Test Environment Toolkit

TETware Design Specification
Revision 1.0
TET3-SPEC-1.0

Released: 30 April 1996

X/Open Company Limited

This document is produced by UniSoft Ltd. at:

150 Minories
LONDON
EC3N 1LS
United Kingdom

1. Foreword

1.1 Introduction

X/Open has asked UniSoft to prepare a Design Specification for enhancements to the TET. The name of the revised TET will be TETware.

This document is being provided for review purposes, in order to encourage discussion among X/Open member companies and other organisations considered important by X/Open. It is not intended to include this document in a future TETware release.

1.2 Background

During evolution of the TET, two distinct development threads have emerged. One – dTET2 – provides support for local, remote and distributed testing through a client/server architecture. The other – ETET – while supporting only local testing, contains a number of features which have become popular among certain members of the TET user community.

Both of these developments are derived from TET release 1.10. This is the latest release of the base TET and was made by X/Open during 1992.

It is proposed to re-integrate these two threads into a single standard product with the intention of satisfying the needs of users of both of the existing toolkits.

1.3 Project goals

The goals of this project are:

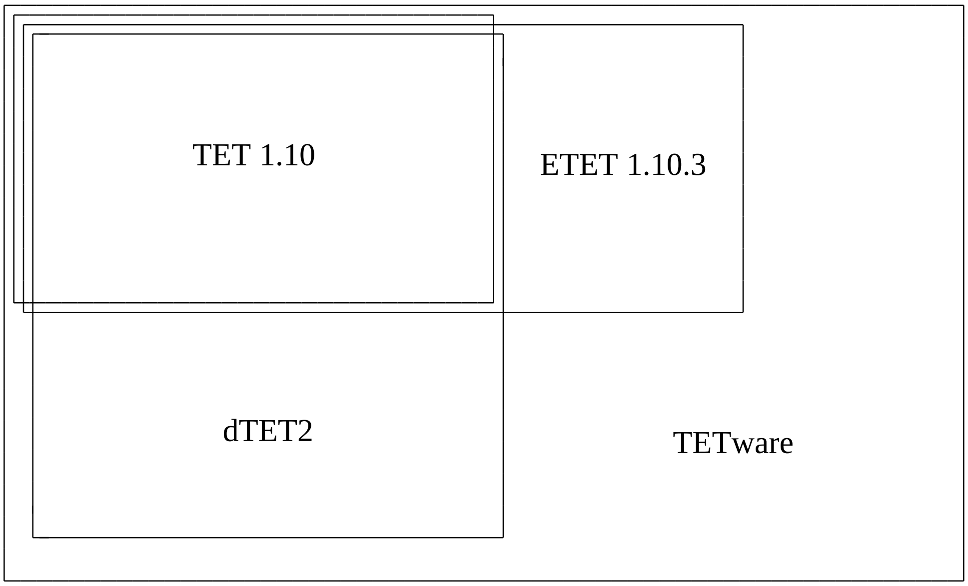
1. To create a single implementation of the TET which is a superset of the features within the existing implementations.
2. To add a number of new features to this revised implementation and to improve, integrate, or rewrite certain features which have been implemented by third parties.
3. To ensure that the revised software provides backward compatibility with existing TET implementations.
4. To ensure that the revised software is portable to a wide range of popular platforms outside the traditional sphere of Unix-like and POSIX-conforming environments.

In addition to the functional requirements presented above, X/Open has the following goals:

1. That support be available to TETware users.
2. That TETware should be suitable for use by the X/Open test suites which are being specified (in particular the ODBC/CLI test suite) and the existing X/Open test suites.
3. That TETware should be usable in conjunction with test cases generated by ADL (which is currently being developed by X/Open).

Given the wide ranging scope of these goals, X/Open and UniSoft have agreed that the first step should be to develop a design specification for TETware. Feedback will be sought from X/Open and other organisations which are considered important by X/Open. This will ensure that TETware meets all the requirements set by X/Open and the other organisations.

The following diagram illustrates the relationship between TETware and existing TET implementations:



Note that this diagram is not to scale.

1.4 Conventions used in this document

The following typographic conventions are used throughout this document:

- `Courier font` is used for function, variable and program names, literals and file names.
- The names of variable values are presented in *italic font*.
- **Bold font** is used for headings and for emphasis.
- In a syntax definition, an optional element is enclosed in square brackets [ ].
- An ellipsis . . . indicates that the previous element may be repeated as required.

1.5 Related documents

The following documents contain additional information about the existing TET, dTET2 and ETET implementations:

- Test Environment Toolkit: Architectural, Functional and Interface Specification Revision 4.5

- Distributed Test Environment Toolkit Version 2: Architectural, Functional and Interface Specification Revision 5.2
- Distributed Test Environment Toolkit Version 2: dTET2 Programmers Guide Revision 1.3
- Distributed Test Environment Toolkit Version 2: dTET2 Installation and User Guide Revision 1.3
- Extended Test Environment Toolkit: Programmers Guide Revision 1.10.3

For ease of reference, portions of some of these documents are reproduced as appendices to this document.

1.6 Terminology

Certain terms described here are used in this document to describe the different types of test case that may be executed by TETware.

A **local test case** is one that executes on the local system; that is, the system on which the Test Case Controller `tcc` is executed.

A **remote test case** is one that executes on a system other than the one on which `tcc` is executed. `tcc` collects the test case's execution results file output from the remote system and includes it in the journal file on the local system. Although it is possible for several remote test cases to execute concurrently on different remote systems, the test harness does not provide for interaction between remote test cases.

A **distributed test case** is one that has several parts; these parts execute concurrently on different systems. When a distributed test case is being executed, the test harness ensures that each test purpose part starts at the same time on each system. Thus each part of a particular distributed test case must always contain identical number of invocable components and test purposes, even if this means that some of the test purpose parts do nothing. It is likely that parts of a distributed test purpose will interact with each other in some way during the course of their execution. In particular, the test harness provides a means by which the different parts of a test purpose may synchronise with each other. Each test purpose part submits a result which indicates the success or failure of that part of the test purpose. The test harness arbitrates between the results submitted by the parts of the test purpose that are executing on each system and enters a single consolidated result in the journal file.

A more complete description of some of the terms and component names used when describing dTET2 is presented in the chapter entitled ‘‘Overview of the Distributed Test Environment Toolkit Version 2’’ in the dTET2 Installation and User Guide.

A diagram showing how the different dTET2 components relate to each other is presented in appendix E, ‘‘dTET2 architecture’’.

2. TETware architecture options

2.1 Lightweight TETware

dTET2 uses a client/server architecture in order to process remote and distributed test cases. This requires a great deal of additional code (including the network code) to be included in dTET2 processes which is not used when processing local test cases.

In order to enable TETware to be ported to a wide range of systems, an option will be provided which will exclude the remote and distributed test case processing facilities from TETware. This will take the form of a compile-time option. Use of this option will affect the way in which both `tcc` and the API library is built. In particular, certain scenario directives and API functions will be unavailable or have no effect when this option is used.

The version of TETware that will result when this option is used will be known as **TETware-Lite**. When this option is used, there will be no requirement for the target system to support networking. The version of TETware that will result when this option is not used will be referred to in this document as **fully-featured TETware** in a context where it is necessary to distinguish between the two versions of TETware.

TETware-Lite will not make calls to any network functions. On Unix-like systems, TETware-Lite will only use system interfaces specified in POSIX.1.

2.2 Master and slave systems

The architectural model presented in the current dTET2 specification describes test case processing in terms of a master system and zero or more slave systems. The master system is identified by a system ID of zero and slave systems by a non-zero system ID.

dTET2 provides facilities to execute test cases in several ways as follows:

- Execution of non-distributed test cases on the local (or master) system (that is, local test cases).
- Execution of non-distributed test cases on one or more remote (or slave) system (that is, remote test cases).
- Execution of distributed test cases with the parts of each test case executing simultaneously on both the local (or master) system and one or more remote (or slave) systems.

The dTET2 specification differentiates between the master and slave systems in several ways as follows:

i. The master system is the one on which `tcc` is invoked.

ii. The difference between a remote and a distributed test case is determined by whether or not the master system is specified in the `remote` directive in the scenario file.[1]

iii. The master system is included implicitly in a user synchronisation request.

The differentiation between master and slave systems will be eliminated in TETware, resulting in a symmetrical relationship between all the systems which participate in a distributed test case. Advantages of this enhancement are:

i. The processing logic in `tcc` is greatly simplified, since there is no need to treat different systems in different ways when processing test cases.

ii. When a distributed test case is to be executed on systems where facilities or resources are insufficient to support the execution of a fully-featured `tcc`, it will be possible to control test case execution from a (more feature-rich) system which is not participating in the test case.[2]

The implication that this enhancement will have on the design of `tcc` is expanded in the section entitled ‘‘Local and remote procedures’’ which appears in chapter 3.

1. When the master system appears in the system list which accompanies the `remote` directive in the scenario file, this indicates that distributed test case parts are to be executed on each of the named systems. When the master system does not appear in the `remote` directive’s system list, this indicates that remote test cases are to be executed in parallel on the named systems.

2. When this is done it will be possible to execute parts of a distributed test case on (say) one or more PC-type systems under the control of a `tcc` running on (say) a Unix-like system. In this arrangement, the Unix-like system hosts the scenario and configuration files and provides storage for journal files, together with test case output files transferred from the PC-type systems using `TET_OUTPUT_CAPTURE` mode or the `TET_TRANSFER_SAVE_FILES` mechanism.

3. The Test Case Controller `tcc`

3.1 Introduction

The requirements imposed by remote and distributed testing have made demands of the existing `tcc` structure far beyond those envisaged in the original `tcc` design. Therefore it will be necessary to undertake a thorough overhaul of `tcc` in order to enable the requirements of dTET2 and ETET to coexist and, at the same time, provide a sound basis for future enhancement.

3.2 Process structure

When processing remote, distributed and parallel test cases, the dTET2 `tcc` forks a child process to supervise the execution of each test case element. A disadvantage of this approach is that it relies on behaviour of the Unix `fork()` system call which is not always available on non-Unix systems.

The TETware `tcc` will not use child processes in this way, but will instead implement a flat process structure where the progress of all test cases is monitored in a single `tcc` process. Use of this process control implementation will assist in porting `tcc` to non-Unix systems.

3.3 Local and remote procedures

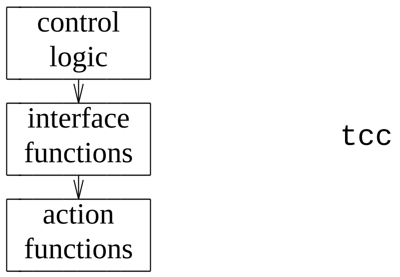
In the existing dTET2 architecture, `tcc` performs its processing on the master system directly, while processing on slave systems is performed by a server process called `tccd`. Once `tcc` has established communication with `tccd` on each slave system, each `tccd` performs such actions on its slave system as may be required, on receipt of a request from `tcc` on the master system. When `tcc` wishes to send such a request, it does so by calling a server interface function. Descriptions of server interface functions that are implemented in dTET2 are presented in appendix H, ‘‘Server interface functions’’.

When `tcc` is invoked, processing takes place according to which mode(s) are specified on the command line. This processing consists of a series of actions – reading configuration variables, building, executing or cleaning test cases, gathering results and so on.

In TETware, each of these operations is performed by a particular function – an **action** function – which will reside in a library. When the `tcc` processing logic wishes to perform one of these actions, it will do so by calling an interface function. Each of these interface functions will contain two code sections; one section will make a direct call to the action function, while the other will make a call to the equivalent server interface function. Only one of these sections will be selected by means of conditional compilation, depending on whether TETware or TETware-Lite is to be built.

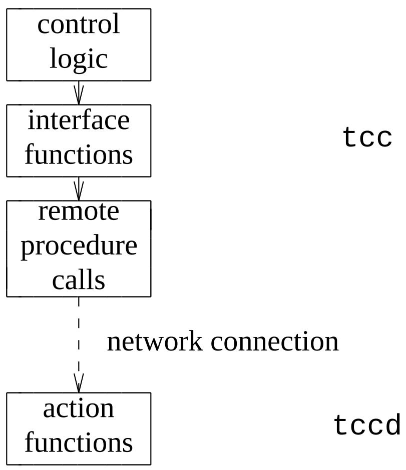
When TETware-Lite is built, `tcc` will be linked with the library containing the action functions; thus the processing logic and the action functions will be contained in a single process. As a result, TETware-Lite will only support test case execution on a single system (as has been noted earlier).

In TETware-Lite the flow of control through `tcc` is as follows:



When fully-featured TETware is built, `tcc` will not be linked with the action function library. Instead, this library will be used when building `tccd` and action functions will be called as a result of requests received from `tcc`. As a result, consistency of behaviour between the two TETware versions will be assured. In addition, symmetry of operation between all participating systems will be achieved because `tcc` actions will be performed by `tccd` on all systems, both local and remote. A further benefit is that it will be possible to run `tcc` on a system that is not participating in the test case run, thus reducing the impact of the test harness software on the system(s) under test.

In fully-featured TETware the flow of control through `tcc` and `tccd` is as follows:



3.4 TCM/API interface

The interface between `tcc` and the TCM/API is defined in the base TET specification. This interface is supported by TETware-Lite.

However, this interface is insufficient to support the requirements imposed by distributed testing, and so a different interface was introduced in dTET2.[3] This interface is not defined by a specification and so is only available for use in conjunction with the C TCM/API that is supplied in dTET2.

TETware-Lite does not support remote and distributed testing, and so does not need to support a dTET2-style `tcc` to TCM/API interface.

Fully-featured TETware will support a dTET2-like private interface in addition to the public TET-style interface. The public interface will be provided mainly for the benefit of APIs other than the TETware C API and may be used when local and remote test cases are being processed. The private interface will be used only by the TETware C API and may be used when local, remote and distributed test cases are being processed.

It is not intended to define the private interface in a specification as part of this project. If X/Open considers that it would be beneficial to produce such a specification, it is suggested that this should be made the subject of a separate project.

3.5 Locking strategy

The TET specification requires `tcc` to acquire various exclusive and non-exclusive locks when a test case is being processed. This is done so as to guard against unwelcome interaction between processes when a test case is being processed by more than one `tcc` invocation, or when multiple instances of a test case are being executed in parallel under the control of `tcc`.

In TET 1.10 and dTET2, locks are acquired both in the test case directory and possibly in the alternate execution directory as well. In ETET 1.10.3, it is possible to prevent locks from being acquired in this way by specifying certain communication variables.

In TETware, locking will be performed mostly as required by the TET specification. However, failure to acquire a lock because of an inability to write to a read-only file system will be ignored. Thus it will be possible for TETware to process a test suite which resides on a read-only file system.

---

3. Note that the dTET2 also supported the base TET `tcc` to TCM/API interface for non-distributed test cases.

3.6 Supported features

3.6.1 Configuration variables

3.6.1.1 Introduction

Configuration variables are variables which are defined by the user in a configuration file. There is one file for each of `tcc`'s modes of operation; namely, **build** mode, **execute** mode and **clean** mode. In dTET2, configuration variables may be defined both in files on the master system and in files on each slave system. Generally speaking, a variable defined in a file on the master system is visible on all systems, whereas a variable defined in a file on a slave system is visible only on the system on which it is defined.[4] In addition, dTET2 uses a file on the master system which contains variables that specify certain information about slave systems. This is known as the **distributed configuration file**.

Certain variables whose names begin with `TET_` are reserved for use by the TET implementation. In dTET2, a variable whose name begins with the prefix `TET_REM`*nnn*`_` has the prefix removed before the variable is sent to system *nnn*.

Further information about configuration variables is presented in appendix A, ‘‘Comparison tables for configuration variables’’.

3.6.1.2 Configuration variables derived from existing TET implementations

All the configuration variables which are implemented in TET 1.10 and dTET2 will be supported in TETware.

The following configuration variables will be supported in TETware in the same way as in ETET 1.10.3:

```
TET_API_COMPLIANT
TET_PASS_TC_NAME
```

3.6.1.3 New configuration variables

3.6.1.3.1 `TET_COMPAT` – select TETware compatibility mode

During the evolution of dTET2 and ETET, certain specification differences have emerged. An example of such a difference is the syntax of the `:parallel:` and `:repeat:` scenario file directives. A new configuration variable will be provided to enable test suite authors to select which behaviour is required.

The name of this variable is `TET_COMPAT`. Possible values for this variable are `dtet2` to select dTET2 behaviour or `etet` to select ETET 1.10.3 behaviour. No default value is defined for this variable.

---

4. It is possible to override this behaviour using the `TET_REM`*nnn*`_` mechanism.

`tcc` will inspect the value of this variable when a decision must be made whether to provide compatibility with dTET2 or ETET 1.10.3. A diagnostic will be printed if `tcc` needs to decide which compatibility mode to use and this variable is not set or set to an invalid value.

3.6.1.3.2 `TET_REM`*nnn*`_TET_RUN` – specify runtime test root directory

These variables may be set in the distributed configuration file. The effect of setting the variable for system *nnn* is the same as that of setting the `TET_RUN` environment variable on the master system.

3.6.2 Communication variables

3.6.2.1 Introduction

Communication variables are environment variables which are used to pass information to TET processes. They may be set in one of the following ways:

- Environment variables which influence the behaviour of `tcc` are set by the user before `tcc` is run.
- Environment variables which are used to communicate information to the TCM/API are set by `tcc` before a test case is processed.

Further information about communication variables is presented in appendix B, ‘‘Comparison tables for communication variables’’.

3.6.2.2 Communication variables derived from TET 1.10 and dTET2

All the communication variables which are implemented in TET 1.10 and dTET2 will be supported in TETware.

3.6.2.3 Communication variables derived from ETET 1.10.3

3.6.2.3.1 `TET_SUITE_ROOT` – change the default test suite root directory

This variable will be supported in TETware.[5] Since TETware is able to process remote and distributed test cases in addition to local test cases, the behaviour of this variable must be extended in order to be effective on remote systems.

In TETware, the `TET_SUITE_ROOT` communication variable will be mapped on to the `TET_TSROOT` distributed configuration variable on the master system. When `tcc` is executed, it determines the value of `TET_TSROOT` automatically on the master system. However, if `TET_SUITE_ROOT` is present in the environment, its value will instead be used for `TET_TSROOT` on the master system. Values of `TET_TSROOT` for other systems will be specified in the distributed configuration file as is done in dTET2.

---

5. A distributed configuration variable called `TET_TSROOT` is implemented in dTET2 which performs a similar function.

When `tcc` processes a test case on either a local system or a remote system, the value of the `TET_SUITE_ROOT` environment variable made available to the test case will be taken from the setting of the `TET_TSROOT` distributed configuration variable that is in effect on that system.

3.6.2.3.2 `TET_RUN` – specify runtime test suite root directory

This variable will be supported in TETware. Its setting will only be effective on the master system. Similar behaviour can be specified for remote system *nnn* by setting the corresponding `TET_REM`*nnn*`_TET_RUN` variable in the distributed configuration file on the master system.

3.6.2.3.3 Other communication variables in ETET 1.10.3

Given the nature of the following communication variables, it is not considered necessary to implement these variables at this time:

```
TET_EXTENDED
TET_JOURNAL_PATH
TET_LOCK
```

However, `tcc` will not prevent these variables from being passed in the environment when a test case is being processed. Thus, test cases written to an API which requires one of these variables to be in the environment will still work when run under the control of the TETware `tcc`.

The way that TETware performs locking when processing a test suite on a read-only file system is described in the section entitled ‘‘Locking strategy’’ elsewhere in this chapter.

3.6.3 Scenario directives

3.6.3.1 Introduction

The names of test cases to be processed by `tcc` are specified in a scenario file. Scenario directives specify how these test cases are to be processed.

Further information about scenario directives is presented in appendix C, ‘‘Comparison tables for scenario directives’’.

3.6.3.2 Directives derived from existing TET implementations

All the scenario file directives which are implemented in TET 1.10 and dTET2 will be supported in TETware.

The following scenario file directives will be supported in TETware in the same way as in ETET 1.10.3:

^*scenario-name*
@*executable-path*
`:group:`
`:parallel:`
`:repeat:`
`:timed_loop:`
`:random:`

In addition, ETET 1.10.3 permits the use of `;` to group colon-delimited directives using the following syntax:

`:`*group-execution*`[;...]:`*test-list*

This syntax will be supported in TETware.

3.6.3.3 Conflicts between dTET2 and ETET 1.10.3 scenario file syntax

There are conflicts between dTET2 and ETET 1.10.3 in the way that certain scenario directives are interpreted as follows:

- Where ETET 1.10.3 expects a directive to be followed by a test list, dTET2 expects the directive to be followed by the name of a file which contains the list of tests.
- Where a dTET2 directive of the form `:`*xxx*`:` has a list of tests associated with it, the lists of tests is delimited by a matching `:end`*xxx*`:` directive.

The TETware `tcc` will support both types of syntax for scenario file directives. When it is necessary to distinguish between dTET2 and ETET 1.10.3 syntax, the setting of the `TET_COMPAT` configuration variable will determine which syntax is accepted.[6]

Note that dTET2-style `:end`*xxx*`:` delimiters will not be accepted when `;` separated directives are used. In this case, only ETET-style syntax will be accepted.

An indication of which directives are affected by `TET_COMPAT` is given in one of the tables presented in appendix C, ‘‘Comparison tables for scenario directives’’.

3.6.3.4 New scenario directives

3.6.3.4.1 `:variable:` – specify configuration variables

A directive will be provided to enable configuration variables to be specified in a scenario file.

The syntax of this directive is as follows:

`:variable,`*name*=*value*`[,...]:`*test-case*

The configuration variable *name* is set to *value* while *test-case* is being processed according to `tcc`’s current mode of operation.[7]

---

6. The behaviour of this configuration variable is described in the section entitled ‘‘`TET_COMPAT` – select TETware compatibility mode’’ elsewhere in this chapter.

7. That is: **build**, **execute** or **clean** mode.

When assigning a value to a variable, *value* is interpreted as a fixed string.[8]

In order not to confuse the test harness, attempts to use this directive to set most variables whose name begins with `TET_` will be ignored. However, when a remote or distributed test case is being processed, it will be possible to assign a user-defined configuration variable[9] on a specific system by prefixing its name with `TET_REM`*nnn*`_` in the usual way.

The precedence of configuration variables set using the `:variable:` directive is above that of variables specified in configuration files but lower than that of variables specified with a `-v` option on the `tcc` command line.

For completeness, dTET2-style syntax for this variable will be accepted by `tcc` when dTET2 compatibility mode is in effect.

3.6.3.4.2 `:distributed:` – specify a distributed test case

dTET2 permits the use of the `:remote:` directive to specify both remote and distributed test cases. Test cases within the scope of this directive are processed either as remote or as distributed test cases, depending on whether or not the master system appears in the system list which accompanies the directive.

A directive will be provided to enable a test case to be processed as a distributed test case irrespective of whether the master system appears in the accompanying system list.

The syntax of this directive takes one of two forms, as follows:

> `:distributed,`*nnn*`[,...]:`*file*
>
> `:distributed,`*nnn*`[,...]:`
> *test-case*
> ...
> `:enddistributed:`

In the first form, each test case listed (one per line) in *file* is processed on the systems specified by *nnn* ... as a distributed test case. In the second form, each named *test-case* is processed on the systems specified by *nnn* ... as a distributed test case.

For completeness, ETET-style syntax for this variable will be accepted by `tcc` when ETET compatibility mode is in effect.

---

8. It is understood that in an existing unpublished extension to the base TET in which the `:variable:` directive is implemented, *value* may also be derived from another configuration variable or from the environment. While variable substitution may be desirable in a scenario file, it is considered more appropriate to perform this kind of processing by using a **scenario file preprocessor**. Such a preprocessor can also perform macro expansion and other similar operations. It is likely that a TET scenario file preprocessor will be the subject of a separate design specification exercise.

9. That is: a variable whose name does not begin with `TET_` .

4. Test case support

4.1 The C API

4.1.1 Introduction

The TETware C API will be derived from the one supplied with dTET2. Thus, local, remote and distributed test cases may be built using this API.

Certain changes and enhancements will be made to the supported functions in order to enable this API to be ported to non-Unix platforms. Features will be added which will enhance the functionality available to test case writers. These are described in the sections which follow.

In addition, support will be provided to enable test cases to operate in a multi-threaded environment on Unix-like systems. This is described in chapter 5, ‘‘Thread support’’.

Readers may find it helpful to refer to the section entitled ‘‘C language binding’’ in the dTET2 Programmers Guide when reading the sections that follow. For ease of reference, this section is reproduced in appendix F.

Further information about C API interfaces is presented in appendix D, ‘‘Comparison tables for C API interfaces’’.

4.1.2 Error reporting

At present, some API functions which can fail report errors in various ways, while others don’t really attempt to report errors at all. The reason for this is largely historical.

In TET, only `tet_fork()` and `tet_exec()` can fail and the reason for failure can (presumably) be found by examining `errno` (although the specification does not state this explicitly).

In dTET2, one of the functions which can fail (`tet_sync()`) prints an error message to the journal file on failure, whereas others (`tet_remexec()`, `tet_remwait()` and `tet_remkill()`) are required by the specification to set `errno` to indicate some (but not all) failure modes.

This behaviour will be retained in TETware in order to provide backward compatibility with the existing implementation. In addition, a more comprehensive error reporting mechanism for API functions will be available by the provision of certain global variables by the API.

The syntax of these variables is as follows:

```
extern int tet_errno;
extern char *tet_errlist[];
extern int tet_nerr;
```

When an API function fails, it will store a failure code in the global variable `tet_errno`. These failure codes will be defined in the file `tet_api.h`.

An array of short message strings will be provided in the tet_errlist[] array. Test suite authors may use tet_errno to index tet_errlist[] in order to obtain a string describing the reason for the failure of an API function. The number of strings will be made available in the global variable tet_nerr; test suite authors should check this value in order to avoid an array subscript error when indexing tet_errlist[].

The list of tet_errno values and the associated error message strings will be finalised during the development of TETware.

4.1.3 Changes in API function specifications

4.1.3.1 Introduction

The move to a symmetrical model for distributed processing in TETware will have a small impact on the specification of certain API functions. These changes will not affect the portability of existing distributed test cases which use the dTET2 API.

These changes are described in the sections which follow.

4.1.3.2 Remote system designations

At present, the function tet_remgetlist() returns the number of slave systems participating in a distributed test case. The system name list returned indirectly through *sysnames contains the (numerical) names of the slave systems.

In TETware, tet_remgetlist() will return the number of other systems participating in a distributed test case. The system name list returned indirectly through *sysnames will contain the names of the other systems. The system name list will always be zero-terminated. If tet_remgetlist() is called on system 0, the zero list entry is not counted when determining the function's return value. If tet_remgetlist() is called on systems other than system 0, the list entry for system 0 will be the last entry in the list and so will be counted when determining the function's return value. This behaviour will ensure that existing test cases which pass the system name list generated by tet_remgetlist() to a subsequent call to tet_sync() will continue to function correctly when linked with the TETware C API.

4.1.4 New API functions

4.1.4.1 Introduction

This section describes API functions which will be added both to TETware and TETware-Lite. Experience has shown that each of these functions will be of use to test suite authors.

Prototypes for the functions described here will be included in the file tet_api.h.

4.1.4.2 Making journal entries

4.1.4.2.1 `tet_printf()` – write formatted information line

A function will be provided to write one or more formatted information lines to the execution results file.

The syntax of this function is as follows:

```
int tet_printf(char *format, /* [arg,] */ ...);
```

This function formats the string specified by `format` which may contain `printf()`-like conversion specifications. If the string contains more than one information line, each line except the last should be delimited by a newline character.

If the formatted string contains a line that is longer than the maximum permitted for a journal information line, the API adds extra newlines in order to break the long line into two or more shorter lines. If possible, an added newline will replace a blank character in the string so that the string is broken on a word boundary.

When all formatting is complete, the lines are written to the execution results file by a call to `tet_minfoline()`[10].

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

4.1.4.2.2 `tet_vprintf()` – write formatted information line

A function will be provided to write one or more formatted information lines to the execution results file, which takes its arguments from a `varargs` argument list.

The syntax of this function is as follows:

```
int tet_vprintf(char *format, va_list ap);
```

The operation and return value of this function are the same as for `tet_printf()`, except that instead of being called with a variable number of arguments, it is called with a `varargs` argument list.

10. Thus ensuring that in a distributed test case the lines are written to the execution results file in a single operation.

4.1.5 New API functions for use in distributed test cases

4.1.5.1 Introduction

This section describes API functions which will be added to fully-featured TETware. Experience has shown that each of these functions will be of use to test suite authors when writing distributed test cases.

Prototypes for functions, structure declarations and symbolic constants described here will be included in the file `tet_api.h`.

4.1.5.2 Remote system information

4.1.5.2.1 `tet_remtime()` – return system time on remote system

A function will be provided to return the system time on a remote system.

The syntax of this function is as follows:

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

A successful call to `tet_remtime()` returns zero and the time on the remote system is returned indirectly through `*tp`. If a call to `tet_remtime()` is unsuccessful, −1 is returned and `tet_errno` is set to indicate the cause of the error.

4.1.5.2.2 `tet_getsysbyid()` – return `systems` file entry

A function will be provided to return the `systems` file entry for a specified system. This function will enable part of a distributed test case to determine the host (or node) names of other systems participating in the test case.

The syntax of this function is as follows:

```
struct tet_sysent {
        int ts_sysid;                   /* dTET2 system ID */
        char ts_name[TET_SNAMELEN];     /* system's host name */
};

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

`sysid` specifies the system ID for which the `systems` file entry is required. `sysp` points to a (user-supplied) area of memory in which the information is to be placed after a successful call.

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

4.1.5.3 Test case synchronisation

4.1.5.3.1 Introduction

Functions will be provided to enable authors of distributed test cases to gain access to all of the user sync facilities provided by the dTET2 synchronisation subsystem[11]. These facilities are as follows:

- Synchronisation between distributed test case parts on two or more systems to a particular sync point.
- The ability to sync unsuccessfully as well as successfully (that is, the ability to specify the **sync vote** that is to be used in the request).[12]
- The ability to send and receive **sync message data** when making a sync request.
- The ability to access status information about other participating systems that is available within the API after the completion of a sync event.

4.1.5.3.2 `tet_remsync()` – synchronise with other systems

A function will be provided to enable one part of a distributed test case to synchronise with other parts of the same test case executing on other systems.

The syntax for this function is as follows:

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

The behaviour of this function and meanings of its arguments are similar to those described for `tet_msync()` in the section entitled ‘‘Test case synchronisation’’ in the dTET2 Programmers Guide. Only the differences will be described here; these are as follows:

- The list of participating system names pointed to by `sysnames` is not terminated by a zero. Instead, the number of system names in the list must be specified explicitly by the `nsysname` argument.
- Since the distinction between master and slave systems is to be removed in TETware, system zero will not automatically be added to the list of participating systems. Instead, system zero must be included explicitly if it is to participate in the sync event.

---

11. For a description of these facilities and an explanation of the terminology used in this section, please refer to the chapter entitled ‘‘Test case synchronisation’’ in the dTET2 Installation and User Guide. For ease of reference, this section is reproduced in appendix G.

12. This will enable one part of a distributed test case to indicate to other parts its intention to abandon processing in a particular test purpose. At present the only way that this can be achieved is for a test case part not to sync as expected, thus causing the event to fail. This results in a number of unnecessary error messages being sent to the journal file.

- The `vote` argument should be set to one of the defined constants `TET_SV_YES` or `TET_SV_NO`, depending on whether the calling process wishes to sync successfully or unsuccessfully.

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

The `tet_remsync()` function will replace the existing `tet_sync()` and `tet_msync()` functions in the current dTET2 API. Although `tet_sync()` and `tet_msync()` will continue to be supported in TETware in order to provide backwards compatibility with previous dTET2 releases, they will be marked ‘‘to be withdrawn’’. Test suite authors should use `tet_remsync()` when writing new test cases.

4.1.5.3.3 Control over sync error reporting

At present, the API prints a message to the journal file when a call to `tet_sync()` or `tet_msync()` is unsuccessful. Although this behaviour is useful when a sync event fails unexpectedly, in practice the failure is often because one part of a distributed test purpose wants to abandon processing for some reason, usually after submitting an `UNRESOLVED` result. The test purpose prints its own diagnostic message in this circumstance, rendering the API-generated messages redundant. This is because the test purpose **intends** the sync event to fail; indeed, the sync failure is used to indicate to other participating systems that they should also abandon processing. Furthermore, the number of messages generated by the API can become quite large, particularly when more than two systems are involved. This is because the API on each system prints its own error messages detailing the status of all the other participating systems.

A mechanism will be introduced to enable authors of distributed test cases to exercise some control over the default error handling provided by the API and, optionally, provide customised sync error reporting. When a test suite author uses the ability to specify a **sync vote** in a `tet_remsync()` call in conjunction with this mechanism, spurious API-generated messages about intentional sync event failures will be eliminated from journal files.

If a call to an API sync function is unsuccessful, the API will call the sync error handling function pointed to by the global variable `tet_syncerr`.

This variable is declared as follows:

```
extern void (*tet_syncerr)(struct tet_syncstat *statp, int nstat);
```

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

When `(*tet_syncerr)()` is called by the API, `statp` points to the first in a list of structures describing the sync status of each of the other systems participating in the event. `nstat` specifies the number of structures in the list. The global variable `tet_errno` will be set to indicate the cause of the error before `(*tet_syncerr)()` is called.

The sync status structure is defined as follows:

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

/* sync state values */
#define TET_SS_NOTSYNCED        1       /* sync request not received */
#define TET_SS_SYNCYES          2       /* system voted YES */
#define TET_SS_SYNCNO           3       /* system voted NO */
#define TET_SS_TIMEDOUT         4       /* system timed out */
#define TET_SS_DEAD             5       /* process exited */
```

4.1.5.4 Making journal entries

4.1.5.4.1 `tet_minfoline()` – write multiple information lines

A function will be provided to write multiple information lines to the execution results file. This will enable a distributed test case part to write more than one information line to the execution results file as an atomic operation. The possibility that these lines might be interspersed with information lines from other parts of the same test is avoided.[13]

The syntax of this function is as follows:

```
int tet_minfoline(char **lines, int nlines);
```

`lines` points to the first in a list of pointers to strings which are to be written to the execution results file as an atomic operation. A NULL pointer in the list is ignored. `nlines` specifies the number of string pointers in the list.

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

4.1.6 API considerations for non-Unix systems

4.1.6.1 Introduction

The original TET specification was intended to be implemented on Unix-like (or, more specifically) POSIX.1-conforming systems. It will be necessary to modify the C API in order to enable TETware to operate on non-Unix systems.

The sections that follow describe the additions and changes that will be required. The information presented here is summarised in a table in appendix D, ‘‘Comparison tables for C API interfaces’’.

---

13. This API function will enable authors of distributed test cases to access a facility to write multiple lines atomically which already exists in the dTET2 execution results file access subsystem.

Refer to the section entitled ‘‘C language binding’’ in the dTET2 Programmers Guide for details of existing API functions. For ease of reference, this section is reproduced in appendix F.

4.1.6.2 Test case structure and management functions

The global variable `tet_nosigreset` will be provided on all platforms. However, its effectiveness on a non-Unix platform will be determined by the extent to which the platform supports Unix-style signal handling.

4.1.6.3 Insulating from the test environment

On a non-Unix platform, the ability to leave signals untouched or to ignore them will only be supported to the extent that the platform supports Unix-like signals. In particular, it should be noted that non-Unix platforms do not necessarily support all the POSIX.1 signals.

4.1.6.4 Generating and executing processes

4.1.6.4.1 Introduction

At present, the functions in this category are `tet_fork()` and `tet_exec()`. The specification for these functions assumes a Unix-like hierarchical process model which is not necessarily available on non-Unix platforms. In TETware these functions will only be supported on Unix-like systems; thus, test cases which use these functions will not be portable to non-Unix platforms.

New API functions for performing process control within test cases will be supported (to the extent that is possible) on all platforms where TETware is implemented. Prototypes for these functions will be provided in the file `tet_api.h`. These functions are described in the sections that follow.

4.1.6.4.2 `tet_spawn()` – start a new process

This function will initiate a new process and will perform an operation similar to a call to `tet_fork()` with a NULL `parentproc` argument, followed by a call to `tet_exec()` in the child process.

The syntax of this function is as follows:

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

The meanings of the arguments to `tet_spawn()` are the same as for the arguments to `tet_exec()`.

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

An appropriate definition for `pid_t` will be included in the file `tet_api.h` on platforms on which `pid_t` is not defined in the file `<sys/types.h>`.

A process that is started by `tet_spawn()` should be linked with the child process controller `tcmchild.o` and will otherwise behave as if it had been launched by a call to `tet_exec()`, subject to any limitations imposed by the different process model implemented on a non-Unix platform.

4.1.6.4.3 `tet_wait()` – wait for a process started by `tet_spawn()` to terminate

This function will enable a test case to wait for a process started by `tet_spawn()` to terminate and to obtain that process's exit status.

The syntax of this function is as follows:

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

`tet_wait()` waits for the process identified by `pid` to terminate and returns that process's exit status indirectly through `*statp`. `pid` is the process ID returned by a previous successful call to `tet_spawn()`. A successful call to `tet_wait()` returns zero. If a call to `tet_wait()` is unsuccessful, –1 is returned and `tet_errno` is set to indicate the cause of the error.

To the extent that the concepts are supported on non-Unix platforms, the value returned through `*statp` can be examined by the macros defined in `<sys/wait.h>` on Unix-like platforms. When these macros are not provided on a non-Unix platform, appropriate definitions will be included in the file `tet_api.h`.

Note that on platforms which do not support multi-tasking, a call to `tet_spawn()` will block until the new process terminates. However, `tet_wait()` can still be used to obtain the exit status of the new process on such platforms.

4.1.6.4.4 `tet_kill()` – terminate a process started by `tet_spawn()`

This function will enable a test case to terminate a process started by a previous call to `tet_spawn()`.

The syntax of this function is as follows:

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

`pid` is the process ID returned by a previous successful call to `tet_spawn()`. `sig` specifies the signal which is to be sent to the named process. `sig` is ignored on non-Unix platforms where signals are not supported; instead, an appropriate mechanism is used to terminate the process if possible.

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

This function is a no-op on platforms which do not support multi-tasking.

4.1.7 API considerations for TETware-Lite

4.1.7.1 Introduction

As has been indicated elsewhere in this document, TETware-Lite does not support remote or distributed processing. Therefore, certain API functions and variables which may be used in distributed test cases are not supported in TETware-Lite.

Certain API functions which are only appropriate for use in distributed test cases cannot operate in the environment provided by TETware-Lite. These functions will not be present in the API library supplied with TETware-Lite. This is to enable test case authors to identify at compile time the use of functions which are only supported in fully-featured TETware.

This section describes the way that each TETware version will affect the availability and behaviour of interfaces in the C API. The information presented here is summarised in a table in appendix D, ‘‘Comparison tables for C API interfaces’’.

4.1.7.2 Making journal entries

A call to `tet_minfoline()` will be functionally equivalent to calling `tet_infoline()` once for each line which is to be printed to the journal file.

4.1.7.3 Executed process functions

A call to `tet_exit()` will be functionally equivalent to calling `exit()`.

A call to `tet_logoff()` will have no effect.

4.1.7.4 Test case synchronisation

`tet_sync()`, `tet_msync()`, `tet_remsync()`, `tet_syncreport()` and `(*tet_syncerr)()` are not supported.

4.1.7.5 Remote system designations

A call to `tet_remgetlist()` will always return zero. A pointer to a list containing a single zero value will be returned indirectly through `**sysnames`.

A call to `tet_remgetsys()` will always return zero.

4.1.7.6 Remote process control

`tet_remexec()`, `tet_remwait()` and `tet_remkill()` are not supported.

4.1.7.7 Remote system information

`tet_remtime()` and `tet_getsysbyid()` are not supported.

4.2 The C++ API

A lightweight C++ API will be included in TETware. This will be implemented in the same way as in ETET 1.10.3. The file `tet_api.h` which is part of the C API will be enhanced so as to enable it to be used in this API as well. Local, remote and distributed test cases may be built using this API.

4.3 The `xpg3sh` API

The `xpg3sh` API from TET 1.10 is currently included in dTET2 and will be included in TETware. This API may be used in conjunction with local and remote test cases. As with dTET2, there is no support for distributed test cases when this API is used.

It is not intended to include this API in TETware distributions for non-Unix platforms.

4.4 The `ksh` API

The `ksh` API from ETET 1.10.3 will be included in TETware. This API may be used in conjunction with local and remote test cases. There will be no support for distributed test cases when this API is used.

It is anticipated that, after porting, this API will be usable on PC platforms where the MKS Toolkit is installed.

4.5 The `perl` API

The `perl` API from ETET 1.10.3 will be included in TETware. This API may be used in conjunction with local and remote test cases. There will be no support for distributed test cases when this API is used.

Whether or not this API is useful depends on whether a `perl` implementation is available on the target platform. At the time of writing, it has not been determined whether a `perl` implementation is available for all of the non-Unix platforms to which it is intended to port TETware.

5. Thread support

5.1 Introduction

The C Test Case Manager and API library will be enhanced in TETware so as to enable the use of threads in test cases. The support for threads will be present in both TETware and TETware-Lite versions which run on Unix-like platforms.

When TETware is built on a Unix-like platform, both ‘‘standard’’ and thread-safe versions of the TCM and the API library will be generated. Reasons for having two versions of these files include the following:

- The additional overhead of the thread-safe code will be avoided in the majority of test cases which do not require it.
- The need to link the majority of test cases which do not use threads against additional system libraries will be avoided. Thus, compatibility with existing TET implementations will be maintained.

If it is required to make use of threads in test case code, the test suite author will specify a compile-time option and link the test case with the thread-safe versions of the TCM and the API library.

5.2 API issues

5.2.1 Changes to existing API functions

The internal workings of certain API functions will be enhanced to include an option to generate both ‘‘standard’’ and thread-safe versions. When the thread-safe version is built, code will be enabled to control the execution of critical code sections with the use of mutexes, and to control access to global data with the use of data locks.

The interface to API functions will be the same, regardless of whether the standard or thread-safe versions are used. It is not anticipated that it will be necessary to provide special reentrant versions of any of the existing API functions.

5.2.2 New API functions

5.2.2.1 Generating and executing processes

5.2.2.1.1 `tet_fork1()` – create a new process containing a single thread

When the thread-safe API is used, a call to `tet_fork()` creates a child process which contains all the threads which exist in the parent process. A function will be provided which creates a child process containing only the thread of the calling process, but will otherwise behave as does `tet_fork()`.

The syntax of this function is as follows:

```
int tet_fork1(void (*childproc)(void), void (*parentproc)(void),
        int waittime, int validresults);
```

The arguments and return value of this function are the same as those of `tet_fork()`.

5.2.2.2 Thread control

5.2.2.2.1 `tet_thr_create()` – create a new thread

A function will be provided to create a new thread in a test purpose. When this function is used to create a new thread, the API stores information about the newly-created thread in order to enable the TCM to perform appropriate actions when the test purpose returns control to the TCM, or when an unexpected signal occurs.

The syntax of this function is as follows:

```
int tet_thr_create(void *stack_base, size_t stack_size,
        void (*start_routine)(void *), void *arg,
        long flags, thread_t *new_thread);
```

The arguments and return value are the same as those for the `thr_create()` function. A call to `tet_thr_create()` attempts to create a new thread by calling `thr_create()` with the same arguments. The return value of `tet_thr_create()` is the same as that of the underlying `thr_create()` call. Unlike other API calls, `tet_thr_create()` does not set `tet_errno` if the call fails.

Unexpected results may occur if a test suite creates a new thread other than by using this function.

5.2.2.3 Access to global API variables

Functions in the thread-safe API library will maintain per-thread versions of certain global variables that are defined in the API. Examples of such variables are `tet_errno` and `tet_child`.

Functions will be provided to return per-thread values of these variables. The names of the functions themselves will not be part of the API.[14] Instead, the file `tet_api.h` will redefine each variable when a compile-time option is in effect. When the global variable is redefined in this way, a reference to the variable will instead access the per-thread copy of the variable that is maintained by the API.

For example, if the name of the function to access the per-thread value of `tet_child` is `tet_thr_child()`, the following definition will be visible in `tet_api.h` when the compile-time option is used:

```
#define tet_child       (*tet_thr_child())
```

14. Therefore an API-conforming test case should never call any of these functions directly.

5.3 TCM enhancements

It is expected that, if a test purpose creates a new thread, the test purpose should take responsibility for terminating the new thread and waiting for them before return. However, in order to guard against the case where this does not happen, and also to provide for the handling of unexpected signals, the thread-safe version of the TCM will be enhanced as follows:

- The TCM will call each test purpose using a single thread (the **main** thread). When a test purpose thread returns control to the TCM:
  - if the thread is not the main thread, the thread exits
  - if the thread is the main thread, it waits for each of the other threads to exit
- When an unexpected signal is caught by the TCM:
  - if the thread that catches the signal is not the main thread, it forwards the signal to the main thread and exits
  - if the thread that catches the signal is the main thread, it terminates each of the other threads
- When the TCM waits for a thread to exit, it terminates the thread if the thread does not exit after a certain time has expired.
- When the TCM terminates a thread, it does so by sending a `SIGTERM` signal to the thread. If the thread does not exit after a certain time, the TCM sends a `SIGKILL` signal which forcibly terminates the thread.

5.4 Client/server issues

5.4.1 Introduction

In fully-featured TETware, certain API calls result in requests being sent to TETware servers. Examples of such calls are `tet_remsync()` which sends a request to the synchronisation daemon `tetsyncd`, and `tet_infoline()` which sends a request to the execution results daemon `tetxresd`. Although it would be possible for each thread in a multi-threaded test purpose to make its own connection to each TETware server, in practice this is not considered necessary. The reasons for this are different for each type of server and are presented in the sections that follow.

Instead, the flow-of-control for each API function that passes though a particular server will be regarded as a critical code section. One mutex will be used to control access to each server and its associated process table entry in the client process.

5.4.2 The test case controller daemon tccd

A test case sends a request to a tccd running on a remote system as a result of a call to one of the remote process control functions.[15] One such call – tet_remwait() – can block when a non-zero timeout parameter is specified. Since access to each server is to be controlled by a mutex, a call to tet_remwait() which blocks will prevent access to a particular tccd by other threads in the same process.

However, the use of remote process control functions is deprecated in TETware; indeed, these functions are only supported in order to provide backwards compatibility with dTET2. Instead, if it is necessary to execute a process on another system which is participating in a distributed test case, it is recommended that the process should be started by the part of the test case that is executing on that system.

The limitation on the use of remote process control functions in a multi-threaded environment will be documented (together with this recommendation) in the function descriptions in the TETware Programmers Guide.

5.4.3 The synchronisation daemon tetsyncd

A test case sends a request to tetsyncd as a result of a call to one of the API's synchronisation functions. Since synchronisation is defined in terms of **systems** and not processes, only one process on a particular system may represent that system in a particular synchronisation event. A consequence of this requirement is that it is an error for two processes on the same system to make overlapping synchronisation requests. It follows, therefore, that in a multi-threaded environment it would be an error for two threads on the same system to make overlapping synchronisation requests (whether or not the threads are in the same process).

In view of this there would be no advantage to be gained by enabling individual threads to make their own connections to tetsyncd.

5.4.4 The execution results daemon tetxresd

A test case sends a request to tetxresd as a result of a call to one of the API functions which process information which is to appear in the journal file. Since all file i/o operations must be single-threaded, there would be no significant advantage to be gained by enabling individual threads to make their own connections to tetxresd.

15. The functions in this category are: tet_remexec(), tet_remwait() and tet_remkill().

6. Portability to non-Unix platforms

6.1 Introduction

The TET specification assumes that TET will be implemented on a Unix-like platform. In addition, the design of existing TET and dTET2 implementations make use of Unix-style operating system services and library routines which are not always available on non-Unix platforms.

Examples of such services include:

- Multi-tasking using a preemptive scheduler.
- A hierarchical process structure.
- Notification of asynchronous events using POSIX.1-style signals.
- Use of `fork()` to clone a copy of the calling process.
- Use of `read()` and `write()` system calls to perform i/o using arbitrary record sizes.
- Use of pipes to transfer data between processes.
- Access to network services using socket calls or XTI.
- The availability of a software development system with a user interface which is, for the most part, the same on all systems.

In addition, it is common for compilers which run on some popular PC platforms to require the use of extensions to the C language that are necessary in order to support certain features of the PC's machine architecture.

It is intended to make TETware available on a number of popular platforms outside the traditional sphere of Unix-like and POSIX.1-conforming environments. The impact that the non-availability of certain Unix-like system services might have on TETware functionality and portability will vary, depending on the nature of the difference between the Unix-like and non-Unix platform.

When considering the differences between the various platforms and how they might be overcome, several classes of difference emerge. The first class of difference relates to the case where there is an ANSI and a non-ANSI way of performing a particular task. For example, the lack of Unix-style `read()` and `write()` system calls can be overcome fairly easily by the use of their ANSI equivalents. Furthermore, if code that uses the Unix-style calls is replaced by code that uses the ANSI calls, the modified code is still portable to all systems.

A second class of difference between Unix and non-Unix systems is a little more difficult to accommodate. If TETware uses a system interface or header file element which is not available on a non-Unix system, a compatibility library function or header file element must be supplied which may be different for each affected platform. For example, on a platform which supports the concept of a process ID but does not include a definition for `pid_t` in `<sys/types.h>`, a suitable definition must be visible in one of the TETware header files when the code is compiled on that platform.

A third class of difference involves situations where the design of the code must be changed. For example, the lack of a `fork()` system call on many non-Unix systems means that a design which makes use of the properties of `fork()` must be changed. This is one of the motivations behind the intention to re-design `tcc` in TETware, in the way that is described in chapter 3, ‘‘The Test Case Controller `tcc`’’.

A final class of difference is one where it is pragmatic simply to accept the limitation imposed by the difference. An example of such a difference is the lack of multi-tasking which uses a preemptive scheduler. In such cases TETware will still work on the affected platform but with some loss of functionality.

The way in which differences between Unix-like and non-Unix systems will be catered for in TETware are described in the sections which follow. It should be understood that the information presented here refers to the part of TETware that is written in C; namely, the Test Case Controller subsystem and the C TCM/API. Information related to the portability of each of the other TCM/APIs is presented in the section describing the API concerned in chapter 4, ‘‘Test case support’’.

Comparisons of functionality between TETware on different platforms are summarised in tables which are presented in several appendices at the end of this document.

6.2 Supported platforms

To the extent possible, it is intended to port TETware to the following non-Unix platforms:

- DOS + Windows 3.11
- Windows 95
- Windows NT

When TETware is ported to a platform which supports a graphical user interface, it will be compiled as a console application. It is suggested that the provision of a graphical user interface to TETware should be undertaken as a separate project if this is considered important by X/Open.

6.3 TETware-Lite

TETware-Lite will be ported to all of the supported platforms.

6.4 Fully-featured TETware

Fully-featured TETware uses a client-server architecture and so the demands that are placed on the underlying operating system are rather greater than those imposed by TETware-Lite. In particular, the design of the Test Case Controller subsystem in fully-featured TETware requires the underlying operating system to support multi-tasking using a preemptive scheduler.

The effect of this is that it will be possible to run `tcc` on a platform which supports multi-tasking and so these platforms may operate either as a local (or master) system or as a remote system. However, it will only be possible for a platform which does not support multi-tasking to operate as a remote system under the control of a `tcc` which is running on a multi-tasking platform.

It is anticipated that the following non-Unix platforms will support TETware both as a local and a remote system:

- Windows NT

whereas the following platforms will only support TETware as a remote system which does not run `tcc`:

- DOS + Windows 3.11
- Windows 95

It should be noted that it will only be possible for platforms that do not support multi-tasking to participate in a single remote or distributed test case at a time.

6.5 Compiler subsystem issues

When TETware is ported to each non-Unix system, it will be ported in a form suitable for use with a particular software development environment. The resulting code will only be guaranteed to compile successfully if the specified environment is in place on the target system.

Details of the environment required to compile TETware for each supported platform will be presented in the Installation and Build Notes for the target platform.[16]

In order to accommodate the differences between the environments on each of the target platforms, conditional compilation will be used in the source code wherever it is practical to do so. Where this is not practical, platform-specific code will be separated out into platform-specific source files. Platform-specific `make` definition files will contain sufficient information to enable the appropriate version of each source file to be selected when each TETware component is to be built.

16. Refer to the section entitled ‘‘Installation and Build Notes’’ in chapter 7 for further details.

7. Documentation

7.1 Introduction

The sections that follow describe the documents that will be supplied with the TETware distribution. These documents will be supplied in Postscript form.

The documents included in the TETware distribution will all describe the state of the product at the time the distribution is made. Out-of-date documents and documents describing previous TET releases will not be included.

7.2 Document source format

The documentation in existing TET implementations is written and supplied in `troff` source format. More recently, Postscript format documentation has also been included in most TET releases.

It is understood that X/Open is considering as a general policy the merits of supplying documentation in HTML format. This is in order to enable documents to be viewed conveniently with a suitable Web browser. It will be seen from the sections that follow that it is proposed to base the TETware guides on existing (`troff`- and Postscript-format) TET and dTET2 documents.

If HTML format documents are to be produced for the next TET release, it will be necessary to convert and/or re-structure the `troff` source of each document to a format and structure more suitable for use with HTML. It is anticipated that such a conversion will require a fair amount of effort. In view of this, it is suggested that if X/Open wish to have HTML-format documentation at this time, the conversion of TET documentation to HTML format should be undertaken as a separate project.

7.3 Programmers Guide

Material for this guide will be taken from the Programmers Guides that are supplied with dTET2 and ETET 1.10.3.

The structure of the TETware Programmers Guide will differ from that of previous Programmers Guides as follows:

- Sections which consist mainly of material copied from the original TET specification will be reviewed for relevance to the target audience. These sections will be edited, relocated or removed as necessary, so that the guide's clarity and usability is enhanced.
- A chapter describing how to write a distributed test case will be added. In particular, programming techniques for the effective use of synchronisation between test case parts will be described.
- A section will be added which describes how to use the `(*tet_startup)()` facility to support Internationalisation and the use of locales.

- A section will be added which describes how to use TETware to process a test suite which resides on a read-only file system.
- A section will be added which describes how to write a multi-threaded test case.
- Detailed descriptions of C API functions will be presented in `man`page format.
- The comparison tables contained in several of the appendices to this document will be included in the updated Programmers Guide.

An indication of the status of each C API function will be included in the description of each function as follows:

— whether the function is specific to TETware implementations on Unix-like systems or supported on all TETware implementations

— whether the function should be used in new test cases, provided only for backwards compatibility with previous TET releases, or deprecated for some other reason

When use of a function is deprecated for any reason, the function's description will indicate which current function or facility should be used in order to achieve the same effect.

7.4 Installation and User Guide

This guide will be derived from the dTET2 Installation and User Guide. Sections will be updated and/or added as necessary to describe the operation of TETware, including at least the following areas:

- The symmetrical architecture of TETware systems.
- TETware-Lite *vs*. fully-featured TETware.
- Using TETware on different supported platforms.
- The different TCM/APIs supplied with TETware.
- Configuration variables.
- Communication (or environment) variables.
- Scenario file directives.
- Sections describing TETware for users familiar with TET 1.10, dTET2 and ETET 1.10.3.

Most of the information in the section describing how to install and build the test harness will be relocated to a separate document entitled ‘‘TETware Installation and Build Notes’’. This information will be replaced by a pointer to that document.

7.5 Installation and Build Notes

One set of Installation and Build Notes will be produced for each platform on which TETware is supported. These Notes will replace the section in the Installation and User Guide which describes how to build and install the test harness.

As has been noted elsewhere, it is intended to supply TETware in binary form on certain platforms. The versions of the Installation and Build Notes for such platforms will make a clear distinction between information related to the source distribution and information relating to the binary distribution on that platform.

7.6 Specification document

It is not intended to include a specification document with TETware distributions.

The base TET was first described in a document entitled ‘‘Test Environment Toolkit – Architectural, Functional and Interface Specification’’. Although some changes have been made to this document during the evolution of the TET, information in the document has not always been kept in step with TET implementations. As a result, existing versions of the Specification document are not really suitable for use as the basis of a TETware Specification document and it is anticipated that a fair amount of effort would be required to produce a definitive Specification document for TETware.

In view of this, it is suggested that if the availability of a definitive specification for TETware is important to X/Open, this should be undertaken as a separate project. It is anticipated that a definitive TETware Specification would include material from the base TET Specification and also from this document.

7.7 Release Notes

7.7.1 Generic Release Notes

Each release of TETware will be accompanied by a set of generic Release Notes. It is intended that this document will be the very first document that a user reads when receiving the release.[17] The Release Notes document will contain information about the particular TETware release that it accompanies, together with pointers to the other documents in the release.

It is anticipated that the generic Release Notes will contain information under at least the following headings:

- New features in this release
- Status of this release

17. And, hopefully, before attempting to start work with the release!

- Problems fixed since the last release
- Known problems in this release
- Building and installing TETware
- Problem reporting

In order to maximise the clarity and usability of the Release Notes, this document will not contain detailed instructions concerning how to install or operate TETware. Instead, references will be provided to the chapters and sections in the main documents where this information is presented.

7.7.2 Supplementary Release Notes for specific platforms and distribution types

It is anticipated that there may be circumstances where it is necessary to issue a supplement to the generic Release Notes. When a supplement is issued, it will contain information which is relevant only to a particular platform on which a release of TETware is made.

When it is necessary to issue such a supplement, it is anticipated that it will have a title similar to ‘‘TETware Release *M.N* – Release Notes Supplement for *type* Distributions on *XXX* Platforms’’, where *M.N* is the release number, *type* indicates a **Source**, a **Binary** or a **Source and Binary** distribution and *XXX* specifies the target platform for which the release is made.

8. Miscellaneous issues

8.1 Compatibility with existing TET implementations

8.1.1 Introduction

TETware provides backwards compatibility for test cases written to run under the control of existing TET implementations; namely, TET 1.10, dTET2 and ETET 1.10.3.

Compatibility information relating to various TETware features is summarised in the tables presented in several appendices at the end of this document.

8.1.2 Compatibility issues

8.1.2.1 C language test cases

TETware provides compatibility at source code level for all existing API-conforming test cases that are written using the C and C++ APIs. Existing test cases should be re-linked with the TETware TCM and API library in order to take full advantage of the facilities provided by the TETware API.

In addition, support is provided in TETware for all existing TET 1.10 and most ETET 1.10.3[18] test case binaries. However, dTET2 test cases must be relinked with the TETware TCM and API library in order to function correctly.

In a few cases it will be necessary to make changes to test cases which have been written to run under the control of previous TET implementations as follows:

- If a TET or ETET test purpose makes a call to `exit()`, it is desirable to replace this with a call to `tet_exit()`.[19] If this is not done, the test case will still function correctly in every respect but a harmless ‘‘client closed connection’’ message will be emitted by `tetsyncd` and `tetxresd` when the test case exits.[20]

---

18. Except test cases which rely on the behaviour associated with `TET_EXTENDED`, `TET_JOURNAL_PATH` and `TET_LOCK`. Refer to the section entitled ‘‘Communication variables derived from ETET 1.10.3’’ in chapter 3 for a discussion concerning these variables.
19. Refer to the section entitled ‘‘Executed process functions’’ in the dTET2 Programmers Guide for a description of `tet_exit()`. For ease of reference the chapter containing this section is reproduced in appendix F.
20. This behaviour will only be observed in fully-featured TETware. It will not occur in TETware-Lite since, in this case, `tcc` does not make use of server processes.

- For the same reason, it is desirable to insert a call to `tet_logoff()` immediately before an explicit call to one of the `exec()` functions.[21]
- If a scenario file contains one of the directives which has different syntax in ETET 1.10.3 and dTET2, it will be necessary to include a definition for the TETware-specific `TET_COMPAT` variable in the configuration file for each of `tcc`'s modes of operation.[22]

In TET 1.10 and ETET 1.10.3 it is possible to execute a C language test case directly from the command line without the use of `tcc`. However, in dTET2 it is not possible to execute a C language test case directly from the command line; instead, such test cases must always be executed under the control of `tcc`.

When TETware-Lite is used, it is possible to execute a C language test case either independently or under the control of `tcc`. However, when fully-featured TETware is used, it is not possible for the user to set up the environment that the C API needs in order to support remote or distributed testing. Thus a C language test case cannot be executed independently but must always be run under the control of `tcc`.

8.1.2.2 Test cases which use other APIs

Support is provided for test cases whose APIs use the interface between `tcc` and the TCM/API that is described in the base TET specification. APIs which use this interface include the `xpg3sh`, `ksh` and `perl` APIs. This support is present in both TETware and TETware-Lite.

As in existing TET implementations, it is possible to execute a test case independently in TETware as well as under the control of `tcc` when one of these APIs is used.

---

21. Refer to the section entitled ‘‘Executed process functions’’ in the dTET2 Programmers Guide for a description of `tet_logoff()`. For ease of reference the chapter containing this section is reproduced in appendix F.

22. This issue is discussed in the section entitled ‘‘Conflicts between dTET2 and ETET 1.10.3 scenario file syntax’’ in chapter 3.

    When the TETware `tcc` encounters a syntax ambiguity in a scenario file and this variable is not set, it prints a diagnostic and does not process the scenario. Thus the user is protected from the possibility of unexpected default behaviour when a test suite designed to run under the control of ETET 1.10.3 or dTET2 is processed by TETware.

    Refer to the section entitled ‘‘`TET_COMPAT` – select TETware compatibility mode’’ in chapter 3 for a description of this configuration variable.

8.1.3 Combining test cases from existing TET implementations

TETware is able to process a scenario which contains a mixture of test cases taken from TET 1.0, dTET2 and ETET 1.10.3 test suites. However, if such a scenario uses directives whose syntax is different in dTET2 and ETET 1.10.3, it will be necessary to decide which syntax to use and then use the chosen syntax throughout the scenario file.

APPENDICES

A. Comparison tables for configuration variables

A.1 Introduction

This appendix contains tables which indicate the status of each configuration variable in several contexts.

The meanings of symbols which appear in these tables are as follows:

√	configuration variable has the effect described in the specification
∂	configuration variable sometimes has effect
○	configuration variable has no effect
1 2 etc.	refer to notes at the end of each table table

A.2 Support for configuration variables in different TETware versions

Configuration variable name	TETware	TETware-Lite
TET_API_COMPLIANT	√	√
TET_BUILD_FAIL_TOOL	√	√
TET_BUILD_FILE	√	√
TET_BUILD_TOOL	√	√
TET_CLEAN_FILE	√	√
TET_CLEAN_TOOL	√	√
TET_COMPAT	√	√
TET_EXEC_FILE	√	√
TET_EXEC_IN_PLACE	√	√
TET_EXEC_TOOL	√	√
TET_LOCALHOST	√	○
TET_OUTPUT_CAPTURE	√	√
TET_PREBUILD_TOOL	√	√
TET_REM*nnn*_TET_EXECUTE	√	○
TET_REM*nnn*_TET_ROOT	√	○
TET_REM*nnn*_TET_RUN	√	○
TET_REM*nnn*_TET_TSROOT	√	○
TET_REM*nnn_variable*	√	∂ 1
TET_RESCODES_FILE	√	√
TET_SAVE_FILES	√	√
TET_SIG_IGN	√	√
TET_SIG_LEAVE	√	√
TET_TC_PASS_NAME	√	√
TET_TRANSFER_SAVE_FILES	√	○
TET_XTI_MODE	√	○
TET_XTI_TPI	√	○

Notes:

1. Only effective when *nnn* is `000`.

A.3 Support for configuration variables on different platforms

When remote or distributed test cases are being processed, it is possible to define a variable in a configuration file on the master system which applies to a remote system. In addition, it is possible to define a variable in a configuration file on a remote system which applies only to that system. When interpreting the symbols in this table it should be understood that the symbol refers to the platform on which the variable is **defined** and not necessarily to the platform to which the variable applies.

Configuration variable name	Platform		
	Unix POSIX.1	DOS Windows	Windows NT
TET_API_COMPLIANT	√	√	√
TET_BUILD_FAIL_TOOL	√	√	√
TET_BUILD_FILE	√	√	√
TET_BUILD_TOOL	√	√	√
TET_CLEAN_FILE	√	√	√
TET_CLEAN_TOOL	√	√	√
TET_COMPAT	√	√	√
TET_EXEC_FILE	√	√	√
TET_EXEC_IN_PLACE	√	√	√
TET_EXEC_TOOL	√	√	√
TET_LOCALHOST	√	○	○
TET_OUTPUT_CAPTURE	√	√	√
TET_PREBUILD_TOOL	√	√	√
TET_REM*nnn*_TET_EXECUTE	√	○	√
TET_REM*nnn*_TET_ROOT	√	○	√
TET_REM*nnn*_TET_RUN	√	○	√
TET_REM*nnn*_TET_TSROOT	√	○	√
TET_REM*nnn_variable*	√	∂ 1	√
TET_RESCODES_FILE	√	√	√
TET_SAVE_FILES	√	√	√
TET_SIG_IGN	√	○	√
TET_SIG_LEAVE	√	○	√
TET_TC_PASS_NAME	√	√	√
TET_TRANSFER_SAVE_FILES	√	○	√
TET_XTI_MODE	√	○	○
TET_XTI_TPI	√	○	○

1. A fully-featured TETware master system is not supported on this platform. Thus, setting this variable is only effective when the platform acts as a remote system.

A.4 Compatibility with configuration variables in existing TET implementations

Configuration variable name	TET implementation			
	TET 1.10	ETET 1.10.3	dTET2 2.3	TETware
TET_API_COMPLIANT	○	√	○	√
TET_BUILD_FAIL_TOOL	√	√	√	√
TET_BUILD_FILE	√	√	√	√
TET_BUILD_TOOL	√	√	√	√
TET_CLEAN_FILE	√	√	√	√
TET_CLEAN_TOOL	√	√	√	√
TET_COMPAT	○	○	○	√
TET_EXEC_FILE	√	√	√	√
TET_EXEC_IN_PLACE	√	√	√	√
TET_EXEC_TOOL	√	√	√	√
TET_LOCALHOST	○	○	√ 1	√ 1
TET_OUTPUT_CAPTURE	√	√	√	√
TET_PREBUILD_TOOL	○	○	√	√
TET_REM*nnn*_TET_EXECUTE	○	○	√	√
TET_REM*nnn*_TET_ROOT	○	○	√	√
TET_REM*nnn*_TET_RUN	○	○	○	√
TET_REM*nnn*_TET_TSROOT	○	○	√	√
TET_REM*nnn_variable*	○	○	√	√
TET_RESCODES_FILE	√	√	√	√
TET_SAVE_FILES	√	√	√	√
TET_SIG_IGN	√	√	√	√
TET_SIG_LEAVE	√	√	√	√
TET_TC_PASS_NAME	○	√	○	√
TET_TRANSFER_SAVE_FILES	○	○	√	√
TET_XTI_MODE	○	○	√ 1	√ 1
TET_XTI_TPI	○	○	√ 1	√ 1

Notes:

1. This variable is only effective when the XTI network transport interface is used.

B. Comparison tables for communication variables

B.1 Introduction

This appendix contains tables which indicate the status of each communication variable in several contexts.

The meanings of symbols which appear in these tables are as follows:

√	communication variable has the effect described in the specification
∂	communication variable sometimes has effect
○	communication variable has no effect
1 2 etc.	refer to notes at the end of each table table

When interpreting the information contained in these tables, it should be understood that communication variables are provided for use by the test harness and are not part of the API. Therefore, the effectiveness or otherwise of a particular communication variable will not affect the portability of API-conforming test cases between different TET implementations or supported platforms.

The following variables are part of the published interface between the user and `tcc`:

```
TET_EXECUTE
TET_EXTENDED
TET_LOCK
TET_ROOT
TET_RUN
TET_SUITE_ROOT
TET_TMP_DIR
```

The following variables are part of the published interface between `tcc` and the TCM/API:

```
TET_ACTIVITY
TET_CODE
TET_CONFIG
TET_EXTENDED
TET_JOURNAL_PATH
TET_LOCK
TET_ROOT
TET_RUN
TET_SUITE_ROOT
```

B.2 Support for communication variables in different TETware versions

Communication variable name	TETware	TETware-Lite
TET_ACTIVITY	√	√
TET_CODE	√	√
TET_COM_VAR	√	√
TET_CONFIG	√	√
TET_DIST	√	○
TET_EXECUTE	√	√
TET_EXTENDED	○	○
TET_JOURNAL_PATH	○	○
TET_LOCK	○	○
TET_ROOT	√	√
TET_RUN	√	√
TET_SUITE_ROOT	√	√
TET_TIARGS	√	○
TET_TMP_DIR	√	√
TET_TSARGS	√	○

B.3 Support for communication variables on different platforms

Communication variable name	Platform		
	Unix POSIX.1	DOS Windows	Windows NT
TET_ACTIVITY	√	√	√
TET_CODE	√	√	√
TET_COM_VAR	√	√	√
TET_CONFIG	√	√	√
TET_DIST	√	√	√
TET_EXECUTE	√	√	√
TET_EXTENDED	○	○	○
TET_JOURNAL_PATH	○	○	○
TET_LOCK	○	○	○
TET_ROOT	√	√	√
TET_RUN	√	√	√
TET_SUITE_ROOT	√	√	√
TET_TIARGS	√	√	√
TET_TMP_DIR	√	√	√
TET_TSARGS	√	√	√

B.4 Compatibility with communication variables in existing TET implementations

Communication variable name	TET implementation			
	TET 1.10	ETET 1.10.3	dTET2 2.3	TETware
`TET_ACTIVITY`	√	√	√	√
`TET_CODE`	√	√	√	√
`TET_COM_VAR`	○	○	√	√
`TET_CONFIG`	√	√	√	√
`TET_DIST`	○	○	√	√
`TET_EXECUTE`	√	√	√	√
`TET_EXTENDED`	○	√	○	○1
`TET_JOURNAL_PATH`	○	√	○	○1
`TET_LOCK`	○	√	○	○1
`TET_ROOT`	√	√	√	√
`TET_RUN`	○	√	○	√
`TET_SUITE_ROOT`	○	√	○	√
`TET_TIARGS`	○	○	√	√
`TET_TMP_DIR`	√	√	√	√
`TET_TSARGS`	○	○	√	√

Notes:

1. It is not considered necessary to implement this variable at present. Refer to the section entitled ‘‘Other communication variables in ETET 1.10.3’’ in chapter 3.

C. Comparison tables for scenario directives

C.1 Introduction

This appendix contains tables which indicate the status of each scenario file directive in several contexts.

The meanings of symbols which appear in these tables are as follows:

√	scenario directive is fully supported as described in the specification
∂	scenario directive is partially supported
×	scenario directive is not supported
1 2 etc.	refer to notes at the end of each table table

Certain TET scenario directives consist of more than one line. This is indicated by indenting the second and subsequent lines in a multi-line directive.

C.2 Support for scenario directives in different TETware versions

Scenario directive	TETware	TETware-Lite
# *comment*	√	√
scenario-name	√	√
test-case	√	√
"*text*"	√	√
^*scenario-name*	√	√
@*test-case*	√	√
`:include:`*file*	√	√
`:parallel`[`,`*count*]`:`*test-list*	√	√
`:parallel`[`,`*count*]`:`*file*	√	√

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

`:group`[`,`*count*]`:`*test-list*	√	√
`:group`[`,`*count*]`:`*file*	√	√

`:group`[`,`*count*]`:`
test-case
[...]
`:endgroup:`
	√	√

`:repeat,`*count*`:`*test-list*	√	√
`:repeat,`*count*`:`*file*	√	√

`:repeat,`*count*`:`
test-case
[...]
`:endrepeat:`
	√	√

`:random:`*test-list*	√	√
`:random:`*file*	√	√

`:random:`
test-case
[...]
`:endrandom:`
	√	√

`:timed_loop,`*seconds*`:`*test-list*	√	√
`:timed_loop,`*seconds*`:`*file*	√	√

`:timed_loop,`*seconds*`:`
test-case
[...]
`:endtimed_loop:`
	√	√

`:remote,`*nnn*[`,`...]`:`*test-list*	√	×
`:remote,`*nnn*[`,`...]`:`*file*	√	×

`:remote,`*nnn*[`,`...]`:`
test-case
[...]
`:endremote:`
	√	×

`:distributed,`*nnn*[`,`...]`:`*test-list*	√	×
`:distributed,`*nnn*[`,`...]`:`*file*	√	×

`:distributed,`*nnn*[`,`...]`:`
test-case
[...]
`:enddistributed:`
	√	×

`:variable,`*name*=*value*[`,`...]`:`*test-list*	√	√
`:variable,`*name*=*value*[`,`...]`:`*file*	√	√

`:variable,`*name*=*value*[`,`...]`:`
test-case
[...]
`:endvariable:`
	√	√

`:`*group-execution*[`;`...]`:`*test-list*	√	√
`:`*group-execution*[`;`...]`:`*file*	√	√

C.3 Support for scenario directives on different platforms

Scenario directive	Platform		
	Unix POSIX.1	DOS Windows	Windows NT
# *comment*	√	√	√
scenario-name	√	√	√
test-case	√	√	√
"*text*"	√	√	√
ˆ*scenario-name*	√	√	√
@*test-case*	√	√	√
`:include:`*file*	√	√	√
`:parallel`[`,`*count*]`:`*test-list*	√	√	√
`:parallel`[`,`*count*]`:`*file*	√	√	√
`:parallel`[`,`*count*]`:` *test-case* [. . .] `:endparallel:`	√	√	√
`:group`[`,`*count*]`:`*test-list*	√	√	√
`:group`[`,`*count*]`:`*file*	√	√	√
`:group`[`,`*count*]`:` *test-case* [. . .] `:endgroup:`	√	√	√
`:repeat,`*count*`:`*test-list*	√	√	√
`:repeat,`*count*`:`*file*	√	√	√
`:repeat,`*count*`:` *test-case* [. . .] `:endrepeat:`	√	√	√
`:random:`*test-list*	√	√	√
`:random:`*file*	√	√	√
`:random:` *test-case* [. . .] `:endrandom:`	√	√	√
`:timed_loop,`*seconds*`:`*test-list*	√	√	√
`:timed_loop,`*seconds*`:`*file*	√	√	√
`:timed_loop,`*seconds*`:` *test-case* [. . .] `:endtimed_loop:`	√	√	√
`:remote,`*nnn*[`,`. . .]`:`*test-list*	√	√	√
`:remote,`*nnn*[`,`. . .]`:`*file*	√	√	√
`:remote,`*nnn*[`,`. . .]`:` *test-case* [. . .] `:endremote:`	√	√	√
`:distributed,`*nnn*[`,`. . .]`:`*test-list*	√	√	√
`:distributed,`*nnn*[`,`. . .]`:`*file*	√	√	√
`:distributed,`*nnn*[`,`. . .]`:` *test-case* [. . .] `:enddistributed:`	√	√	√
`:variable,`*name*=*value*[`,`. . .]`:`*test-list*	√	√	√
`:variable,`*name*=*value*[`,`. . .]`:`*file*	√	√	√
`:variable,`*name*=*value*[`,`. . .]`:` *test-case* [. . .] `:endvariable:`	√	√	√
`:`*group-execution*[`;`. . .]`:`*test-list*	√	√	√
`:`*group-execution*[`;`. . .]`:`*file*	√	√	√

C.4 Compatibility with scenario directives in existing TET implementations

Scenario directive	TET implementation			
	TET 1.10	ETET 1.10.3	dTET2 2.3	TETware
# *comment*	√	√	√	√
scenario-name	√	√	√	√
test-case	√	√	√	√
"*text*"	√	√	√	√
ˆ*scenario-name*	×	√	×	√
@*test-case*	×	√	×	√
:include:*file*	×	×	√	√
:parallel[,*count*]:*test-list*	×	√	×	√ 1
:parallel[,*count*]:*file*	×	×	∂ 2	√ 1
:parallel[,*count*]: *test-case* [. . .] :endparallel:	×	×	∂ 2	√
:group[,*count*]:*test-list*	×	√	×	√ 1
:group[,*count*]:*file*	×	×	∂ 2	√ 1
:group[,*count*]: *test-case* [. . .] :endgroup:	×	×	∂ 2	√
:repeat,*count*:*test-list*	×	√	×	√ 1
:repeat,*count*:*file*	×	×	√	√ 1
:repeat,*count*: *test-case* [. . .] :endrepeat:	×	×	√	√
:random:*test-list*	×	√	×	√ 1
:random:*file*	×	×	×	√ 1
:random: *test-case* [. . .] :endrandom:	×	×	×	√
:timed_loop,*seconds*:*test-list*	×	√	×	√ 1
:timed_loop,*seconds*:*file*	×	×	×	√ 1
:timed_loop,*seconds*: *test-case* [. . .] :endtimed_loop:	×	×	×	√
:remote,*nnn*[,. . .]:*test-list*	×	×	×	√ 1
:remote,*nnn*[,. . .]:*file*	×	×	√	√ 1
:remote,*nnn*[,. . .]: *test-case* [. . .] :endremote:	×	×	√	√
:distributed,*nnn*[,. . .]:*test-list*	×	×	×	√ 1
:distributed,*nnn*[,. . .]:*file*	×	×	×	√ 1
:distributed,*nnn*[,. . .]: *test-case* [. . .] :enddistributed:	×	×	×	√
:variable,*name*=*value*[,. . .]:*test-list*	×	×	×	√ 1
:variable,*name*=*value*[,. . .]:*file*	×	×	×	√ 1
:variable,*name*=*value*[,. . .]: *test-case* [. . .] :endvariable:	×	×	×	√
:*group-execution*[;. . .]:*test-list*	×	√	×	√ 1
:*group-execution*[;. . .]:*file*	×	×	×	√ 1

Notes:

1. Where there is a conflict between dTET2 and ETET syntax for these directives, `tcc` will determine which syntax to accept by examining the value of the `TET_COMPAT` configuration variable. An error will occur if `TET_COMPAT` is not set, or set to an incorrect value.
2. The optional `count` argument is not supported.

D. Comparison tables for C API interfaces

D.1 Introduction

This appendix contains tables which indicate the status of each C API function and variable in several contexts. Some of the information in these tables is provisional and will be finalised during the course of the development of TETware.

The meanings of symbols which appear in these tables are as follows:

√	function or variable is fully supported as described in the specification
∂	function or variable is partially supported
×	function or variable is not supported
○	function or variable is present but has no effect
1 2 etc.	refer to notes at the end of each table table

D.2 Support for interfaces in different TETware versions

Function or variable name	TETware	TETware-Lite
Test case structure and management		
tet_testlist[]	√	√
(*tet_startup)()	√	√
(*tet_cleanup)()	√	√
tet_thistest	√	√
tet_nosigreset	√	√
tet_pname	√	√
Making journal entries		
tet_setcontext()	√	√
tet_setblock()	√	√
tet_infoline()	√	√
tet_minfoline()	√	√
tet_printf()	√	√
tet_vprintf()	√	√
tet_result()	√	√
Cancelling test purposes		
tet_delete()	√	√
tet_reason()	√	√
Manipulating configuration variables		
tet_getvar()	√	√
Generating and executing processes		
tet_fork()	√	√
tet_fork1()	√ 1	√ 1
tet_exec()	√	√
tet_child	√	√
tet_spawn()	√	√
tet_wait()	√	√
tet_kill()	√	√
Executed process functions		
tet_main()	√	√
tet_exit()	√	√ 2
tet_logoff()	√	○
tet_thistest	√	√
tet_pname	√	√
Test case synchronisation		
tet_sync()	√	×
tet_msync()	√	×
tet_remsync()	√	×
(*tet_syncerr)()	√	×
tet_syncreport()	√	×
Remote system designations		
tet_remgetlist()	√	√ 3 4
tet_remgetsys()	√	√ 3
Remote process control		
tet_remexec()	√ 5	×
tet_remwait()	√ 5 6	×
tet_remkill()	√ 5	×
Error reporting		
tet_errno	√	√
tet_errlist[]	√	√
tet_nerr	√	√
Remote system information		
tet_remtime()	√	×
tet_getsysbyid()	√	×
Thread control		
tet_thr_create()	√ 1	√ 1

Notes:

1. Only present in the thread-safe API library on Unix-like systems.
2. Equivalent to calling `exit()`.
3. Always returns zero.
4. System name list always contains a single entry for system zero.
5. The use of these functions is discouraged. The required processing should instead be performed by the part of the test case that is executing on the remote system.
6. A call to this function may have undesirable side effects in a multi-threaded environment.

D.3 Support for interfaces on different platforms

Function or variable name	Platform		
	Unix POSIX.1	DOS Windows	Windows NT
Test case structure and management			
`tet_testlist[]`	√	√	√
`(*tet_startup)()`	√	√	√
`(*tet_cleanup)()`	√	√	√
`tet_thistest`	√	√	√
`tet_nosigreset`	√	○	√
`tet_pname`	√	√	√
Making journal entries			
`tet_setcontext()`	√	√	√
`tet_setblock()`	√	√	√
`tet_infoline()`	√	√	√
`tet_minfoline()`	√	√	√
`tet_printf()`	√	√	√
`tet_vprintf()`	√	√	√
`tet_result()`	√	√	√
Cancelling test purposes			
`tet_delete()`	√	√	√
`tet_reason()`	√	√	√
Manipulating configuration variables			
`tet_getvar()`	√	√	√
Generating and executing processes			
`tet_fork()`	√	×	×
`tet_fork1()`	√ 1	×	×
`tet_exec()`	√	×	×
`tet_child`	√	○	√
`tet_spawn()`	√	√ 2	√
`tet_wait()`	√	○2	√
`tet_kill()`	√	○	∂ 3
Executed process functions			
`tet_main()`	√	∂ 4	√
`tet_exit()`	√	√	√
`tet_logoff()`	√	√	√
`tet_thistest`	√	√	√
`tet_pname`	√	√	√
Test case synchronisation			
`tet_sync()`	√	√	√
`tet_msync()`	√	√	√
`tet_remsync()`	√	√	√
`(*tet_syncerr)()`	√	√	√
`tet_syncreport()`	√	√	√
Remote system designations			
`tet_remgetlist()`	√	√	√
`tet_remgetsys()`	√	√	√
Remote process control			
`tet_remexec()`	√ 5 6	√ 5 6	√ 5 6
`tet_remwait()`	√ 5 6 7	√ 5 6	√ 5 6
`tet_remkill()`	√ 5 6	√ 5 6	√ 5 6
Error reporting			
`tet_errno`	√	√	√
`tet_errlist[]`	√	√	√
`tet_nerr`	√	√	√
Remote system information			
`tet_remtime()`	√ 5	√ 5	√ 5
`tet_getsysbyid()`	√	√	√
Thread control			
`tet_thr_create()`	√ 1	×	×

Notes:

1. Only present in the thread-safe API library.
2. Execution of the calling process is suspended until the new process terminates.
3. The `sig` parameter is ignored.
4. Processes launched by `tet_remexec()` are not supported.
5. A call to this function will fail if the remote system does not support multi-tasking.
6. The use of these functions is discouraged. The required processing should instead be performed by the part of the test case that is executing on the remote system.
7. A call to this function may have undesirable side effects in a multi-threaded environment.

D.4 Compatibility with interfaces in existing TET implementations

Function or variable name	TET implementation			
	TET 1.10	ETET 1.10.3	dTET2 2.3	TETware
Test case structure and management				
`tet_testlist[]`	√	√	√	√
`(*tet_startup)()`	√	√	√	√
`(*tet_cleanup)()`	√	√	√	√
`tet_thistest`	√	√	√	√
`tet_nosigreset`	√	√	√	√
`tet_pname`	√	√	√	√
Making journal entries				
`tet_setcontext()`	√	√	√	√
`tet_setblock()`	√	√	√	√
`tet_infoline()`	√	√	√	√
`tet_minfoline()`	×	×	×	√
`tet_printf()`	×	×	×	√
`tet_vprintf()`	×	×	×	√
`tet_result()`	√	√	√	√
Cancelling test purposes				
`tet_delete()`	√	√	√	√
`tet_reason()`	√	√	√	√
Manipulating configuration variables				
`tet_getvar()`	√	√	√	√
Generating and executing processes				
`tet_fork()`	√	√	√	√ 1
`tet_fork1()`	×	×	×	√ 2
`tet_exec()`	√	√	√	√ 1
`tet_child`	√	√	√	√
`tet_spawn()`	×	×	×	√
`tet_wait()`	×	×	×	√
`tet_kill()`	×	×	×	√
Executed process functions				
`tet_main()`	√	√	√	√
`tet_exit()`	×	×	√	√
`tet_logoff()`	×	×	√	√
`tet_thistest`	√	√	√	√
`tet_pname`	√	√	√	√
Test case synchronisation				
`tet_sync()`	×	×	√	√ 3
`tet_msync()`	×	×	√	√ 3
`tet_remsync()`	×	×	×	√
`(*tet_syncerr)()`	×	×	×	√
`tet_syncreport()`	×	×	×	√
Remote system designations				
`tet_remgetlist()`	×	×	√	√
`tet_remgetsys()`	×	×	√	√
Remote process control				
`tet_remexec()`	×	×	√ 4	√ 4
`tet_remwait()`	×	×	√ 4	√ 4 5
`tet_remkill()`	×	×	√ 4	√ 4
Error reporting				
`tet_errno`	×	×	×	√
`tet_errlist[]`	×	×	×	√
`tet_nerr`	×	×	×	√
Remote system information				
`tet_remtime()`	×	×	×	√
`tet_getsysbyid()`	×	×	×	√
Thread control				
`tet_thr_create()`	×	×	×	√ 2

Notes:

1. Only supported on Unix-like systems. Use `tet_spawn()` in order to ensure portability to all systems.
2. Only present in the thread-safe API library on Unix-like systems.
3. Provided for backwards compatibility with existing dTET2 test cases. New test cases should use `tet_remsync()` instead.
4. The use of these functions is discouraged. The required processing should instead be performed by the part of the test case that is executing on the remote system.
5. A call to this function may have undesirable side effects in a multi-threaded environment.

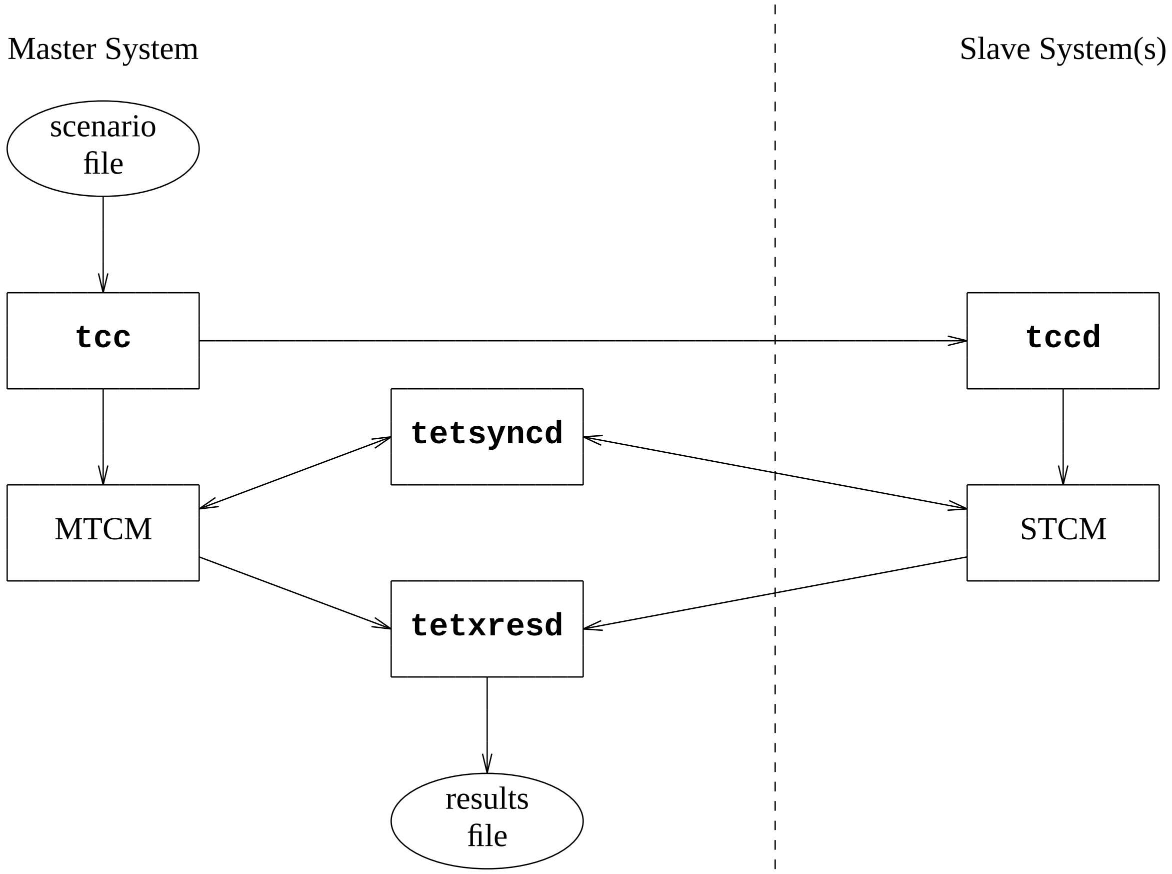
E. dTET2 architecture

E.1 Origin

The diagram in this appendix is reproduced from the dTET2 Installation and User Guide. It is presented here in order to assist readers in understanding the architecture and component names used in the current dTET2 implementation.

E.2 Diagram and component definitions

The following diagram provides a simplified view of how the different dTET2 components relate to each other:



tcc	– dTET2 Test Case Controller
tccd	– Test Case Controller daemon
tetsyncd	– Synchronisation daemon
tetxresd	– Execution results daemon
MTCM	– Master Test Case Manager + master test case parts
STCM	– Slave Test Case Manager + slave test case parts

F. C language binding

F.1 Origin

The body of this appendix is taken from the section entitled ‘‘C language binding’’ in the dTET2 Programmers Guide. It is reproduced here for ease of reference and for use in conjunction with the section entitled ‘‘The C API’’ elsewhere in this document.

Please note that references in this appendix to other sections and chapters refer to sections and chapters in the dTET2 Programmers Guide and not to parts of this document.

F.2 Introduction

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

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

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

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

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

F.3 Test case structure and management functions

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

Synopsis

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

struct tet_testlist tet_testlist[];

void (*tet_startup)(void);

void (*tet_cleanup)(void);

int tet_thistest;

int tet_nosigreset;

char *tet_pname;
```

Description

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

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

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

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

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

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

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

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

F.4 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.

F.5 Making journal entries

These functions support functionality used in creating journal entries.

Synopsis

```
void tet_setcontext(void);
void tet_setblock(void);
void tet_infoline(char *data);
void tet_result(int result);
```

Description

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

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

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

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

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

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

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

F.6 Canceling test purposes

These functions support functionality used to cancel test purposes.

Synopsis

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

char *tet_reason(int testno);
```

Description

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

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

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

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

F.7 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.

F.8 Generating and executing processes

These functions support forking new child processes.

Synopsis

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

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

extern pid_t tet_child;
```

Description

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

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

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

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

F.9 Executed process functions

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

Synopsis

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

void tet_exit(int status);

void tet_logoff(void);

int tet_thistest;

char *tet_pname;
```

Description

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

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

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

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

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

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

F.10 Test case synchronisation

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

Synopsis

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

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

Description

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

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

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

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

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

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

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

- More than `waittime` seconds elapse between synchronisation requests from participating systems.
- A related synchronisation request times out on one of the other participating systems.
- The user-supplied function in a test case on one of the other participating systems returns control to its TCM before synchronising.
- The sync point specified by `syncptno` has already occurred.
- The calling process is running on the master system and `syncnames` is `NULL` or points to an empty system ID list.
- A system ID appears more than once in the array pointed to by `syncnames`.
- An invalid parameter is specified in the call.
- The API encounters a problem while processing the request.

`tet_msync()` operates in the same way as does `tet_sync()`, with the additional facility of enabling systems to exchange sync message data during a successful call. One participating system may send sync message data which will be made available to the other systems when the call returns.

`tet_msync()` takes an additional `msgp` argument which points to a `tet_synmsg` structure (as defined in `<dtet2/tet_api.h>`). This structure contains the following elements:

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

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

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

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

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

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

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

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

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

i. Decide from which system to accept data and redesignate the other sending systems as receiving systems.

ii. Process the redesignated systems as described above.

iii. Clear the TET_SMSNDMSG bit and set the TET_SMRCVMSG bit in tsm_flags on the redesignated systems.

iv. Set the TET_SMDUP bit in tsm_flags on all systems.

If a system tries to send a message which is larger than the maximum permitted message size (as defined by the value TET_SMMSGMAX in <dtet2/tet_api.h>), the API perform the following actions:

i. Truncate the message to the maximum size before accepting it.

ii. Set the TET_SMTRUNC bit in tsm_flags on all systems.

If the call to tet_msync() is unsuccessful, the values of members of the tet_synmsg structure are undefined when the call returns.

If a system calls tet_msync() with a msgp of NULL, the API regards it as a receiving system but does not return any message data to it. Thus a call to tet_msync() with a msgp of NULL is functionally equivalent to calling tet_sync().

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

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

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

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

F.11 Remote system designations

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

Synopsis

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

int tet_remgetsys (void);
```

Description

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

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

F.12 Remote process control

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

Synopsis

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

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

int tet_remkill(int remoteid);
```

Description

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

G. Test case synchronisation

G.1 Origin

The body of this appendix is taken from the chapter entitled ‘‘Test case synchronisation’’ in the dTET2 Installation and User Guide. It is reproduced here in order to assist readers in understanding the concepts and processing involved when parts of a distributed test case synchronise with each other.

G.2 Introduction

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

G.3 Synchronisation request concepts

G.3.1 Request types

There are two types of synchronisation performed by dTET2 processes. Automatic synchronisation requests are generated when Test Case Managers synchronise with each other at certain pre-defined points during test case execution. User synchronisation requests are generated when different parts of a distributed test purpose call the `tet_sync()` or `tet_msync()` API library routines.

G.3.2 Request parameters

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

Processes on systems which want to synchronise with each other send requests to the dTET2 Synchronisation daemon (`tetsyncd`). `tetsyncd` waits until all systems have submitted their requests and then notifies each participating process of the result.

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

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

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

G.3.3 Sync events

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

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

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

G.3.4 Sync states

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

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

`SYNC-YES` The system has synchronised with a `yes` sync vote.

`SYNC-NO` The system has synchronised with a `no` vote.

`NOT-SYNCED` The system has not yet participated in this sync event.

`TIMED-OUT` The system has synchronised but the associated timeout has expired before the sync event completed.

`DEAD` The system has synchronised but the participating process has disconnected from `tetsyncd` before the sync event completed.

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

G.3.5 With what to synchronise?

As indicated above, when a dTET2 process makes a synchronisation request, it specifies a list of **system IDs** with which is wishes to synchronise. This means that any one dTET2 process running on a particular system can participate in a sync event on behalf of that system. It is not possible for a process to use dTET2 synchronisation facilities to synchronise with a particular process on a named system, or for processes on the same system to use these facilities to synchronise with each other.[27]

G.4 Automatic synchronisation requests

G.4.1 Description

Automatic synchronisation requests are generated by the dTET2 Test Case Manager, and by the API when a remote executed process is started. The list of systems that are expected to participate in automatic sync events for each distributed test case is defined before the first request is made. Each automatic synchronisation request is accompanied by a **sync ID** which identifies this list of systems. Processes which make automatic synchronisation requests do not send or receive sync message data.

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

G.4.2 Test case manager synchronisation

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

27. Note that the term **system** refers to a logical system ID, not to a physical machine. Therefore, it is possible for two or more co-operating processes with different system IDs running on the same physical machine to use dTET2 synchronisation facilities to synchronise with each other.

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

Stage in test case processing	Sync point number[28]	Timeout (seconds)
At TCM startup time	1	60
Before the startup function (if any) is called	2	60
At the start of each invocable component	$ICno * 2^{16}$	60
At the start of each test purpose	$(ICno * 2^{16}) + (TPno * 2)$	60
At the end of each test purpose	$(ICno * 2^{16}) + (TPno * 2) + 1$	600
Before the cleanup function (if any) is called	$((ICcount + 1) * 2^{16}) + 2$	60

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

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

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

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

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

G.4.3 Remote executed process synchronisation

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

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

---

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

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

G.4.4 Error handling

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

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

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

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

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

G.4.5 Example error messages

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

G.4.5.1 Example 1

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

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

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

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

G.4.5.2 Example 2

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

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

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

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

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

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

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

This indicates that the master TCM has missed the sync event because the event has already completed.

G.5 User synchronisation requests

G.5.1 Description

A user synchronisation request is generated when a test purpose in a distributed test case calls the `tet_sync()` or `tet_msync()` API library routines. The sync point number, system ID list and timeout are specified in each call. The sync vote is always `yes` for user synchronisation requests. In addition to these parameters, a process can send, or indicate willingness to receive, sync message data by calling `tet_msync()` instead of `tet_sync()`. When this is done and all participating systems use the same sync point number, message data sent by the sending system is returned to the receiving systems on successful completion of the event. Apart from this distinction, everything in the description of `tet_sync()` that follows applies equally to `tet_msync()`.

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

The dTET2 specification requires that all user synchronisation requests include the master part of a distributed test case. The `tet_sync()` function automatically includes the system ID of the master system in the system ID list associated with each user synchronisation request. Therefore, when a distributed test purpose part makes a call to `tet_sync()`, it does not itself have to include the system ID of the master system in the accompanying system ID list.

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

G.5.2 Error handling

Synchronisation request failures and synchronisation errors for user synchronisation requests are defined in the same way as for automatic synchronisation requests. The API prints a test case manager message to the journal file when a user synchronisation request is unsuccessful.

Each diagnostic indicates the sync point number of the request that was unsuccessful and the system IDs and sync states of the systems which failed to synchronise or timed out. However, the formats of diagnostics printed to the journal file are different from those generated for unsuccessful automatic synchronisation requests. Examples of the formats that may be used to report an unsuccessful user synchronisation request are presented in the next section.

G.5.3 Example error messages

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

G.5.3.1 Example 1

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

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

G.5.3.2 Example 2

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

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

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

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

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

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

```
system 0: tet_sync() failed, syncptno = 12, event already happened
```

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

G.5.3.3 Example 3

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

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

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

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

H. Server interface functions

H.1 Origin

This appendix describes interface functions for use by clients when communicating with dTET2 servers. The body of this appendix is taken from a project document which was used by members of the original dTET2 development team.

H.2 Introduction

The functions described here are internal interfaces for use within dTET2 processes, and are not available for use by test cases. They are not part of the API.

These descriptions are presented here to help readers understand the internal workings of the existing dTET2 implementation. It is not guaranteed that these interfaces will exist in any future TET implementation.

H.3 SYNCD functions

```
int sd_start()
```

Start a SYNCD process. Return `0` if successful or `−1` on error.

This function must be called by the MTCC if SYNCD services are required.

```
int sd_logon()
```

Log on to the SYNCD, making a connection if necessary. Return `0` if successful or `−1` on error.

```
int sd_logoff(stayopen)
int stayopen;
```

Log off from the SYNCD. Return `0` if successful or `−1` on error.

This function may be called even if the caller is not currently logged on to the SYNCD (e.g., from an error or cleanup function). If *stayopen* is non-zero, the existing connection is retained so that a subsequent `sd_logon()` call will not need to make a new connection.

```
long sd_snget()
```

Get a sync ID from the SYNCD for use in sync requests. Return the sync ID if successful or `−1` on error. The server reply code is available in `sd_errno`.

```
int sd_snsys(snid, snames, nsname)
long snid;
int *snames, nsname;
```

Assign a system name list to a sync ID. Return `0` if successful or `−1` on error. The server reply code is available in `sd_errno`.

This function specifies which systems should be expected to participate in auto-syncs associated with a particular sync ID. *snid* identifies the sync ID to which the system name list should be assigned. *snames* points to the start of the system name list. *nsname* specifies the number of system names in the list.

```
int sd_masync(snid, xrid, spno, vote, timeout, synreq, nsys)
long snid, xrid, spno;
int vote, timeout, *nsys;
struct synreq *synreq;
```

Perform an auto sync from the MTCM. Return `0` if successful or `−1` on error. The server reply code is available in `sd_errno`.

snid is a sync ID obtained by the MTCC from a previous `sd_snget()` call. *xrid* is an xres file ID obtained by the MTCC from a previous `xd_xropen()` call.

spno is the sync point number for this sync request; for auto-syncs at TP start and end, this should be generated by the `MK_ASPNO()` macro.

vote is the sync vote that the caller wishes to register; it should be either `SV_YES` to sync successfully or `SV_NO` to sync unsuccessfully.

timeout is the number of seconds to wait before the request times out. If *timeout* is zero, the request returns immediately; if the calling process is not the last one to register a sync vote, the effect is the same as if the request timeout had expired. If *timeout* is less than zero, no timeout is specified for the request.

synreq points to the first in an array of structures which may receive details of other processes' sync states on return. *nsys* points to a location containing the number of elements in the `synreq` array; on successful return, this location is updated to contain the number of other systems actually participating in the sync. *synreq* and/or *nsys* may be `NULL` if this information is not required.

Note that `sd_masync()` returns `0` even if the sync itself failed; the `−1` error return is used to indicate server or transport errors. However, if the sync fails because one or more processes vote `SV_NO`, time out or die, this is indicated by the value in `sd_errno`. More precise details of which process(es) caused the sync to fail in these ways can be obtained from information returned in the `synreq` array.

The MTCM should use `doasync()` to access this function.

```
int sd_sasync(snid, spno, vote, timeout, synreq, nsys, xrid)
long snid, spno, *xrid;
int vote, timeout, *nsys;
struct synreq *synreq;
```

Perform an auto sync from somewhere other than the MTCM. Return `0` if successful or `−1` on error. The server reply code is available in `sd_errno`.

This function behaves in the same way as `sd_masync()`, except that the xres ID specified in the corresponding MTCM `sd_masync()` call is returned in the location pointed to by *xrid*. This is the means by which the MTCM communicates its xres ID to all the STCMs.

The STCM should use `doasync()` to access this function.

```
int doasync(spno, vote, timeout, synreq, nsys)
long spno;
int vote, timeout, *nsys;
struct synreq *synreq;
```

Perform an auto sync from a MTCM or STCM. Return `0` if successful or `−1` on error. The server reply code is available in `sd_errno`. The meanings of the arguments are the same as for those used with `sd_masync()` and `sd_sasync()`.

This function is the preferred interface to `sd_masync()` or `sd_sasync()` for TCM processes, and handles the propagation of xres IDs between them. The sync ID used is that received from the TCM's parent process. Whether `doasync()` calls `sd_masync()` or `sd_sasync()` depends on the process type of the calling process.

Appropriate versions of this function are part of the MTCM, the STCM and processes that are started by `tet_remexec()`.

```
int sd_usync(snid, xrid, spno, vote, timeout, synreq, nsys)
long snid, xrid, spno;
int vote, timeout, nsys;
struct synreq *synreq;
```

Perform a user sync request. Return `0` if successful or `−1` on error. The server reply code is available in `sd_errno`.

This function provides support for the API `tet_sync()` function. The meanings of the arguments are the same as for `sd_masync()` except that:

- *xrid* is used to help identify the sync event and is not propagated to other processes
- *synreq* points to an array of user details structures specifying the other processes that are to participate in the sync event
- *nsys* specifies the number of elements in the `synreq` array

(Note that, unlike the `sd_masync()` *nsys* parameter, the *nsys* argument to `sd_usync()` is the value itself and not a pointer to the value.)

On successful return, the `synreq` array is updated to contain details of the other processes participating in the sync event.

Note that `sd_usync()` returns `0` even if the sync itself failed; the `−1` error return is used to indicate server or transport errors. However, if the sync fails because one or more processes vote `SV_NO`, time out or die, this is indicated by the value in `sd_errno`. More precise details of which process(es) caused the sync to fail in these ways can be obtained from information returned in the `synreq` array.

H.4 TCCD functions

```
int tc_logon(sysid)
int sysid;
```

Connect to a TCCD on a remote system and log on to it. Return `0` if successful or `−1` on error.

sysid identifies the remote system to which the calling process wishes to connect.

Each call to `tc_logon()` causes a slave TCCD to be generated on the specified remote system, to service subsequent TCCD requests from the calling process. However, a TCM may not be logged on more than once to a particular remote system.

```
int tc_logoff(sysid)
int sysid;
```

Log off from a TCCD and close the connection. Return `0` if successful or `−1` on error.

sysid specifies the system ID of the remote system.

A `SIGHUP` signal is sent to each unterminated child process on the remote system, then the slave TCCD exits.

This function may be called even if the caller is not currently logged on to a particular TCCD (e.g., from an error or cleanup function).

```
long tc_mexec(sysid, path, argv, outfile)
int sysid;
char *path, **argv, *outfile;
```

Execute a non-DTET process on a remote system. Return the process ID of the *exec*'d process if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *path* specifies the name of the file to execute. *argv* points to a null-terminated list of arguments to pass to the *exec*'d process.

If *outfile* is non-null, the *stdout* and *stderr* of the *exec*'d process are connected to *outfile*; otherwise they are connected to the TCCD log file. *stdin* is connected to `/dev/null`. All other file descriptors are closed.

The process is executed on the remote system by a call to `execvp()`, so as to allow normal `PATH` searching and shell-script execution to take place.

```
long tc_texec(sysid, path, argv, outfile, snid, xrid)
int sysid;
char *path, **argv, *outfile;
long snid, xrid;
```

Execute a TCM on a remote system. Return the process ID of the *exec*'d process if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *path* specifies the name of the file to execute. *argv* points to a null-terminated list of arguments to pass to the *exec*'d process. *snid* and *xrid* specify the sync ID and xres ID to pass to the *exec*'d process for use in SYNCD and XRESD calls. If the *exec* fails, the TCCD will register a `NO` sync vote on the sync ID specified by *snid*.

If *outfile* is non-null, the *stdout* and *stderr* of the *exec*'d process are connected to *outfile*; otherwise they are connected to the TCCD log file. *stdin* is connected to `/dev/null`. All other file descriptors are closed.

The process is executed on the remote system by a call to `execvp()`, so as to allow normal `PATH` searching and shell-script execution to take place.

```
long tc_uexec(sysid, path, argv, snid, xrid)
int sysid;
char *path, **argv;
long snid, xrid;
```

Execute a user process on a remote system. Return the process ID of the *exec*'d process if successful or `−1` on error. The server reply code is available in `tc_errno`.

This function provides support for the API `tet_remexec()` function. *sysid* specifies the system ID of the remote system. *path* specifies the name of the file to execute. *argv* points to a null-terminated list of arguments to pass to the *exec*'d process. *snid* and *xrid* specify the sync ID and xres ID to pass to the *exec*'d process for use in SYNCD and XRESD calls. If the *exec* fails, the TCCD will register a `NO` sync vote on the sync ID specified by *snid*.

The *stdout* and *stderr* of the *exec*'d process are connected to the TCCD log file. *stdin* is connected to `/dev/null`. All other file descriptors are closed.

The TCCD changes directory to `TET_EXECUTE` if it is specified, otherwise to `TET_ROOT`, before the *exec* takes place.

The process is executed on the remote system by a call to `execvp()`, so as to allow normal `PATH` searching and shell-script execution to take place.

```
int tc_kill(sysid, pid, signum)
int sysid, signum;
long pid;
```

Send a signal to a remote process. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *pid* specifies the process ID of the process that is to receive the signal. *signum* specifies the signal that is to be sent.

The symbolic signal name corresponding to *signum* must exist on both the local and the remote system. The signal actually sent to the remote process is the one associated with the same symbolic name on the remote system as that associated with *signum* on the local system.

```
int tc_wait(sysid, pid, timeout, statp)
int sysid, timeout, *statp;
long pid;
```

Wait for a remote process to terminate. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *pid* specifies the process ID of the process that is to be waited for. *timeout* specifies how long to wait for the remote process to terminate. If *timeout* is zero, the call returns immediately whether or not the remote process has terminated. If *timeout* is less than zero, the call will wait until either the process terminates or an error occurs. If the call is successful, the exit status of the remote process is returned in the location pointed to by *statp*.

```
int tc_sysname(sysid, snames, nsname)
int sysid, *snames, nsname;
```

Send system name list to TCCD. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *snames* points to the start of the system name list. *nsname* specifies the number of system names in the list.

```
int tc_cfname(sysid, cfname)
int sysid;
char *cfname;
```

Send a configuration file name to TCCD. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

cfname specifies the configuration file name that is to be used in a subsequent configuration variable exchange.

```
int tc_configv(sysid, lines, nline)
int sysid, nline;
char **lines;
```

Send merged configuration lines to TCCD. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *lines* points to the first in a list of pointers to configuration lines that are to be sent to TCCD. *nline* specifies the number of lines in the list.

The configuration lines are written to a temporary file on the remote system. The name of this file is assigned to the environment variable `TET_CONFIG`. The file is removed automatically when the slave TCCD exits.

This function should be called only once to send merged configuration lines to a particular slave TCCD; it will generate as many requests as are required to send all the lines to the server.

```
int tc_sndconfv(sysid, lines, nline)
int sysid, nline;
char **lines;
```

Send configuration lines to TCCD as part of a configuration variable exchange. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *lines* points to the first in a list of pointers to configuration lines that are to be sent to TCCD. *nline* specifies the number of lines in the list.

This function should be called only once in a particular configuration variable exchange; it will generate as many requests as are required to send all the lines to the server.

```
char **tc_rcvconfv(sysid, nlines, done)
int sysid, *nlines, *done;
```

Receive merged configuration lines from TCCD as part of a configuration variable exchange. Return a pointer to the first in a list of pointers to configuration lines if successful, or `NULL` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system.

If the call is successful:

- The number of lines in the list is returned in the location pointed to by *nlines*.
- A flag is returned in the location pointed to by *done*, whose value is `0` or `1` depending on whether or not there are any more lines to be returned.

This function should be called repeatedly until **done* is true or an error occurs.

The list of pointers and their associated strings are stored in memory owned by the `tc_talk` subsystem, so they should be copied if required before the next request is sent to the same TCCD.

```
int tc_putenv(sysid, env)
int sysid;
char *env;
```

Add a single environment variable assignment string to the TCCD environment on a remote system (as if `putenv()` had been called on the remote system). Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *env* specifies the assignment string that is to be added to the remote environment.

```
int tc_putenvv(sysid, envp, nenv)
int sysid, nenv;
char **envp;
```

Add one or more environment variables to the TCCD environment on a remote system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *envp* points to the first of a list of pointers to environment strings that are to be added to the remote environment. *nenv* specifies the number of strings in the list.

```
int tc_access(sysid, path, mode)
int sysid, mode;
char *path;
```

Determine the accessibility of a file on a remote system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. The access permissions of *path* are checked with respect to the server's effective user ID and group ID. *mode* is a bit field which determines which access permissions are to be checked; the meanings of each bit is a follows:

`04` check read permission
`02` check write permission
`01` check execute/search permission
`00` check existence of file

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

```
int tc_mkdir(sysid, dir)
int sysid;
char *dir;
```

Make a directory on a remote system. Return `0` if successful or `−1` on error.

The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *dir* specifies the name of the directory that is to be created.

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

```
int tc_rmdir(sysid, dir)
int sysid;
char *dir;
```

Remove a directory on a remote system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *dir* specifies the name of the directory that is to be removed.

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

```
int tc_chdir(sysid, dir)
int sysid;
char *dir;
```

Instruct TCCD to change directory on a remote system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *dir* specifies the name of the new working directory.

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

```
char *tc_mktmpdir(sysid, prefix)
int sysid;
char *prefix;
```

Make a unique temporary directory on a remote system. Return a pointer to the new directory name if successful or `NULL` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *prefix* specifies the prefix for the temporary directory name.

The name of the new directory created by TCCD is *prefix*`/`*NNNNNx* where *NNNNN* is the process ID of the TCCD and *x* is a unique letter.

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

The return value points to memory owned by the `tc_talk` subsystem whose contents should be copied if required before the next request is sent to the same TCCD.

```
int tc_unlink(sysid, file)
int sysid;
char *file;
```

Unlink a file on a remote system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *file* specifies the name of the file to be unlinked.

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

```
int tc_fopen(sysid, fname)
int sysid;
char *fname;
```

Open a file for writing on a remote system. Return the file ID if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *fname* specifies the name of the file that is to be opened.

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

```
int tc_puts(sysid, fid, line)
int sysid, fid;
char *line;
```

Write a single line to a file on a remote system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *fid* is a file ID returned by a previous call to `tc_fopen()`. The string pointed to by *line* is written to the file, followed by a newline.

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

```
int tc_putsv(sysid, fid, lines, nline)
int sysid, fid, nline;
char **lines;
```

Write one or more lines to a file on a remote system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *fid* is a file ID returned by a previous call to `tc_fopen()`. *lines* points to the first of a list of pointers to strings which are to be written to the file, each followed by a newline. *nline* specifies the number of strings in the list.

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

```
int tc_fclose(sysid, fid)
int sysid, fid;
```

Close a file on a remote system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *fid* is a file ID returned by a previous call to `tc_fopen()`.

```
int tc_lockfile(sysid, file, timeout)
int sysid, timeout;
char *file;
```

Create an exclusive lock on a remote system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *file* specifies the path name of a lock file which is to be created on the remote system. If *timeout* is greater than zero, repeated attempts are made to create the lock file until either the operation is successful, the timeout expires or an error occurs. If *timeout* is zero, the call will return after the first attempt to create the file, whether or not the operation is successful. If *timeout* is less than zero, the call will wait until either the lock file can be created or an error occurs.

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

```
char *tc_sharelock(sysid, lockdir, timeout)
int sysid, timeout;
char *lockdir;
```

Create a non-exclusive lock on a remote system. Return a pointer to the name of the created lock file if successful or `NULL` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *lockdir* specifies the path name of a lock directory on the remote system, which is created if necessary. If *lockdir* can be created or exists already, a file is created in that directory whose name is *lockdir*`/`*NNNNNx* where *NNNNN* is the process ID of the calling process and *x* is a unique letter. If *timeout* is greater than zero and *lockdir* exists but is not a directory, repeated attempts are made to create the lock directory until either the operation is successful, the timeout expires or an error occurs. If *timeout* is zero, the call will return after the first attempt to create the lock directory, whether or not the operation is successful. If *timeout* is less than zero, the call will wait until either the lock directory can be created or an error occurs.

If the call fails because the remote system call failed, `errno` is set to the local equivalent of the remote system's `errno` value (if possible); otherwise, `errno` is set to `0`.

The return value points to memory owned by the `tc_talk` subsystem whose contents should be copied if required before the next request is sent to the same TCCD.

```
int tc_rxfile(sysid, fromfile, tofile)
int sysid;
char *fromfile, *tofile;
```

Cause TCCD to transfer a file to the master system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. TCCD transfers the file specified by *fromfile* to the path specified by *tofile*, interpreted relative to the *saved files* directory on the master system. (See the description of `xd_xfile()` for more details).

```
char *tc_mksdir(sysid, dir, suffix)
int sysid;
char *dir, *suffix;
```

Make a *saved files* directory on a remote system. Return a pointer to the directory name if successful or `NULL` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *suffix* specifies the suffix for the name of the *saved files* directory and should be one or more of the characters `bec`, chosen in accordance with existing TCC conventions.

TCCD makes a *saved files* directory whose name is *dir*`/`*NNNNsuffix*. The directory specified by *dir* should already exist. *NNNN* is an ascending sequence number generated by TCCD.

The return value points to memory owned by the `tc_talk` subsystem whose contents should be copied if required before the next request is sent to the same TCCD.

```
int tc_tslfiles(sysid, files, nfile, subdir)
int sysid, nfile;
char **files, *subdir;
```

Copy *save files* locally on a remote system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *files* points to the first in a list of pointers to file or directory names. *nfile* specifies the number of file names in the list.

If a file matching one of these names is found, it is copied to *subdir* in the *saved files* directory on the remote system, or directly to the *saved files* directory if *subdir* is `NULL`. If a directory matching one of these names is found, this action is performed recursively for all files below that directory.

```
int tc_tsmfiles(sysid, files, nfile, subdir)
int sysid, nfile;
char **files, *subdir;
```

Copy *save files* from a remote system to the master system. Return `0` if successful or `−1` on error. The server reply code is available in `tc_errno`.

sysid specifies the system ID of the remote system. *files* points to the first in a list of pointers to file or directory names. *nfile* specifies the number of file names in the list. If a file matching one of these names is found, it is copied (by means of an `xd_xfile()` call) to *subdir* in the *saved files* directory on the master system, or directly to the *saved files* directory if *subdir* is `NULL`. If a directory matching one of these names is found, this action is performed recursively for all files below that directory.

H.5 XRESD functions

```
int xd_start(savedir)
char *savedir;
```

Start an XRESD; Return `0` if successful or `−1` on error.

This function must be called by the MTCC if XRESD services are required. *savedir* specifies the full path name of the directory below which *saved files* are to be placed.

```
int xd_logon()
```

Connect to the XRESD and log on to it. Return `0` if successful or `−1` on error.

```
int xd_logoff()
```

Log off from the XRESD and close the connection. Return `0` if successful or `−1` on error.

This function may be called even if the caller is not currently logged on to the XRESD (e.g., from an error or cleanup function).

```
int xd_xfile(fromfile, tofile)
char *fromfile, *tofile;
```

Copy a *save file* to the master system. Return `0` if successful or `−1` on error. The server reply code is available in `xd_errno`.

The file specified by *fromfile* is copied to the path specified by *tofile*, interpreted relative to the *saved files* directory on the master system. Directories below the *saved files* directory will be made as necessary.

This function may be called by a TCC or a TCM.

The location of the *saved files* directory is specified when XRESD is started, by a parameter to `xd_start()`.

```
long xd_xropen(xfname)
char *xfname;

extern long Xrid;
```

Open an execution results file on the master system. Return the xres ID of the open file if successful or `−1` on error. The server reply code is available in `xd_errno`.

xfname specifies the name of the execution results file to be opened.

This function should be called by the MTCM, and the return value should be stored in the global variable `Xrid` for subsequent use by `doasync()`.

```
int xd_xrsys(xrid, snames, nsname)
long xrid;
int *snames, nsname;
```

Assign a system name list to an execution results file. Return `0` if successful or `−1` on error. The server reply code is available in `xd_errno`.

This function is called by the MTCM and informs XRESD of the systems from which execution results are to be expected. *xrid* is an xres ID obtained by the MTCM from a call to `xd_xropen()`. *snames* points to the start of the system name list. *nsname* specifies the number of system names in the list.

```
int xd_icstart(xrid, icno, activity, tpcount)
long xrid, activity;
int icno, tpcount;
```

Signal IC start to XRESD. Return `0` if successful or `−1` on error. The server reply code is available in `xd_errno`.

xrid is an xres ID obtained by the MTCM from a call to `xd_xropen()`. *icno* specifies the number of the IC that is about to be started. *activity* specifies the TCC activity number that is to appear in certain results file messages. *tpcount* specifies the expected number of TPs in this IC.

This function should be called by the MTCM after all the TCMs have synced at the start of an IC.

```
int xd_icend(xrid)
long xrid;
```

Signal IC end to XRESD. Return `0` if successful or `−1` on error. The server reply code is available in `xd_errno`.

xrid is an xres ID obtained by the MTCM from a call to `xd_xropen()`.

This function should be called by the MTCM after all the TCMs have synced at the end of an IC.

```
int xd_tpstart(xrid, tpno)
long xrid;
int tpno;
```

Signal TP start to XRESD. Return `0` if successful or `−1` on error. The server reply code is available in `xd_errno`.

xrid is an xres ID obtained by the MTCM from a call to `xd_xropen()`. *tpno* specifies the number of the TP that is about to be started.

This function should be called by the MTCM after all the TCMs have synced at the start of a TP.

```
int xd_tpend(xrid)
long xrid;
```

Signal TP end to XRESD. Return `0` if successful or `−1` on error. The server reply code is available in `xd_errno`.

xrid is an xres ID obtained by the MTCM from a call to `xd_xropen()`.

This function should be called by the MTCM after all the TCMs have synced at the end of a TP.

```
int xd_xres(xrid, line)
long xrid;
char *line;
```

Send a single text line to the execution results file. Return `0` if successful or `−1` on error. The server reply code is available in `xd_errno`.

xrid is an xres ID obtained by the MTCM from a call to `xd_xropen()`. *line* specifies the line that is to be written to the execution results file.

```
int xd_xresv(xrid, lines, nlines)
long xrid;
char **lines;
int nlines;
```

Send one or more text lines to the execution results file. Return `0` if successful or `−1` on error. The server reply code is available in `xd_errno`.

xrid is an xres ID obtained by the MTCM from a call to `xd_xropen()`. *lines* points to the first of an array of pointers to xres lines. *nlines* specifies the number of xres lines in the list.

```
int xd_result(xrid, result)
long xrid;
int result;
```

Send a TP result code to XRESD. Return `0` if successful or `−1` on error. The server reply code is available in `xd_errno`.

xrid is an xres ID obtained by the MTCM from a call to `xd_xropen()`. *result* specifies the TP result code that is to be registered with XRESD.

A TCM on each participating system is expected to submit at least one such code before the MTCM signals TP end.

```
int xd_cfname(ecfname, dcfname, ccfname)
char *ecfname, *dcfname, *ccfname;
```

Send the master configuration file names to XRESD. Return `0` if successful or `−1` on error. The server reply code is available in `xd_errno`.

ecfname specifies the name of the master execute mode configuration file. *dcfname* specifies the name of the master distributed configuration file. *ccfname* specifies the name of a file containing configuration variable assignments specified on the MTCC command line.

The contents of these files will be used by TCM processes as the source of master configuration information when performing a configuration variable exchange as part of a `tet_remexec()` call.

```
char **xd_rcfname()
```

Return a pointer to the first in a list pointers to master configuration file names previously registered with XRESD by an `xd_cfname()` call, or `NULL` on error. The server reply code is available in `xd_errno`.

The list of file names pointed to by the return value contains `XD_NCFNAME` items.

The list of pointers and their associated strings are stored in memory owned by the `xd_talk` subsystem, so they should be copied if required before the next request is sent to XRESD.

This function is called by `tet_remexec()` before performing a configuration variable exchange with the target system.

CONTENTS