how this initialization 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 chapter 5, TET 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 component (`ic`*N*) and the test purpose(s) that it contains. 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 three test purposes:

- `tp1`: successful chmod of a file with an expected exit code of 0.
- `tp2`: failed chmod of a non-existent file with an expected exit code of non-zero
- `tp3`: failed chmod through 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, so 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.

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.

8.6 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 TET’s `tet_cleanup` utility. The `cleanup` function is the most practical place to specify the removal of temporary files.

!!!!! DISCARD THIS PAGE !!!!!

A. Manual Pages

A.1 The rpt Command

The report generator, `rpt`, opens a named journal file, reads its contents, and delivers them to a named report treatment filter.

Note: journal files are not modified by the report generator.

The report generator supplies functionality as described in chapter 6.

- **Synopsis**

  `rpt` *Journal_file Report_filter*

- **Description**

  The report generator takes data in *Journal_file* and passes it through a *Report_filter* for processing.

- **Operands**

  Journal_file A file generated by the `tcc` command. The format for this file is defined in the "Journal Files" section of chapter 6.

  Report_filter An executable program that will interpret the *Journal_file* and generate a report. Report filters are provided by test suite authors. They can range from simple `awk` programs (as was used in chapter 2 on the demo journal file), to sophisticated text and graphics processors.

A.2 The tet_ins Command

The test suite installer, `tet_ins`, is supplied with the toolkit to assist with test suite installation. Its syntax is described below.

- **Synopsis**

  `tet_ins` *Suite_name*

- **Description**

  The `tet_ins` utility causes the test suite provided installation tool to be executed. It expects to find a file called `install` in `$TET_ROOT/`*Suite_name*`/bin`. After execution, all TET required data files are checked to ensure that they are syntactically reasonable. Note that this utility does not attempt to move the test suite into the file system. It is the responsibility of the user to ensure that the test suite has been loaded under the main TET directory before executing `tet_ins`.

- **Operands**

  Suite_name names the test suite to be installed.

A.3 The tcc Command

The `tcc` command provides support for building, executing, and cleaning up test scenarios. This interface is appropriate for systems which support a command shell as defined in the X/Open Portability Guide Issue 3 Volume 2 — `sh`.

- **Synopsis**

  `tcc` **[** `-b` **] [** `-e` **] [** `-c` **] [** `-a` *Alt_exec_dir* **] [** `-f` *Clean_cfg_file* **]**
  [`-g` *Build_cfg_file* **] [** `-i` *Int_results_dir* **] [** `-j` *Journal_file* **]**
  [`-l` *Scenario_line* **] [** `-p` **] [** `-n` *Search_string* **] [** `-s` *Scenario_file* **]**
  [`-t` *Timeout* **] [** `-v` *Variable=value* **] [** `-x` *Exec_config_file* **]**
  [`-y` *Search_string* **]**
  [*Test_suite* **[** *Scenario* **]]**

  `tcc -m` *Code_list* **[** *options* **]** *Old_journal_file* **[** *Test_suite* **[** *Scenario* **]]**

  `tcc -r` *Code_list* **[** *options* **]** *Old_journal_file* **[** *Test_suite* **[** *Scenario* **]]**

- **Description** The Test Case Controller (TCC) has three mode selection options, which may be used singly or in any combination. All of the other options modify the behavior of the TCC in one or more modes. Each mode (with optionally modified behavior) is applied to the test cases and invocable components selected for processing.

  Unless otherwise specified, journal information and saved intermediate result files are placed below a sequentially numbered directory under the `results` directory of the test suite selected. The TCC writes the name of the `journal` to standard output on startup.

- **Options**
  Mode Selection

  `-b` Enable build mode.

  `-c` Enable clean mode.

  `-e` Enable execute mode.

  Major Options

  These two options cannot be used together.

  `-m` *Code_list* Resume from a specified point in a previous run. The argument *Code_list* specifies a comma-separated list of result code names. Upon encountering one of the specified result code names in the *Old_journal_file*, the associated invocable component and all subsequent invocable components in the scenario are built, executed and cleaned up, as appropriate.

  `-r` *Code_list* Re-run specified invocable components from a previous run. The argument *Code_list* specifies a comma-separated list of result code names which, when encountered in the *Old_journal_file*, cause the associated invocable component to be built, executed, and cleaned up, as appropriate.

  Other Options

  `-a` *Alt_exec_dir* The argument *Alt_exec_dir* specifies the pathname of the root of a directory hierarchy in which executable test cases should be placed (by the build tool in build mode), executed from (in execute mode), and cleaned from (by the clean tool in clean mode). If unspecified, the alternate execution directory is taken from the environment variable `TET_EXECUTE`, if set, otherwise the default behavior is to execute test cases from the test suite hierarchy.

`–f` *Clean_cfg_file* — The argument *Clean_cfg_file* specifies the pathname of the file of configuration variables to use in clean mode. If unspecified this value defaults to `tetclean.cfg` in the alternate execution directory, or failing that, `tetclean.cfg` in the root directory of the test suite selected.

`–g` *Build_cfg_file* — The argument *Build_cfg_file* specifies the pathname of the file of configuration variables to use in build mode. If unspecified this value defaults to `tetbuild.cfg` in the alternate execution directory, or failing that, `tetbuild.cfg` in the root directory of the test suite selected.

`–i` *Int_results_dir* — The argument *Int_results_dir* specifies the pathname of the root of a directory hierarchy to store intermediate result files (matching the configuration variable `TET_SAVE_FILES`). If unspecified the intermediate results directory defaults to a sequentially numbered directory under the `results` directory of the test suite.

`–j` *Journal_file* — The argument *Journal_file* specifies the pathname of the file in which to record journal information. If unspecified, the journal file is named `journal` in the root of the intermediate results hierarchy (see **-i** option).

`–l` *Scenario_file* — The argument *Scenario_line* specifies a single line that is be treated as though it appeared in a scenario file. Multiple lines may be specified by repeating this option. Any scenario line, other than an invocable component execution line, should be quoted to the shell. If a scenario file is also specified (via the **-s** option) the lines specified with **-l** options are processed before the scenario file. If no **-s** option is used, then no scenario file is processed.

`–p` — Enable progress reporting. As each build, execute or clean is started a line is printed to standard output giving the time, the mode, and the scenario line.

`–n` *Search_string* — The argument *Search_string* specifies a portion of text to search for in any scenario invocable component execution line. Multiple strings may be specified by repeating this option. If a string is found then that line is not executed. This option has a higher precedence than the **-y** option. I.e. lines which match a **-y** search string and a **-n** search string are not processed.

`–s` *Scenario_file* — The argument *Scenario_file* specifies the pathname of a scenario file. If no **-s** or **-l** options are specified, the default is to use a scenario file called `tet_scen` in the root directory of the test suite selected.

`–t` *Timeout* — The argument *Timeout* specifies the number of seconds to allow a single build, execute, or clean of a test case to continue. If unspecified the TCC imposes no time limit.

`-v` *Variable=value* The argument *Variable=value* is used to set the configuration variable *Variable* to *value*. Multiple variables may be manipulated by repeating this option with different settings. If *value* contains white space or other characters special to the shell, it must be quoted.

`-x` *Exec_config_file* The argument *Exec_config_file* specifies the pathname of the file of configuration variables to use in execute mode. If unspecified this value defaults to `tetexec.cfg` in the alternate execution directory, or failing that, `tetexec.cfg` in the root directory of the test suite selected.

`-y` *Search_string* The argument *Search_string* specifies a portion of text to search for in any scenario invocable component execution line. Multiple strings may be specified by repeating this option. A string must be found in order for the line to be processed. This option has a lower precedence than the **-n** option. I.e. lines which match a **-y** search string and a **-n** search string are not processed.

- **Operands**

Old_journal_file This operand must be supplied when either the resume or re-run option is specified. It indicates the pathname of the journal file that is to be searched for matching results codes.

Test_suite This operand indicates the suite to which the scenario must be applied. The suite name is used by the TCC to determine the test suite root directory name. If none is given on the command line, a default test suite may be deduced by the TCC according to the following rules:

1. If the current directory lies under the TET_ROOT directory hierarchy, the test suite is the component of the current directory's pathname which lies immediately below the TET_ROOT. E.g. if the TET_ROOT is `/usr/TET` and the current directory is `/usr/TET/suite1/results`, then the default test suite is `suite1`.

2. If the current directory lies outside of the TET_ROOT directory hierarchy, then no default can be deduced.

Scenario This operand indicates the Test Scenario that is to be processed by the Test Case Controller. If none is given on the command line, the default scenario is `all`. If a scenario is specified but no scenario file is in use (see **-l** and **-s** options), the TCC gives an error message and exits.

- **Environment Variables**

TET_ROOT
The pathname of the TET root directory. When set, this variable has precedence over the value compiled into the TCC.

TET_EXECUTE
The pathname of the root of the execution directory hierarchy. If set this value is used as the alternate execution directory, unless overridden by use of the **-a** option.

TET_TMP_DIR
The location for the temporary directories created by the TCC (see **Files** below). Performance problems due to networked file systems may be helped by setting this variable to a directory on a local file system.

- **Files**

By default the TCC creates a temporary directory in the TET root directory called **tet_tmp_dir**. Each invocation of the TCC creates a unique sub-directory below it. In the sub-directory are placed files that are needed during TCC execution. After completion the sub-directory is removed. However should the TCC terminate abnormally there is a possibility that a sub-directory will remain. Such sub-directories should be removed to avoid wasting space.

!!!!! DISCARD THIS PAGE !!!!!

B. C Language API Test Suite Source Files

B.1 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.2 install

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

B.3 cleantool

```
exec make clean
```

B.4 tet_scen

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

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

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

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

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

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

# EOF
```

B.5 tetbuild.cfg

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

B.6 tetexec.cfg

```
TET_OUTPUT_CAPTURE=False

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

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

B.7 tetclean.cfg

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

B.8 makefile for chmod-tc.c

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

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

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

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

B.9 chmod-tc.c

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

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

#include <tet_api.h>

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

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


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

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

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

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

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

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

    tet_infoline("SUCCESSFUL CHMOD OF FILE");

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

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

    /* check mode was changed correctly */

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

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

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

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

    /* ensure file does not exist */

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

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

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

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

        if (err != ENOENT)
        {
```

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

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

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

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

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

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

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

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

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

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

B.10 makefile for fileno-tc.c

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

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

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

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

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

B.11 fileno-tc.c

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

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

#include <tet_api.h>

extern char **environ;

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

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

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

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

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

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

    /* open stream to test file */

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

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

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

```
    }

    if (fstat(fileno(fp), &buf2) == -1)
    {
        (void) sprintf(msg, "fstat(fileno(fp), buf2) failed - errno %d",
            errno);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else if (buf1.st_ino != buf2.st_ino || buf1.st_dev != buf2.st_dev)
    {
        tet_infoline("fileno(fp) does not refer to same file as fp");
        (void) sprintf(msg, "st_dev, st_ino of file: 0x%lx, %ld",
            (long)buf1.st_dev, (long)buf1.st_ino);
        tet_infoline(msg);
        (void) sprintf(msg, "st_dev, st_ino of fileno(fp): 0x%lx, %ld",
            (long)buf2.st_dev, (long)buf2.st_ino);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else
        tet_result(TET_PASS);

    (void) fclose(fp);
}

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

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

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

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

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

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

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

static void
tp3()        /* on entry to main(), stdin is readable, stdout and stderr
                are writable */
{
    int flags, fail = 0;

    tet_infoline("ON ENTRY TO MAIN, STDIN IS READABLE, STDOUT AND STDERR ARE WRITABLE");
```

```
    /* this code relies on the fact that the TCM does not interfere
       with these streams */

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B.12 fileno-t4.c

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

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

#include <tet_api.h>

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

/* ARGSUSED */

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

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

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

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

    return 0;
}
```

B.13 makefile for stat-tc.c

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

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

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

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

B.14 stat-tc.c

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

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

#include <tet_api.h>

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

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


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

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

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

        /* Prevent tests which use this file from executing */
        tet_delete(1, reason);
        tet_delete(7, reason);
    }
    else
        (void) close(fd);
}

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

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

static void
```

```
tp1()         /* successful stat of plain file: return 0 */
{
    int ret, err;

    tet_infoline("SUCCESSFUL STAT OF PLAIN FILE");

    /* stat file created in startup function and check mode indicates
       a plain file */

    errno = 0;
    if ((ret=stat(tfile, &buf)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
            tfile, ret);
        tet_infoline(msg);
        if (err != 0)
        {
            (void) sprintf(msg, "errno was set to %d", err);
            tet_infoline(msg);
        }
        tet_result(TET_FAIL);
    }
    else if (!S_ISREG(buf.st_mode))
    {
        tet_infoline("S_ISREG(st_mode) was not true for plain file");
        (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else
        tet_result(TET_PASS);
}

static void
tp2()         /* successful stat of directory: return 0 */
{
    int ret, err;
    char *tdir = "stat.d";

    tet_infoline("SUCCESSFUL STAT OF DIRECTORY");

    /* create a test directory */

    if (mkdir(tdir, S_IRWXU) == -1)
    {
        (void) sprintf(msg,
            "mkdir(\"%s\", S_IRWXU) failed in startup - errno %d",
            tdir, errno);
        tet_infoline(msg);
        tet_result(TET_UNRESOLVED);
        return;
    }

    /* stat the directory and check mode indicates a directory */

    errno = 0;
    if ((ret=stat(tdir, &buf)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
            tdir, ret);
        tet_infoline(msg);
        if (err != 0)
        {
            (void) sprintf(msg, "errno was set to %d", err);
            tet_infoline(msg);
        }
        tet_result(TET_FAIL);
    }
    else if (!S_ISDIR(buf.st_mode))
    {
```

```
        tet_infoline("S_ISDIR(st_mode) was not true for directory");
        (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else
        tet_result(TET_PASS);

    (void) rmdir(tdir);
}

static void
tp3()         /* successful stat of FIFO file: return 0 */
{
    int ret, err;
    char *tfifo = "stat.p";

    tet_infoline("SUCCESSFUL STAT OF FIFO");

    /* create a test FIFO */

    if (mkfifo(tfifo, S_IRWXU) == -1)
    {
        (void) sprintf(msg,
            "mkfifo(\"%s\", S_IRWXU) failed in startup - errno %d",
            tfifo, errno);
        tet_infoline(msg);
        tet_result(TET_UNRESOLVED);
        return;
    }

    /* stat the FIFO and check mode indicates a FIFO */

    errno = 0;
    if ((ret=stat(tfifo, &buf)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
            tfifo, ret);
        tet_infoline(msg);
        if (err != 0)
        {
            (void) sprintf(msg, "errno was set to %d", err);
            tet_infoline(msg);
        }
        tet_result(TET_FAIL);
    }
    else if (!S_ISFIFO(buf.st_mode))
    {
        tet_infoline("S_ISFIFO(st_mode) was not true for FIFO file");
        (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else
        tet_result(TET_PASS);

    (void) unlink(tfifo);
}

static void
tp4()         /* successful stat of character device file: return 0 */
{
    int ret, err;
    char *chardev;

    tet_infoline("SUCCESSFUL STAT OF CHARACTER DEVICE FILE");

    /* obtain device name from execution configuration parameter */

    chardev = tet_getvar("CHARDEV");
    if (chardev == NULL || *chardev == '\0')
```

```
    {
        tet_infoline("parameter CHARDEV is not set");
        tet_result(TET_UNRESOLVED);
        return;
    }

    /* check if parameter indicates character devices are not supported */

    if (strcmp(chardev, "unsup") == 0)
    {
        tet_infoline("parameter CHARDEV is set to \"unsup\"");
        tet_result(TET_UNSUPPORTED);
        return;
    }

    /* stat the device and check mode indicates a character device */

    errno = 0;
    if ((ret=stat(chardev, &buf)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
            chardev, ret);
        tet_infoline(msg);
        if (err != 0)
        {
            (void) sprintf(msg, "errno was set to %d", err);
            tet_infoline(msg);
        }
        tet_result(TET_FAIL);
    }
    else if (!S_ISCHR(buf.st_mode))
    {
        (void) sprintf(msg, "S_ISCHR(st_mode) was not true for \"%s\"",
            chardev);
        tet_infoline(msg);
        (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else
        tet_result(TET_PASS);
}

static void
tp5()         /* successful stat of block device file: return 0 */
{
    int ret, err;
    char *blockdev;

    tet_infoline("SUCCESSFUL STAT OF BLOCK DEVICE FILE");

    /* obtain device name from execution configuration parameter */

    blockdev = tet_getvar("BLOCKDEV");
    if (blockdev == NULL || *blockdev == '\0')
    {
        tet_infoline("parameter BLOCKDEV is not set");
        tet_result(TET_UNRESOLVED);
        return;
    }

    /* check if parameter indicates block devices are not supported */

    if (strcmp(blockdev, "unsup") == 0)
    {
        tet_infoline("parameter BLOCKDEV is set to \"unsup\"");
        tet_result(TET_UNSUPPORTED);
        return;
    }

    /* stat the device and check mode indicates a block device */
```

```
    errno = 0;
    if ((ret=stat(blockdev, &buf)) != 0)
    {
        err = errno;
        (void) sprintf(msg, "stat(\"%s\", buf) returned %d, expected 0",
            blockdev, ret);
        tet_infoline(msg);
        if (err != 0)
        {
            (void) sprintf(msg, "errno was set to %d", err);
            tet_infoline(msg);
        }
        tet_result(TET_FAIL);
    }
    else if (!S_ISBLK(buf.st_mode))
    {
        (void) sprintf(msg, "S_ISBLK(st_mode) was not true for \"%s\"",
            blockdev);
        tet_infoline(msg);
        (void) sprintf(msg, "st_mode = 0%lo", (long)buf.st_mode);
        tet_infoline(msg);
        tet_result(TET_FAIL);
    }
    else
        tet_result(TET_PASS);
}

static void
tp6()       /* stat of non-existent file: return -1, errno ENOENT */
{
    int ret, err;

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

    /* ensure file does not exist */

    if (stat("stat.6", &buf) != -1 && unlink("stat.6") == -1)
    {
        tet_infoline("could not unlink stat.6");
        tet_result(TET_UNRESOLVED);
        return;
    }

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

    errno = 0;
    ret = stat("stat.6", &buf);

    if (ret != -1 || errno != ENOENT)
    {
        err = errno;
        if (ret != -1)
        {
            (void) sprintf(msg,
                "stat(\"stat.6\", 0) returned %d, expected -1", ret);
            tet_infoline(msg);
        }

        if (err != ENOENT)
        {
            (void) sprintf(msg,
                "stat(\"stat.6\", 0) set errno to %d, expected %d (ENOENT)",
                err, ENOENT);
            tet_infoline(msg);
        }

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

```
static void
tp7()       /* non-directory path component: return -1, errno ENOTDIR */
{
    int ret, err;

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

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

    errno = 0;
    ret = stat(tndir, &buf);

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

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

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

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

B.15 makefile for uname-tc.c

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

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

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

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

B.16 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);
    }
}
```

!!!!! DISCARD THIS PAGE !!!!!

C. Shell API Test Suite Source Files

C.1 tet_code

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

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

C.2 install

```
echo Installing SHELL-API test suite

cd SHELL-API || exit 1

# create alternate execution directory hierarchy
find ts -type d -print |
while read d
do
    if test ! -d ts_exec/"$d"
    then mkdir ts_exec/"$d"
    fi
done
```

C.3 buildtool

```
:
# Check TET_EXECUTE is set
if [ -z "$TET_EXECUTE" ]
then
	echo >&2 "No alternate execution directory supplied to buildtool"
	exit 1
fi

# Set TET_EXECUTE on command line to override default value in makefile
exec make TET_EXECUTE="$TET_EXECUTE"
```

C.4 cleantool

```
:
# Check TET_EXECUTE is set
if [ -z "$TET_EXECUTE" ]
then
	echo >&2 "No alternate execution directory supplied to cleantool"
	exit 1
fi

# Set TET_EXECUTE on command line to override default value in makefile
exec make TET_EXECUTE="$TET_EXECUTE" clean
```

C.5 tet_scen

```
#       chmod, uname test suite.
all
        "Starting Full Test Suite"
        /ts/chmod/chmod-tc
        /ts/uname/uname-tc
        "Completed Full Test Suite"
chmod
        "Starting chmod Test Case"
        /ts/chmod/chmod-tc
        "Finished chmod Test Case"
uname
        "Starting uname Test Case"
        /ts/uname/uname-tc
        "Finished uname Test Case"
# EOF
```

C.6 tetbuild.cfg

```
TET_OUTPUT_CAPTURE=True
TET_BUILD_TOOL=buildtool
```

C.7 tetexec.cfg

```
TET_OUTPUT_CAPTURE=False
TET_EXEC_IN_PLACE=True

# highest signal number plus 1
TET_NSIG=22
```

C.8 tetclean.cfg

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

C.9 common functions used in shell API test suite

```
# shfuncs : test suite common shell functions

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

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

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

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

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

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

    case $1 in
    "")
        if [ ! -s out.stderr ]
        then
            tet_infoline "Expected output to stderr, but none written"
            FAIL=Y
        fi
```

```
        ;;
    *)
        expfile="$1"
        OK=Y
        exec 4<&0 0< "$expfile" 3< out.stderr
        while read expline
        do
            if read line <&3
            then
                if expr "$line" : "$expline" > /dev/null
                then
                    :
                else
                    OK=N
                    break
                fi
            else
                OK=N
            fi
        done
        exec 0<&4 3<&- 4<&-
        if [ $OK = N ]
        then
            tet_infoline "Incorrect output written to stderr, as shown below"
            infofile "$expfile" "expected stderr:"
            infofile out.stderr "received stderr:"
            FAIL=Y
        fi
        ;;
    esac
}

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

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

C.10 makefile for chmod-tc.sh

```
TET_EXECUTE = ../../ts_exec
INSTALL_DIR = $(TET_EXECUTE)/ts/chmod

$(INSTALL_DIR)/chmod-tc: chmod-tc.sh clean
        cp chmod-tc.sh $@
        chmod 755 $@

clean:
        rm -f $(INSTALL_DIR)/chmod-tc
```

C.11 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
```

C.12 makefile for uname-tc.sh

```
TET_EXECUTE = ../../ts_exec
INSTALL_DIR = $(TET_EXECUTE)/ts/uname

$(INSTALL_DIR)/uname-tc: uname-tc.sh clean
        cp uname-tc.sh $@
        chmod 755 $@

clean:
        rm -f $(INSTALL_DIR)/uname-tc
```

C.13 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