Distributed Test Environment Toolkit Version 2

dTET2 Installation and User Guide
Revision 1.0
DT2-UGDE-1.0

Released: 8th October 1993

X/Open Company Limited

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

1. Introduction

1.1 Preface

This document is the Installation and User Guide for the **Distributed Test Environment Toolkit Version 2** implementation, by X/Open, in ‘C’ for POSIX.1 systems with networking.

Much of dTET2 is based on Release 1.10 of the Test Environment Toolkit (TET) implementation by OSF, UI and X/Open, and on Release 1.0b of the Distributed Test Environment Toolkit (DTET) implementation by X/Open.

The Distributed Test Environment Toolkit Version 2 (dTET2) is one of a set of extensions to the TET by X/Open.

1.2 Audience

This document is intended to be read by systems administrators who will install dTET2 on their computer systems, and by software testing engineers who will use dTET2 to run verification test suites.

Test suite authors should refer to the dTET2 Programmers Guide for information about how to use the dTET2 Application Program Interface.

1.3 Conventions used in this guide

The following typographic conventions are used throughout this guide:

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

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

1.4 Related documents

Refer to the following documents for additional information about the Distributed Test Environment Toolkit Version 2:

- *Distributed Test Environment Toolkit Version 2: Architectural, Functional and Interface Specification*
- *Distributed Test Environment Toolkit Version 2: dTET2 Programmers Guide*

In addition, the dTET2 Release Notes contain important information about how to install and use dTET2. You should read the release notes thoroughly before attempting to install and use each new release of dTET2.

1.5 Problem reporting

If you encounter a problem while installing and using dTET2, you can send a support request by electronic mail to the address given in the dTET2 Release Notes. Please follow the instructions contained in the release notes about how to submit such a request; in particular, please be sure to include all the information asked for by these instructions when submitting the request.

2. Overview of the Distributed Test Environment Toolkit Version 2

2.1 Introduction

The purpose of the Distributed Test Environment Toolkit Version 2 (dTET2) is to provide a uniform framework, or test scaffold, into which both non-distributed and distributed test suites can be incorporated. By providing such a scaffold, test suites from different vendors can share a common interface, allowing for, among other things, ease of portability.

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

2.2 dTET2 components

This implementation of dTET2 consists of:

- A test case controller providing support for the building, execution, and clean-up of test scenarios (`tcc`).
- Server processes to undertake synchronisation and execution results processing (`tetsyncd` and `tetxresd`), and remote system processing (`tccd`).
- A 'C' Application Programming Interface library (`libapi.a`) and partially linked object modules for inclusion in the various parts of non-distributed and distributed test cases (`tcm.o`, `tcmchild.o` and `tcmrem.o`).
- A Shell Application Programming Interface (`tetapi.sh`) and Test Case Manager (`tcm.sh`).

The dTET2 'C' Application Programming Interface (API) consists of a test case manager and an API library which are used in both non-distributed and distributed test cases to handle interfacing between the test case parts and the rest of the dTET2 system.

The dTET2 Shell API consists of a test case manager and API support routines which may be used with non-distributed test cases written in XPG3 Shell. There is no support for distributed Shell test cases.

Reference manual pages for dTET2 programs and file formats are presented in an appendix to this guide. Refer to the dTET2 Programmers Guide and the chapters entitled ''Application programming interfaces'' in the dTET2 Specification for information on the dTET2 APIs.

2.3 dTET2 features and facilities

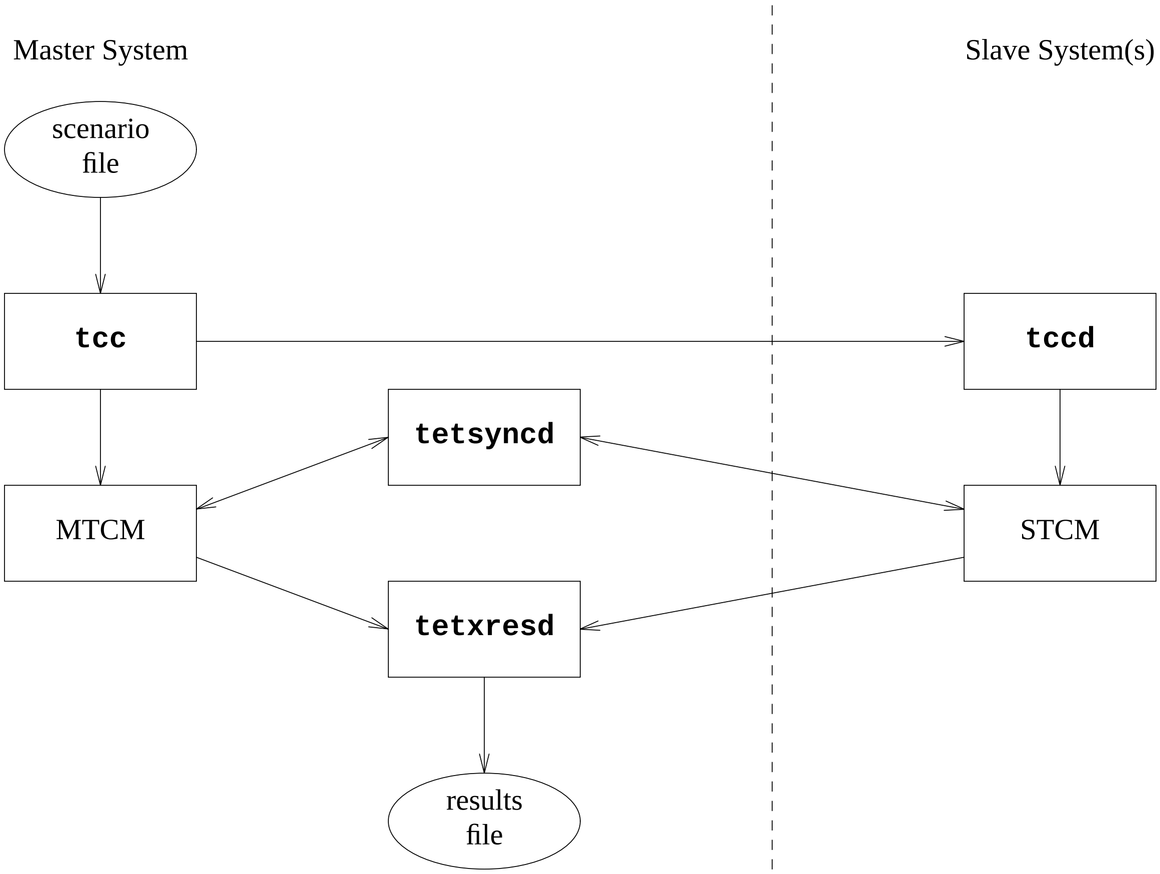
dTET2 provides facilities to execute test cases in several ways as follows:

- Execution of non-distributed test cases on the local system (i.e., local test cases).
- Execution of non-distributed test cases on a single remote system (i.e., remote test cases).
- Concurrent execution of non-distributed test cases on several remote systems.

- Execution of distributed test cases with the parts of each test case executing simultaneously on both the local (master) system and one or more remote (slave) systems.
- Combinations of the above elements executing in parallel.
- Sequences of the above elements executing a specified number of times.

2.4 dTET2 architecture

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

2.5 Test suite structure

A **test suite** is the largest grouping of tests that can be processed by the dTET2 Test Case Controller.

A test suite is made up of one or more **test cases**. If the test case is written in ‘C’, the user-supplied test code is compiled and linked with the Test Case Manager and the API library to produce an executable file. A test case is the smallest test program unit that can be built or cleaned up by the Test Case Controller.

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

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

A **distributed test purpose** is a test purpose which consists of a master part and one or more slave parts. The master part of a distributed test purpose executes on the master system and the slave parts of a distributed test purpose each execute on their respective slave systems. Each part of a distributed test purpose submits a result which indicates the success or failure of that part of the test purpose. The Execution Results daemon `tetxresd` gathers these results from the various parts of the test purpose and arbitrates between them to yield a **consolidated result** for the test purpose which appears in the test case journal file. A description of the way in which this arbitration is performed is presented in the section entitled ‘‘Making journal entries’’ in the dTET2 Programmers Guide.

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

3. Installing dTET2

3.1 Introduction

This chapter describes how to build and install dTET2. You must build and install dTET2 on all the systems that you want to use to run local, remote or distributed tests.

3.2 The TET root directory

You must choose a directory on each system in which to load the dTET2 distribution.

Note: Some of the file names in dTET2 are the same as those in previous TET and DTET distributions. You should **not** load dTET2 in a directory that is occupied by an existing TET or DTET distribution, otherwise some TET and DTET files will be overwritten.

Each directory where dTET2 is loaded is known as the **TET root directory** and will be described here as *tet-root*. When you specify the path name of your TET root directory, replace *tet-root* with your equivalent path name.

3.3 Loading the distribution

You should load the dTET2 distribution on to every system on which you want to run local, remote or distributed tests.

After you load the dTET2, you should have the following subdirectories on each system:

tet-root`/bin`	Directory that will contain executable tools for dTET2.
tet-root`/demo`	Test suite root directory for the dTET2 example distributed test case.
tet-root`/doc/dtet2`	Directory containing dTET2 documentation source files.
tet-root`/inc/dtet2`	Directory containing header files for use by dTET2 test suites.
tet-root`/lib/dtet2`	Directory that will contain library files for use when building dTET2 tools and ‘C’ language test cases.
tet-root`/lib/xpg3sh`	Directory that will contain library files for use when building Shell test cases.
tet-root`/src/dtet2`	Source directory for the dTET2 tools and API.
tet-root`/src/dtet2/apilib`	Directory containing source files for TET-derived API functions.
tet-root`/src/dtet2/dapilib`	Directory containing source files for dTET2-specific API functions.
tet-root`/src/dtet2/defines`	Directory containing example system-specific makefile definition files.

tet-root`/src/dtet2/demo/master`	Directory containing source files for the master part of a simple demonstration distributed test case.
tet-root`/src/dtet2/demo/slave`	Directory containing source files for the slave part of a simple demonstration distributed test case.
tet-root`/src/dtet2/dtet2lib`	Directory containing source files for low-level dTET2 library functions.
tet-root`/src/dtet2/inc`	Directory containing header files for the dTET2 ‘C’ sources.
tet-root`/src/dtet2/inetlib`	Directory containing source files for INET transport-specific library functions.
tet-root`/src/dtet2/servlib`	Directory containing source files for the generic server and server access library functions.
tet-root`/src/dtet2/syncd`	Directory containing source files for the dTET2 Synchronisation daemon `tetsyncd`.
tet-root`/src/dtet2/tcc`	Directory containing source files for the dTET2 Master Test Case Controller `tcc`.
tet-root`/src/dtet2/tccd`	Directory containing source files for the dTET2 Test Case Controller daemon `tccd`.
tet-root`/src/dtet2/tcm`	Directory containing source files for the dTET2 Test Case Manager components.
tet-root`/src/dtet2/xresd`	Directory containing source files for the dTET2 Execution Results daemon `tetxresd`.
tet-root`/src/dtet2/xtilib`	Directory containing source files for XTI transport-specific library functions.
tet-root`/src/xpg3sh/api`	Source directory for the Shell API.

3.4 Compiling the source

This section describes how to build dTET2. You should build and install dTET2 on every system on which you want to run local, remote or distributed tests.

3.4.1 System requirements

Prior to building dTET2 on each system, you should check that the following requirements are met as dTET2 will not build without them. dTET2 requires that each system’s operating system be compliant with all of the following:

- POSIX.1[1] with either standard ‘C’ or common ‘C’ support.

1. IEEE Std 1003.1-1990, *Portable Operating System Interface for Computer Environments*.

- XPG3, Vol. 1[2] (for the Shell and other commands).

In addition, each system should provide one of the following types of inter-process communication facilities:

- The set of network-related system calls and/or library routines, sometimes known as *Berkeley sockets*,[3] that allow access to TCP/IP network facilities.
- X/Open Transport Interface (XTI) library routines.[4]

dTET2 provides a series of makefiles that can be used to build the toolkit in a UNIX[5] type of environment. If you are not running a UNIX type of operating system, you will have to develop your own equivalent method of building the dTET2.

3.4.2 Before you build dTET2

The following subsections describe decisions that you must make before you build dTET2 on each system.

3.4.2.1 Which network interface do you want to use?

As indicated previously, dTET2 uses network facilities to communicate between different systems and processes. At present, dTET2 can be compiled to use one of the following network interfaces:

- BSD-style network calls (i.e., `socket()`, `connect()`, etc.).
- X/Open Transport Interface (XTI).

You must decide which of these you want to use before you build dTET2. The choice that you make will determine which library of dTET2 transport-specific routines are used when dTET2 is built.

In addition, the choice that you make will affect the ease with which dTET2 can be installed and used on any particular system. There are advantages and disadvantages associated with the use of each network access method, as follows:

- Advantages of using BSD-style networking are:
  — BSD networking performs well and is comparatively easy to set up
  — BSD networking is used as the basis of network services on almost all networked UNIX systems
- whereas disadvantages are:
  — since the API is not covered by standards, some systems vary in their network implementation; thus you may find it necessary to change some of the transport-

2. *X/Open Portability Guide Issue 3, Volume 1: XSI Commands and Utilities*.
3. As implemented in BSD4.3 and SunOS.
4. Defined in *X/Open Portability Guide Issue 3, Volume 7: Networking Services*.
The transport provider must support automatic address generation as defined in the description of the `t_bind()` subroutine.
5. UNIX is a registered trademark of UNIX System Laboratories, Inc. in the U.S. and other countries.

specific part of the dTET2 code before you can compile it on your system.

- Advantages of using XTI networking are:
  - since the XTI API is defined by a standard, most of the dTET2 code should compile without change on a conforming system
- whereas disadvantages are:
  - the system is difficult to set up, particularly when machine addresses must be specified
  - the performance of dTET2 is poor when XTI is used, mainly due to the lack of a transport-independent event notification mechanism in XTI[6]
  - since the XTI standard does not define a means to specify transport addresses in a machine-independent way, programs which must deal with addresses need to make use of network-related header files and functions not defined by the XTI standard; thus you may find it necessary to change some of the transport-specific part of the dTET2 code before you can compile it on your system even when XTI is used.

On balance, it is probably best not to consider using XTI for practical testing unless BSD networking is not available on your systems, or the implementation of the BSD network API is such that substantial changes to the transport-specific dTET2 code would be required.

If you **do** decide to use XTI, you must specify which underlying transport provider(s) you want to use. Support is provided for you to use XTI with TCP/IP or the OSI connection-orientated transport.

3.4.2.2 How do you want to start the TCC daemon?

dTET2 uses a client/server architecture and one of the servers — the TCC daemon (`tccd`) — is required to run on each system which may run remote or distributed tests. This daemon may be started in one of several ways as follows:

- At system boot time from an entry in the system initialisation file `/etc/rc`.
- At system boot time from an entry in the `init` configuration file `/etc/inittab`.
- On demand, from an entry in the `inetd` configuration file `/etc/inetd.conf`.[7]

The way in which `tccd` expects to be started depends on certain compiler options that you specify when dTET2 is built. You must decide which version of `tccd` you want to build before proceeding with the dTET2 build stage. You can choose to build a different version of `tccd` on different systems if you want to.

The versions of `tccd` that you may choose to build are known as the **rc** version, the **inittab** version and the **inetd** version of `tccd`, respectively, corresponding to the different ways in which `tccd` may be started, as listed above. If you have not installed dTET2 before and are unsure which version to choose, you may find it best to build the **rc** version of `tccd` initially, since this

6. If your system implements a SVID3-conforming `poll()` system call, you can compile dTET2 to use `poll()` to perform event notification — this helps to improve system performance when XTI is used.

7. This option is only available if you decide to use the socket network interface.

one is probably the easiest version to get started.

3.4.3 Preparing to build dTET2

Once you have made the decisions described above, you can prepare to build dTET2 with the makefiles supplied. The following subsections describe what you must do on each system in order to configure dTET2 in accordance with the decisions that you have made.

3.4.3.1 The dTET2 configuration script

You must run the dTET2 configuration script in the file *tet-root*/`dtetcfg`. This script asks you which network interface you want to use when building dTET2. You should reply with `inet` to use the socket network interface, or `xti` to use the XTI network interface.

Each source directory contains makefile **include** files which contain transport-specific makefile fragments for each supported network interface. The dTET2 configuration script makes a link with a standard name to the appropriate transport-specific **include** file in each directory, so that the required transport-specific definitions are included in each directory's makefile.

3.4.3.2 The `defines.mk` file

There is a makefile definition file that you must supply on each system before dTET2 can be built; the name of this file is *tet-root*/`src/dtet2/defines.mk`. This file contains system-specific definitions and is **include**d in the makefiles that are used to build dTET2.

Several example makefile definition files suitable for use with various systems are included with the distribution and may be found in the directory *tet-root*/`src/dtet2/defines`. If you are building dTET2 on one of the systems for which an example makefile definition file is supplied and are satisfied that the definitions provided are correct for your system, it should be sufficient simply to make a link from the example `.mk` file to *tet-root*/`src/dtet2/defines.mk`. However, if you are building the dTET2 on some other system or wish to change any of the example definitions, you should instead copy one of the supplied `.mk` files in the `defines` directory to *tet-root*/`src/dtet2/defines.mk` and edit it as required.

In addition, a template that can be used to construct your own makefile definitions is provided in the file *tet-root*/`src/dtet2/defines/template.mk` for use in cases where it is necessary to create the `defines.mk` file from scratch.

3.4.3.3 `make` definitions related to the TET-derived source

The `make` variable `TET_CDEFS` in the `defines.mk` file should be set to the compiler definitions that are to be used when compiling the portion of the dTET2 ‘C’ source that is derived from TET release 1.10. This code is written to POSIX.1 using common ‘C’ language, although prototypes are provided in the appropriate header files for use with ANSI ‘C’ compilers. However, one extension to POSIX.1 is required by the TET-derived source; namely, the symbol `NSIG` which defines the number of signals supported by the UNIX implementation. Many UNIX systems already define `NSIG` in `<signal.h>` as an extension to POSIX.1. In this case, `NSIG` can be made available by compiling with the appropriate feature test macro defined in addition to `_POSIX_SOURCE`.

The set of compiler definitions (but not other compiler options) that is to be be used when compiling the TET-derived source should be assigned to the variable `TET_CDEFS` in `defines.mk`. `TET_CDEFS` should always include a definition for `_POSIX_SOURCE` (either explicitly or as a result of defining some other feature test macro).

For example, on a system which conforms fully to POSIX.1 and implements 32 signals, `TET_CDEFS` might be set as follows:

```
TET_CDEFS = -D_POSIX_SOURCE -DNSIG=32
```

`tcc` catches all the signals it can, from 1 up to and including `(NSIG – 1)`. On some systems this can cause problems with non-standard signals; for example, `SIGCLD` if it is supported separately from `SIGCHLD`. Also, with the default signal handling, `tcc` cannot be suspended and restarted using job control signals. Two additional compiler definitions may be added to `TET_CDEFS` to alter the signal handling. The `TET_SIG_IGNORE` definition specifies a (comma-separated) list of signal numbers to ignore, and the `TET_SIG_LEAVE` definition specifies a list of signal numbers to leave alone. If you wish to use job control with `tcc`, you should include the job control signals in `TET_SIG_LEAVE`.

Thus, on the same system as used in the previous example, to arrange for `tcc` to ignore signal 28 and to leave signals 25 and 26 alone, `TET_CDEFS` might be set as follows:

```
TET_CDEFS = -D_POSIX_SOURCE -DNSIG=32 \
        -DTET_SIG_IGNORE=28 -DTET_SIG_LEAVE=25,26
```

3.4.3.4 `make` definitions related to the dTET2-specific source

3.4.3.4.1 Introduction

The `make` variable `DTET_CDEFS` in the `defines.mk` file should be set to the compiler definitions that are to be used when compiling the portion of the dTET2 ‘C’ source that is specific to dTET2. This code implements certain dTET2-specific functions (including transport-specific functions) that may use features that are not specified in POSIX.1, although they are available on many UNIX systems on which TCP/IP networking is implemented. As with the TET-derived source, the dTET2-specific source is written using the common ‘C’ language and prototypes are provided in the appropriate header files for use with ANSI ‘C’ compilers.

3.4.3.4.2 General compiler definitions for the dTET2-specific source

The set of compiler definitions (but not other options) that should be used when compiling the dTET2-specific source should be assigned to the variable `DTET_CDEFS` in `defines.mk`.

3.4.3.4.3 Specifying which version of `tccd` to build

You may need to add a compiler definition to `DTET_CDEFS` to cause the required version of `tccd` to be built, depending on which choice you made earlier as described in the section entitled ‘‘Before you build dTET2’’ above:

- If you decided to build the **rc** version of `tccd`, you do not need to add an extra compiler definition to `DTET_CDEFS`.
- If you decided to build the **inittab** version of `tccd`, you should append `–DINITTAB` to `DTET_CDEFS`.
- If you decided to build the **inetd** version of `tccd`, you should append `–DINETD` to `DTET_CDEFS`.[8]

8. This option is only available if you have chosen to use the socket network interface.

3.4.3.4.4 XTI-related compiler definitions

If you have chosen to use the XTI network interface, you must add extra compiler definitions to `DTET_CDEFS` in order to specify which transport provider(s) may be used. At least one of these definitions must appear when the XTI network interface is used, as follows:

- If you want to use XTI over TCP/IP, you should append `–DTCPTPI` to `DTET_CDEFS`.
- If you want to use XTI over the OSI connection-oriented transport, you should append `–DOSITPI` to `DTET_CDEFS`.

If your system implements a SVID3-conforming `poll()` system call, you can compile dTET2 to use `poll()` to perform event notification instead of having to use a polling loop that includes a `sleep()` function call. It is recommended that you do this if possible since use of `poll()` for this purpose helps to overcome the performance problems that would otherwise occur when only XTI-conforming functions are used. If you want to use `poll()` in this way, you should append `–DSVID3_POLL` to `DTET_CDEFS`.

3.4.3.4.5 Name of the Test Case Controller daemon

The name by which the Test Case Controller daemon is to be known should be assigned to the `TCCD` variable in `defines.mk`. `TCCD` should be set to `in.tccd` on systems where the TCC daemon is to be started by `inetd` (in accordance with established conventions); otherwise, `TCCD` should be set to `tccd`.

3.4.3.4.6 Examples

For example, on a system that makes network-related symbols visible in header files when the feature test macro `_ALL_SOURCE` is defined and you choose to build the **rc** version of `tccd` and to use the socket network interface, you might make the following definitions in the `defines.mk` file:

```
TCCD = tccd
DTET_CDEFS = -D_ALL_SOURCE
```

or, to build the **inetd** version of `tccd` on the same system, you might make the following definitions in the `defines.mk` file:

```
TCCD = in.tccd
DTET_CDEFS = -D_ALL_SOURCE -DINETD
```

3.4.3.5 Other `make` definitions

The following subsections describe other `make` variables that you may need to set in your `defines.mk` file.

3.4.3.5.1 C compiler

The name of the ‘C’ compiler is specified by the `CC` variable. For example, the following is appropriate on many systems:

```
CC = cc
```

3.4.3.5.2 Common compiler definitions and header file directories

Compiler definitions and header file directories that are to be specified when compiling both the TET-derived and dTET2-specific source should be assigned to the `CDEFS` variable in `defines.mk`. These will always include at least the following:

```
CDEFS = -I$(INC) -I$(DINC)
```

and may include other `-I` and `-D` options as well.

3.4.3.5.3 Other compiler options

Other compiler options should be assigned to the `COPTS` variable. For example, the following assignment will invoke the optimiser on many systems:

```
COPTS = -O
```

3.4.3.5.4 Loader options

Loader options should be assigned to the `LDFLAGS` variable. For example, you might make the following assignment to pass the `-z` option to the loader:

```
LDFLAGS = -z
```

3.4.3.5.5 System libraries

System library names and `-l` options to be added to the end of the compiler link-edit command line should be assigned to the `SYSLIBS` variable. For example, if network access routines are not in the standard 'C' library, you might make the following assignment:[9]

```
SYSLIBS = -lnsl
```

3.4.3.5.6 Archive library maintainer

The name of the archive library maintainer is specified by the `AR` variable. For example, the following assignment is appropriate on most systems:

```
AR = ar
```

3.4.3.5.7 `lorder` and `tsort`

If your system has working `lorder` and `tsort` utilities, set the `LORDER` and `TSORT` variables thus:

```
LORDER = lorder
TSORT = tsort
```

otherwise, make the following assignments:

```
LORDER = echo
TSORT = cat
```

9. On many systems, it is necessary to append `-lxti` to this variable when the XTI network interface is used.

3.4.3.5.8 mcs

On some systems, the format of an `a.out` file includes a `.comment` section that can be compressed by `mcs -c`. If your system has the `mcs` utility, set the `MCS` variable thus:

```
MCS = mcs
```

otherwise, make the following assignment:

```
MCS = :@
```

3.4.3.5.9 ranlib

If your system requires that `ranlib` must be used to process an archive library after it has been updated by the archive library maintainer, set the `RANLIB` variable thus:

```
RANLIB = ranlib
```

otherwise, make the following assignment:

```
RANLIB = :@
```

3.4.3.6 make definitions related to the Shell API

Certain `make` variables must be set in the `defines.mk` file for use with the Shell API. These variables are used to customise Shell API source files when those files are installed.

`SH_STD_SIGNALS` defines a space-separated list of standard signal numbers which must be specified to the Shell Test Case Manager. You should assign signal numbers to this variable that the Shell uses to identify the following signals on your system: `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGILL`, `SIGABRT`, `SIGFPE`, `SIGPIPE`, `SIGALRM`, `SIGTERM`, `SIGUSR1`, `SIGUSR2`, `SIGTSTP`, `SIGCONT`, `SIGTTIN` and `SIGTTOU`.

`SH_SPEC_SIGNALS` defines a space-separated list of signal numbers which must be left alone by the Shell Test Case Manager. You should assign signal numbers to this variable that the Shell uses to identify the following signals on your system: `SIGKILL`, `SIGCHLD` and `SIGSTOP`. In addition, many shells cannot trap other signals (e.g., `SIGSEGV`), so you may need to add these signals to `SH_SPEC_SIGNALS` as well.

For example:

```
# SIGHUP, SIGINT, SIGQUIT, SIGILL, SIGABRT, SIGFPE, SIGPIPE, SIGALRM,
# SIGTERM, SIGUSR1, SIGUSR2, SIGTSTP, SIGCONT, SIGTTIN, SIGTTOU
SH_STD_SIGNALS = 1 2 3 4 6 8 13 14 15 16 17 25 26 27 28

# SIGKILL, SIGCHLD, SIGSTOP, SIGSEGV
SH_SPEC_SIGNALS = 9 18 24 11
```

3.4.4 Building dTET2

When all the `make` definitions are correct, you may build dTET2. You should perform the following operations on each system:

1. To start, if you are not already there, change directory to *tet-root*`/src/dtet2`, thus:

   cd *tet-root*`/src/dtet2`

2. Type

   ```
   make install
   ```

   to build the tools and API and install them in their destination directories under *tet-root*.

3.5 Configuring your system to run dTET2

Once you have built dTET2 on each system, you must configure each system to run dTET2. You will need to have administrative privilege in order to perform many of the operations described in the following subsections.

3.5.1 Required system database entries

The following sections describe entries that you must make in system databases on each system.

3.5.1.1 Password database entry

When `tccd` starts up, it attempts to change its user and group IDs to those specified for the user `tet` in the system password database. In addition, `tccd` changes directory to the home directory specified for the user `tet`. (A different user name may be specified using the `−u` command-line option if required; see the `tccd` manual page at the end of this guide for details.)

You should create a home directory for the user `tet` and add a suitable entry to the password database on each system. The user and group IDs allocated to the user `tet` need not imply any special privilege (i.e., they should have value 100 or greater).

3.5.1.2 Services database entry

If you built dTET2 to use the socket network interface, you must add an entry to the **services** database (often the file `/etc/services`) on each system where dTET2 is installed.

When the Test Case Controller daemon `tccd` is built to use the socket network interface, it listens for requests on the well-known Internet TCP port number specified for the **tcc** service in the **services** database. This port number must be the same on all systems that are to participate in a set of remote or distributed tests. The **tcc** port number should be that of a non-privileged port (i.e., 1024 or greater).

For example, to define the well-known port as TCP port 1234, add the following line to the `/etc/services` file on each dTET2 system:

```
tcc     1234/tcp
```

3.5.2 The systems equivalence file

If you built dTET2 to use the socket network interface, you must create a file called `systems.equiv` in the home directory of the user `tet` on each system where dTET2 is installed (or the home directory of the user that you will specify if you decide to run `tccd` with the `−u` command-line option). You should create an entry in this file for each system that is to be permitted to send requests to `tccd` on this system. `tccd` will only process requests from another system if an entry for that system appears in the `systems.equiv` file.

An example `systems.equiv` file is included in the dTET2 distribution. You may copy this file to the `tccd` user's home directory and edit it as required.

3.5.3 Starting `tccd`

You should make arrangements for `tccd` to be started on each system where dTET2 is installed. Although it is possible for some of the versions of `tccd` to be started interactively by users, it is recommended that you arrange for `tccd` to be started automatically on each dTET2 system. This is so as to ensure that all instances of `tccd` execute in a known environment and with a known set of privileges. If this is not done, you will be unable to guarantee that remote or distributed test cases execute in a repeatable way.

The way that you arrange for `tccd` to be started depends on which version you chose to build as described in the section entitled ‘‘Before you build dTET2’’ above. The following sections describe how to arrange to start the different versions of `tccd`. For each system on which you want `tccd` to be started, you should perform the instructions in only **one** of the sections depending on which version of `tccd` you built on each system.

Additional command-line options are required by `tccd` when it is built to use the XTI network interface. For details of these options, refer to the section entitled ‘‘Using `tccd` with XTI’’ later in this chapter.

You should not start `tccd` on a machine connected to an external network without first considering the security implications of doing so. For further details, refer to the section entitled ‘‘Network security considerations’’ elsewhere in this guide.

3.5.3.1 Starting the RC version of `tccd`

If you built the **rc** version of `tccd`, you should add the following lines at a suitable place in one of the `/etc/rc` files:

```
if test -x tet-root/bin/tccd
then
        tet-root/bin/tccd && echo tccd started
fi
```

Once this is done, `tccd` will be started each time the system enters multi-user mode. It will be necessary for you to reboot the system in order to start `tccd`. When you reboot the system, look for the message

```
tccd started
```

on the system console when the system comes up multi-user, and check that `tccd` has printed a time-stamped START message to the file `/tmp/tccdlog`. Then, use `ps` to ensure that an instance of `tccd` is indeed running. If `tccd` failed to start correctly, you should check the file `/tmp/tccdlog` for diagnostic messages indicating the reason why `tccd` was unable to start execution.

3.5.3.2 Starting the INITTAB version of `tccd`

If you built the **inittab** version of `tccd`, you should add the following line to the file `/etc/inittab`:[10]

tet1:3:respawn:*tet-root*/bin/tccd

Then, type

```
init q
```

to make `init` take account of the new `/etc/inittab` entry and start `tccd`. When you have done this, check that `tccd` has printed a time-stamped START message to the file `/tmp/tccdlog`. Then, use `ps` to check that an instance of `tccd` is indeed running. If `tccd` failed to start correctly, you should check the file `/tmp/tccdlog` for diagnostic messages indicating the reason why `tccd` was unable to start execution.

3.5.3.3 Starting `in.tccd` (the INETD version of `tccd`)

If you built `in.tccd`, you should add the following line to the file `/etc/inetd.conf`:[11]

tcc stream tcp nowait tet *tet-root*/bin/in.tccd in.tccd

Then, use `ps` to determine the process ID of `inetd` and type

kill -1 *inetd-pid*

where *inetd-pid* is the process ID of `inetd`. This will cause `inetd` to take account of the new entry in `/etc/inetd.conf` and start an instance of `in.tccd` each time a dTET2 process connects to the well-known **tcc** port.

Since `in.tccd` is run on demand by `inetd`, no START message is printed to the file `/tmp/tccdlog` and it is not possible to use `ps` to check whether or not `in.tccd` is running until another process connects to the well-known **tcc** port. If you have difficulties in getting `in.tccd` to start when you attempt to run a remote or distributed test case, it might be helpful to try and run one of the other versions of `tccd` first in order to isolate the cause of the problem.

3.5.4 Using `tccd` with XTI

If you built `tccd` to use the XTI network interface, you must specify additional command-line options when you start `tccd`.

The following options should appear on the `tccd` command line:

`-M` *mode* Specifies which transport provider to use. *mode* should be one of the following:

`TCP` use XTI over TCP/IP (this is the default)

10. Note that this example assumes that your system starts multi-user operation with network services enabled when `init` enters run level 3. If your system uses a different run level (or levels) for this purpose, you must replace the `3` in this example with the correct value(s).

11. Note that older versions of `inetd` require that this line should appear **before** lines describing `inetd` internal services.

`OSICO` use XTI over OSI connection-oriented transport

`-P` *tpi* Specifies the name of the transport provider identifier to use in XTI function calls. This is usually the name of a streams special file; for example `/dev/tcp`.

`-p` *addr* Specifies the XTI address string to use when listening for incoming connections. *addr* is a sequence of 2-digit hexadecimal values whose contents depend on the transport provider, network implementation and machine architecture.

For example:

```
tccd -P /dev/tcp -M TCP -p 100223800a010200000000000000000000
```

Note that only the **rc** and **inittab** versions of `tccd` are able to be built to use the XTI network interface.

4. Running the dTET2 demonstration

4.1 Introduction

This chapter describes how to run the dTET2 demonstration program. Before you can run the demonstration, you should build and install dTET2 on all the systems that you want to use to run local, remote or distributed tests, as described in the chapter entitled ‘‘Installing dTET2’’ elsewhere in this guide.

The dTET2 demonstration program is a simple test suite consisting of three distributed test cases that are built, executed and cleaned on each system participating in the test. You may run the demonstration program to verify that dTET2 has installed correctly.

In addition, the demonstration program will be helpful to users who wish to understand the basic components of a distributed test case in its simplest form. The operation of the program is discussed in the section entitled ‘‘A simple distributed test suite’’ in the dTET2 Programmers Guide.

4.2 System requirements

In order to run the demonstration program, you will need to make use of two machines connected to the same local area network. When the demonstration is run, one of these machines will act as the master system and the other will act as the slave system.

In addition to choosing these two systems, you must perform the following actions:

1. Ensure that dTET2 is installed on each system that will participate in the demonstration.
2. Verify that the `tccd` daemon is running on the slave system.
3. Ensure that you know the name of the *tet-root* directory on each system.
4. If you have built dTET2 on the slave system to use the socket network interface, ensure that an entry for the master system exists in the file `systems.equiv` in the `tet` user’s home directory on the slave system.

4.3 Installation

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

1. Log on to the system that you want to use as the master system.
2. Copy the master part of the demonstration to *tet-root*`/demo` on the master system, thus:

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

3. Log on to the system that you want to use as the slave system.
4. Copy the slave part of the demonstration to *tet-root*`/demo` on the slave system, thus:

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

4.4 Configuration

Before you can run the demonstration program, you must customise the `systems` file and the distributed configuration file `tetdist.cfg` that are supplied with the demonstration. In order to do this, you should log in to the master system and change directory to *tet-root*. Then, you should work through the instructions presented in the following subsections.

4.4.1 The `systems` file

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

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

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

If you have built dTET2 on the master system to use the XTI network interface, you must add a third field to the entry for system 001. This field should contain the XTI address string of the Test Case Controller daemon `tccd` that is running on the slave system. The format of XTI address strings is described in the section entitled ‘‘System definitions’’ in the dTET2 Programmers Guide.

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

4.4.2 The `tetdist.cfg` file

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

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

# The following variables are only referenced by XTI based TCCs

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

If your **tet root** directory on the slave system is not `/home/dtet2/dtet2`, you must change the first two variables to refer to the **tet root** and **test suite root** directories on the slave system.

The last three variables are only required if dTET2 on the master system has been built to use the XTI network interface. If you built dTET2 on the master system to use the XTI network interface, you should refer to the descriptions of these variables in the section entitled ‘‘Configuration variables used for distributed testing’’ elsewhere in this guide and then set the variables to values that are appropriate for your system.

4.4.3 Makefiles

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

You may need to customise the `$(CC)` command lines in order to resolve external function names used in the dTET2 Test Case Manager and API. For example, if you have built dTET2 to use the XTI network interface, you will probably need to append `–lxti` to the compiler command line.

4.5 Running the demonstration

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

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

1. Log in to the master system and change directory to *tet-root*.
2. Make sure that the setting of your PATH environment variable includes *tet-root*`/bin`.
3. Set your TET_ROOT environment variable to refer to *tet-root*, thus:

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

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

   ```
   tcc -bec demo
   ```

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

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

   Then the demonstration test suite will be built, executed and cleaned up.

5. Once `tcc` has finished execution, you can examine the journal file and verify the results of the demonstration.

An example of a journal file that might be generated when the demonstration program is run is presented in the appendix entitled ‘‘dTET2 demonstration journal file’’ at the end of this guide.

4.6 Problem diagnosis

If the demonstration did not run as expected, it is likely that one or more dTET2 components will have generated diagnostic messages describing what went wrong. You should check for diagnostic information in locations listed in the following table:

Location of diagnostic information	Source of diagnostic message

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

Journal file (on the master system)	`tcc` Test Case Managers on both systems APIs on both systems
`tccd` log file `/tmp/tccdlog` on the slave system	`tccd` slave Test Case Manager API on the slave system

5. Using dTET2

5.1 Introduction

This chapter describes how to use dTET2. Before you read this chapter, you should build and install dTET2 on all the systems that you want to use to run local, remote or distributed tests, as described in the chapter entitled ‘‘Installing dTET2’’ elsewhere in this guide.

5.2 dTET2 concepts

This section describes some concepts employed in dTET2.

5.2.1 dTET2 components

The components that make up dTET2 are described in the chapter entitled ‘‘Overview of the Distributed Test Environment Toolkit Version 2’’ elsewhere in this guide. You should read this chapter now if you have not already done so.

5.2.2 Modes of operation

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

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

5.2.3 Local and remote systems

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

dTET2 systems are identified by a three-digit **system ID**. System ID `000` refers to the master system and other system ID values refer to slave systems.

5.2.4 Test case types

dTET2 is able to process **local** test cases, **remote** test cases and **distributed** test cases. For a description of what is meant by these different test case types, refer to the chapter entitled ‘‘Overview of the Distributed Test Environment Toolkit Version 2’’ elsewhere in this guide.

5.2.5 dTET2 directory layout

The whole of dTET2 and any accompanying test suites reside in a single directory hierarchy on each system that is to participate in local, remote or distributed testing. The top of this hierarchy is known as the **tet root** directory. Below **tet root** there are directories containing dTET2 sources, executables and documentation.

In addition, directory hierarchies that contain the test suites that are to be used with dTET2 must reside below the **tet root** directory. The top of each test suite directory hierarchy is known as the **test suite root** directory.

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

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

An example dTET2 directory hierarchy is presented in the appendix entitled ‘‘dTET2 directory structure’’ at the end of this guide.

5.2.6 Environment variables

The following environment variables are used by `tcc` on the master system:

`TET_ROOT` Specifies the location of the **tet root** directory on the master system. This environment variable must always be set when `tcc` is run.

`TET_EXECUTE` If this variable is set, it specifies the location of the alternate execution directory on the master system.

`TET_TMP_DIR` If this variable is set, it specifies the location of `tcc`'s temporary directory on the master system.

5.2.7 Configuration variables

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

variable=*value*

Some configuration variables alter the behaviour of dTET2, whereas others are specific to the test suite being processed and may be accessed by test cases and by build and clean tools.

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

Configuration variables that alter the behaviour of dTET2 are described further in the section entitled ‘‘Configuration files’’ later in this chapter.

5.2.8 Test purpose results

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

The following result codes are defined in dTET2:

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

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

5.3 dTET2 data files

This section describes the formats and locations of data files used by dTET2.

5.3.1 The `systems` and `systems.equiv` files

The `systems` file resides in the **tet root** directory on each system and is used by dTET2 processes to map dTET2 system IDs to some information (such as a host name) that can be used by the network interface to identify a machine. The format of the `systems` file varies according to which network interface is used by dTET2.

The `systems.equiv` file resides in the `tccd` user's home directory on each system (usually that of the user `tet`) and may be used by `tccd` to determine whether or not to accept a request from a remote system. Note that not all dTET2 network interfaces make use of this file.

Example `systems` and `systems.equiv` files are included in the dTET2 distribution. Refer to the manual pages at the end of this guide for details of the formats of these files.

5.3.2 The scenario file

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

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

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

Simple descriptions of these scenario file elements are presented in the following sections, together with some scenario examples. A more complete description of the format of the scenario file and some more comprehensive examples are presented in the section entitled ‘‘Scenario files’’ in the dTET2 Programmers Guide.

5.3.2.1 Scenario directives

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

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

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

`:parallel:`*file*
: The test cases listed (one per line) in *file* are processed in parallel.

`:parallel:`
test-case
. . .
`:endparallel:`
: The named *test-cases* are processed in parallel.

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

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

`:remote,`nnn_1`,`nnn_2`,...:`*file*
: Each test case listed (one per line) in *file* is processed concurrently on systems nnn_1, nnn_2 etc.

`:remote,`nnn_1`,`nnn_2`,...:`
test-case
. . .
`:endremote:`
: Each named *test-case* is processed concurrently on systems nnn_1, nnn_2 etc.

If the list of systems specified with a `:remote:` directive includes the master system, then the test cases are processed as **distributed** test cases; otherwise, the test cases are processed as **remote** test cases.

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

5.3.2.2 Test case names

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

— in build or clean mode, test case names are interpreted relative to the test suite root directory

— in execute mode, test case names are interpreted relative to the alternate execution directory if one has been specified, otherwise they are interpreted relative to the test suite root directory

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

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

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

5.3.2.3 Scenario examples

This section presents some simple test scenario examples.

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

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

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

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

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

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

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

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

5.3.3 Configuration files

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

5.3.3.1 Configuration file format

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

variable=*value*

Lines beginning with # and blank lines are ignored.

5.3.3.2 Configuration file names

Configuration files for each test suite are located in the test suite root directory on each system. The name of the build mode configuration file is `tetbuild.cfg`, that of the execute mode configuration file is `tetexec.cfg` and that of the clean mode configuration file is `tetclean.cfg`.

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

5.3.3.3 Setting configuration variables on master and slave systems

Configuration variables may be specified both on the master system and on any slave systems that are to participate in remote or distributed testing. Configuration variable assignments made on the master system are propagated to the slave systems; however, configuration variable assignments made on a slave system normally have precedence over those made on the master system.

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

```
TET_BUILD_TOOL=make
```

then, the value of `TET_BUILD_TOOL` will be set to `make` on the master system and on all the slave systems.

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

```
TET_BUILD_TOOL=augmake
```

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

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

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

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

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

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

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

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

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

5.3.3.4 Configuration variables used by dTET2

The following configuration variables affect `tcc`'s operation. This list only includes those variables that it might be sensible for a dTET2 user to change. Other dTET2 configuration variables and testsuite-specific variables may be specified by the test suite author. Refer to the dTET2 Programmers Guide for a complete list of dTET2 configuration variables and their meanings.

`TET_EXEC_IN_PLACE`

Possible values: `True` or `False`; default: `False`.
Specifies whether or not `tcc` should execute test cases ‘‘in place’’. If false, `tcc` copies test case files to a temporary directory before executing them. The setting of this variable is only meaningful in execute mode. Test cases that are executed in parallel are never executed in place, regardless of the setting of this variable.

`TET_OUTPUT_CAPTURE`

Possible values: `True` or `False`; default: `False`.
Specifies whether or not test cases use the API. If false, `tcc` expects test cases to use the dTET2 API to print diagnostics and register results.

If true, `tcc` pretends that the test case contains a single invocable component. `tcc` prints IC and TP start and end messages to the journal file, and generates a result based on the test case's exit status (zero = PASS, non-zero = FAIL). In addition, `tcc` arranges to capture the test case's output and print it to the journal file as test purpose messages.

It is not possible to have a distributed test case whose parts consist of a mixture of API-conforming and non-API-conforming elements; therefore, the value of this variable

	must be the same on all systems that are to participate in distributed testing.
`TET_SAVE_FILES`	This variable specifies a (comma separated) list of file names. If, after `tcc` executes a test case, a file matching one of these names is found below the execution directory hierarchy, that file is transferred to the saved file directory tree on the same system. If a directory is found that matches one of the names, then its contents are transferred recursively. Shell file name matching syntax may be used in the list of file names.
`TET_TRANSFER_SAVE_FILES`	Possible values: `True` or `False`; default: `False`. If true, files processed on a slave system in accordance with the description of `TET_SAVE_FILES` above are transferred to the saved file directory on the master system instead of being saved locally. The setting of this variable is only meaningful on the master system.

5.3.3.5 Example configuration file

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

```
# example build mode configuration file

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

5.3.3.6 Distributed configuration file

The file `tetdist.cfg` in the test suite root directory on the master system holds configuration variables used for remote and distributed testing. Distributed configuration variables are used to specify parameters for slave systems that are equivalent to those parameters on the master system that `tcc` obtains from environment variables or deduces from its current working directory. In addition, the distributed configuration file may include variable assignments relating to the dTET2 network interface.

These variables are not accessible by test cases or by build and clean tools.

5.3.3.7 Configuration variables used for distributed testing

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

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

TET_REM*nnn*_TET_EXECUTE — The values of these variables specify the locations of alternate execution directories on slave systems. The use of these variables is optional but, if they appear, they perform the equivalent functions on slave systems to that performed by the value of the TET_EXECUTE environment variable on the master system (refer to the section entitled ‘‘Environment variables’’ earlier in this chapter).

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

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

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

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

5.3.3.8 Example distributed configuration file

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

```
# example distributed configuration file

TET_REM001_TET_ROOT=/user1/tet
TET_REM001_TET_TSROOT=/user1/tet/testsuite
TET_REM002_TET_ROOT=/user6/project/dtet2
TET_REM002_TET_TSROOT=/user6/project/dtet2/testsuite
```

5.3.4 The results code file

A test suite may provide a results code file containing valid result codes for use with that test suite. If no such file is provided, then dTET2 utility programs that perform result processing access an internal table of result codes.

If a test suite provides a result code file, the file is named tet_code and is located in the test suite root directory on the master system.

Each line in the file consists of (blank-separated) fields as follows:

Result code value
Result name
Action indicator

The result code value is a positive number between 0 and 127. Values between 0 and 31 are reserved for use by dTET2. The result name is a text string enclosed between double quotes and indicates the name of the result. The text string may contain embedded blanks. The action indicator informs the TCM what to do when a test purpose returns the corresponding result and should be one of `Continue` to cause the TCM to continue processing the next test purpose, or `Abort` to cause the TCM to abandon processing the test case.

Blank lines and lines beginning with # are ignored.

Here is an example of a result code file:

```
# example result code file

# first, the codes required by dTET2
0       "PASS"          Continue
1       "FAIL"          Continue
2       "UNRESOLVED"    Continue
3       "NOTINUSE"      Continue
4       "UNSUPPORTED"   Continue
5       "UNTESTED"      Continue
6       "UNINITIATED"   Continue
7       "NORESULT"      Continue

# then, the codes specific to this test suite
32      "INSPECT"       Continue
33      "STOP RUN"      Abort
```

5.3.5 The journal file

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

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

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

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

Each line in the journal file consists of three fields separated by `|` characters. The first field contains a number which indicates the type of the journal line. The second field contains information depending on the type of the journal line. The third field generally contains some kind of message text. Descriptions of journal lines are presented in the appendix entitled ‘‘dTET2 journal lines’’ at the end of this guide.

5.4 dTET2 programs

This section describes the function and use of dTET2 utility programs.

5.4.1 The Master Test Case Controller `tcc`

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

`tcc` always runs on the master system; indeed, the master system is defined as the system on which `tcc` runs. If `tcc` needs to perform an operation on a slave system, it sends a request to a server process running on the slave system (`tccd`) which performs the operation on `tcc`’s behalf.

5.4.1.1 Required environment variables

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

5.4.1.2 Normal scenario processing

5.4.1.2.1 Description

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

5.4.1.2.2 Examples

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

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

   ```
   tcc -b mytestsuite
   ```

   or

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

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

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

   ```
   tcc -bec mytestsuite
   ```

   or

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

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

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

   ```
   tcc -e mytestsuite myscenario
   ```

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

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

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

5.4.1.3 Rerun and Resume processing

5.4.1.3.1 Description

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

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

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

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

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

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

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

5.4.1.3.2 Examples

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

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

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

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

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

5.4.1.4 Saved files processing

As indicated earlier in the section entitled ‘‘Configuration variables used by dTET2’’, `tcc` can be instructed to save certain files and/or directories that may be created during test case execution. This action is controlled by the values of the configuration variables `TET_SAVE_FILES` and `TET_TRANSFER_SAVE_FILES`.

If `TET_TRANSFER_SAVE_FILES` is false, files are saved locally on the system on which they were created. However, if `TET_TRANSFER_SAVE_FILES` is true, files are transferred to the master system and saved there.

When files are to be saved locally on a particular system, `tcc` creates a saved files directory in the *test-suite-root*`/results` directory on that system, whose name consists of an ascending sequence number followed by one or more of `b`, `e` and `c`, indicating which operation mode(s) was being used by `tcc`, thus:[12]

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

However, when files are to be transferred and saved on the master system, `tcc` instead creates additional directories named `REMOTE`*nnn* below this directory on the master system, where *nnn* indicates the system ID from which the files are to be transferred, thus:

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

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

The variable `TET_SAVE_FILES` may be set to a (comma-separated) list of file names. If, after `tcc` executes a test case, a file matching one of these names is found below the execution directory hierarchy, that file is transferred to the saved file directory hierarchy (either locally or on the master system, depending on the value of the `TET_TRANSFER_SAVE_FILES` variable). Directories are created as required below the saved files directory so that, after each file has been saved, the saved file’s path name below the top of the saved files hierarchy is the same as the portion of the file’s path name below the test case’s execution directory.

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

The processing described above is undertaken on each system for which a `TET_SAVE_FILES` variable has been specified.

5.4.2 The Test Case Controller daemon `tccd`

The Test Case Controller daemon (`tccd`) performs functions on behalf of the Test Case Controller on slave systems. `tccd` must be started on each slave system that is to participate in remote or distributed testing, before `tcc` is invoked on the master system. In addition, `tccd` must be started on the master system if a slave test case part needs to start a remote executed process on the master system.

It is recommended that you should ask your system administrator to arrange for `tccd` to be started automatically as part of each machine’s startup procedure. Details of how to arrange to start `tccd` are presented in the section entitled ‘‘Starting `tccd`’’ elsewhere in this guide.

---

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

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

5.4.2.1 `tccd` versions and modes of operation

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

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

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

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

A result of all this is that a machine running `tccd` can service requests from more than one master system at once, since each client obtains its own copy of `tccd` to use.

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

5.4.2.2 `tccd` user and group ID

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

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

5.4.2.3 tccd log file

When tccd is executed, it connects its standard output and standard error streams to the file /tmp/tccdlog, and connects its standard input to /dev/null. If a different log file is specified by the -l command-line option, tccd reconnects its standard output and error streams to that file as soon as possible. The result of this is that test cases and other processes which may be executed by tccd start execution with their standard input, standard output and standard error streams connected to these files as well[13].

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

When the parent tccd daemon (but not in.tccd) starts, it prints a START message to the log file.

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

5.4.2.4 Terminating tccd

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

5.4.3 The Test Case Manager

The Test Case Manager (TCM) is not a separate program but is instead part of the dTET2 API.

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

The ‘C’ TCM sends diagnostic output to the journal file as test case manager messages. If this is not possible, the TCM writes diagnostic output to its standard error stream[14].

The Shell TCM is sourced by the script containing the user-supplied test code. The Shell TCM and API do not support distributed testing.

---

13. If a test case or a build or clean tool is executed with TET_OUTPUT_CAPTURE set to True, then tcc arranges for tccd to execute the tool with standard output and standard error streams connected to a different file. The contents are transferred to the master system when the tool finishes execution for inclusion in the journal file.

14. On remote systems, the TCM’s standard error stream is connected to the tccd log file.

5.4.4 The Synchronisation daemon `tetsyncd`

The Synchronisation daemon `tetsyncd` is a server that runs on the master system and undertakes the processing of automatic and user synchronisation requests. An automatic synchronisation request is generated by the Test Case Manager, and a user synchronisation request is made when a test purpose makes a call to the `tet_sync()` API library routine.

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

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

5.4.5 The Execution Results daemon `tetxresd`

The Execution Results daemon `tetxresd` is a server that runs on the master system and undertakes the processing of journal output and test purpose results for API-conforming test cases. In addition, `tetxresd` performs certain administrative functions on the master system, mostly related to **transfer save files** processing.

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

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

5.5 Network security considerations

Since dTET2 is intended for use in a testing environment which is not accessible from an external network, network security issues are not really addressed by dTET2.

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

`tetsyncd` and `tetxresd` both offer services to other processes without any authentication, apart from the dTET2 server logon procedure which does not perform host name checking. However, they do not use well-known ports and so are protected to some extent by ‘‘security through obscurity’’.

As a result of all this:

> **Users are strongly advised against running dTET2 on machines that can be accessed from an external network, unless they are satisfied that adequate measures are in place to prevent unauthorised access to networks that serve those machines.**

6. Test case synchronisation

6.1 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.

6.2 Synchronisation request concepts

6.2.1 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()` API library routine.

6.2.2 Request parameters

Each synchronisation request is accompanied by a **sync point number**, a **system ID list**, a **sync vote** and an optional **timeout**. 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.

6.2.3 Sync events

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

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

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

6.2.4 Sync states

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

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

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

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

6.2.5 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.[15]

6.3 Automatic synchronisation requests

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.

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

15. 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.

6.3.1 Test case manager synchronisation

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

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

Stage in test case processing	**Sync point number**[16]	**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.

6.3.2 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

16. 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.

in the request.

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

6.3.3 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.

6.3.4 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.

6.3.4.1 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
```

6.3.4.2 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.

6.4 User synchronisation requests

A user synchronisation request is generated when a test purpose in a distributed test case calls the `tet_sync()` API library routine. The sync point number, system ID list and timeout are specified in the `tet_sync()` call. The sync vote is always `yes` for user synchronisation requests.

`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.

6.4.1 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.

6.4.2 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.

6.4.2.1 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 synced unsuccessfully or timed out
system 0: system =  1, state = NOT-SYNCED
```

6.4.2.2 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 synced unsuccessfully \
        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.

6.4.2.3 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 synced unsuccessfully \
        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 synced unsuccessfully \
        or timed out
system 2: system =  0, state = SYNC-YES
system 2: system =  1, state = DEAD
```

APPENDICES

A. dTET2 directory structure

The following diagram illustrates the directory structure used by dTET2. Each directory level is represented by a level of indentation. Directory names are followed by a `/` character.

- *tet-root*`/`
    - `bin/`
        - `tcc`
        - `tccd`
        - `tetsyncd`
        - `tetxresd`
        - other dTET2 executables
    - `doc/`
        - dTET2 documentation
    - `inc/`
        - `dtet2/`
            - dTET2 API header files
    - `lib/`
        - `dtet2/`
            - `libapi.a`
            - `tcm.o`
            - `tcmchild.o`
            - `tcmrem.o`
        - `xpg3sh/`
            - `tetapi.sh`
            - `tcm.sh`
    - `src/`
        - `dtet2/`
            - dTET2 C source tree
        - `xpg3sh/`
            - Shell API source tree
    - `systems`
    - `systems.equiv`
    - *test-suite-root*`/`
        - `tet_code`
        - `tet_scen`
        - `tetbuild.cfg`
        - `tetclean.cfg`
        - `tetdist.cfg`
        - `tetexec.cfg`
        - `results/`
            - *nnnn*`[bec]/`
                - `REMOTE`*nnn*`/`
                    - transfer save files

`journal`
other results files
`tet_tmp_dir/`
nnnn[`bec`]`/`
test case files and directories

B. dTET2 journal lines

This appendix describes the various journal lines that may appear in a dTET2 journal file.

Test Case Controller Start

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

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

Test Case Start

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

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

Test Case Manager Start

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

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

Master System Configuration Start

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

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

Slave System Configuration Start

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

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

Configuration Variable Setting

`30||`*variable=value*

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

Configuration End

`40||`*text*

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

Test Case Controller Message

`50||`*text*

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

Scenario Information

`70||`*text*

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

Test Case End

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

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

Operator Abort

`90|`*time*`|`*text*

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

Captured Output

`100|`*activity*`|`*text*

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

Build Start

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

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

Build End

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

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

Test Purpose Start

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

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

Test Purpose Result

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

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

Clean Start

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

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

Clean End

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

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

Invocable Component Start

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

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

Invocable Component End

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

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

Test Case Manager Message

`510|`*activity*|*text*

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

Test Case Information

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

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

Parallel Start

`600||`*text*

The scenario file `parallel` directive start marker.

Parallel End

`620||`*text*

The scenario file `parallel` directive end marker.

Repeat Start

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

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

Repeat End

`720||`*text*

The scenario file `repeat` directive end marker.

Remote Start

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

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

Remote End

`820||`*text*

The scenario file `remote` directive end marker.

Test Case Controller End

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

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

C. dTET2 demonstration journal file

C.1 Introduction

The chapter entitled ‘‘Running the dTET2 demonstration’’ contains instructions on how to run the dTET2 demonstration program. This appendix contains an example journal file that might be generated when the program is run.

C.2 Example journal file

```
0|2.0 13:50:19 19930920|User: tet (664) TCC Start, \
        Command Line: tcc -bec demo
20|/home/dtet2/dtet2/demo/tetbuild.cfg 0|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=True
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
30||TET_BUILD_TOOL=make
30||TET_BUILD_FILE=-f makefile
40||Config End
20|/home/dtet2/dtet2/demo/tetexec.cfg 1|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=False
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
40||Config End
20|/home/dtet2/dtet2/demo/tetclean.cfg 2|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=True
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
30||TET_CLEAN_TOOL=rm
30||TET_CLEAN_FILE=-f
40||Config End
20|/home/dtet2/dtet2/demo/tetdist.cfg 3|Config Start
30||TET_REM000_TET_ROOT=/home/dtet2/dtet2
30||TET_REM000_TET_TSROOT=/home/dtet2/dtet2/demo
30||TET_REM000_TET_EXECUTE=
30||TET_REM000_TET_TMP_DIR=/home/dtet2/dtet2/tet_tmp_dir/12199a
30||TET_REM001_TET_ROOT=/home/dtet2/dtet2
30||TET_REM001_TET_TSROOT=/home/dtet2/dtet2/demo
30||TET_REM002_TET_ROOT=/home/dtet2/dtet2
30||TET_REM002_TET_TSROOT=/home/dtet2/dtet2/demo
30||TET_XTI_TPI=/dev/tcp
30||TET_XTI_MODE=tcp
30||TET_LOCALHOST=89.200.0.7
40||Config End
20|remote_001 0|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=True
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
30||TET_BUILD_TOOL=make
```

```
30||TET_BUILD_FILE=-f makefile
40||Config End
20|remote_001 1|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=False
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
40||Config End
20|remote_001 2|Config Start
30||TET_VERSION=2.0
30||TET_OUTPUT_CAPTURE=True
30||TET_RESCODES_FILE=tet_code
30||TET_EXEC_IN_PLACE=False
30||TET_CLEAN_TOOL=rm
30||TET_CLEAN_FILE=-f
40||Config End
70||"starting scenario"
800|000,001|Remote Start, scenario ref 4-1
110|0 /ts/tc1 13:50:20|Build Start, scenario ref 5-1
100|0|  cc -I../../inc/dtet2 -o tc1 tc1.c ../../lib/dtet2/tcm.o \
100|0|          ../../lib/dtet2/libapi.a
130|0 0 13:50:26|Build End
110|0 /ts/tc1 13:50:20|Build Start, scenario ref 5-1
100|0|  cc -I../../inc/dtet2 -o tc1 tc1.c ../../lib/dtet2/tcm.o \
100|0|          ../../lib/dtet2/libapi.a        -lxti
130|0 0 13:50:26|Build End
10|1 /ts/tc1 13:50:26|TC Start, scenario ref 5-1
15|1 2.0 1|TCM Start
400|1 1 1 13:50:28|IC Start
200|1 1 13:50:28|TP Start
520|1 1 00012220 1 1|This is the first test case (tc1)
520|1 1 00116558 1 1|This is the first test case (tc1)
220|1 1 0 13:50:28|PASS
410|1 1 1 13:50:28|IC End
80|1 0 13:50:30|TC End
10|1 /ts/tc1 13:50:26|TC Start, scenario ref 5-1
80|1 0 13:50:29|TC End
300|2 /ts/tc1 13:50:30|Clean Start, scenario ref 5-1
320|2 0 13:50:30|Clean End
300|2 /ts/tc1 13:50:30|Clean Start, scenario ref 5-1
320|2 0 13:50:31|Clean End
110|3 /ts/tc2 13:50:31|Build Start, scenario ref 6-1
100|3|  cc -I../../inc/dtet2 -o tc2 tc2.c ../../lib/dtet2/tcm.o \
100|3|          ../../lib/dtet2/libapi.a
130|3 0 13:50:36|Build End
110|3 /ts/tc2 13:50:31|Build Start, scenario ref 6-1
100|3|  cc -I../../inc/dtet2 -o tc2 tc2.c ../../lib/dtet2/tcm.o \
100|3|          ../../lib/dtet2/libapi.a        -lxti
130|3 0 13:50:36|Build End
10|4 /ts/tc2 13:50:36|TC Start, scenario ref 6-1
15|4 2.0 1|TCM Start
400|4 1 1 13:50:38|IC Start
200|4 1 13:50:38|TP Start
520|4 1 00012241 1 1|This is the second test case (tc2, master)
520|4 1 00116572 1 1|This is the second test case (tc2, slave)
```

```
220|4 1 1 13:50:39|FAIL
410|4 1 1 13:50:39|IC End
80|4 0 13:50:40|TC End
10|4 /ts/tc2 13:50:36|TC Start, scenario ref 6-1
80|4 0 13:50:39|TC End
300|5 /ts/tc2 13:50:40|Clean Start, scenario ref 6-1
320|5 0 13:50:41|Clean End
300|5 /ts/tc2 13:50:40|Clean Start, scenario ref 6-1
320|5 0 13:50:41|Clean End
70||"next is the last test case"
110|6 /ts/tc3 13:50:41|Build Start, scenario ref 9-1
100|6|  cc -I../../inc/dtet2 -o tc3 tc3.c ../../lib/dtet2/tcm.o \
100|6|          ../../lib/dtet2/libapi.a
130|6 0 13:50:46|Build End
110|6 /ts/tc3 13:50:41|Build Start, scenario ref 9-1
100|6|  cc -I../../inc/dtet2 -o tc3 tc3.c ../../lib/dtet2/tcm.o \
100|6|          ../../lib/dtet2/libapi.a -lxti
130|6 0 13:50:46|Build End
10|7 /ts/tc3 13:50:46|TC Start, scenario ref 9-1
15|7 2.0 1|TCM Start
400|7 1 1 13:50:49|IC Start
200|7 1 13:50:49|TP Start
520|7 1 00116586 1 1|This is the third test case (tc3, slave)
520|7 1 00116586 1 2|sync with master (sysid: 000)
520|7 1 00012262 1 1|This is the third test case (tc3, master)
520|7 1 00012262 1 2|sync with slave (sysid: 001)
220|7 1 0 13:50:49|PASS
410|7 1 1 13:50:49|IC End
80|7 0 13:50:50|TC End
10|7 /ts/tc3 13:50:47|TC Start, scenario ref 9-1
80|7 0 13:50:50|TC End
300|8 /ts/tc3 13:50:50|Clean Start, scenario ref 9-1
320|8 0 13:50:52|Clean End
300|8 /ts/tc3 13:50:50|Clean Start, scenario ref 9-1
320|8 0 13:50:52|Clean End
820||Remote End
70||"done"
900|13:50:52|TCC End
```

D. Server reply codes

D.1 Introduction

When a dTET2 client process makes a request of a server, a server reply code is returned to the client which indicates the success or failure of the request. Many of these reply codes are interpreted by the client and so are never visible to the user. However, some reply codes may be printed in dTET2 diagnostic messages which appear in the journal file on the master system or in the `tccd` log file on one of the slave systems.

D.2 List of server reply codes

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

`ER_OK` No error.
The request completed successfully.

`ER_ABORT` Abort test case.
The action associated with the consolidated result of a distributed test purpose indicated that the test case should be aborted.

`ER_CONTEXT` Request out of context.
A request arrived before a pre-requisite message had been received by the server.

`ER_DONE` Event already happened.
A synchronisation request arrived after the related sync event had completed, or IC or TP start or end requests arrived in the wrong order, or a CONFIG request violated the configuration variable exchange protocol.

`ER_DUPS` Duplicate system IDs in list.
The list of system IDs specified in the request contained duplicate entries.

`ER_ERR` General error code.
Diagnostics which include this reply code are always preceded by another message describing the error in more detail. If the error actually occurred on a slave system, it is likely that the more detailed message will appear in the `tccd` log file on the slave system rather than in the journal file on the master system.

`ER_FID` Bad file ID.
An invalid file ID was specified in a file transfer request. This usually indicates an internal program error of some kind.

`ER_FORK` Can’t fork.
`tccd` was unable to fork while processing an `OP_EXEC` request. A message describing the reason for the failure is printed in the `tccd` log file on the affected system.

`ER_INPROGRESS` Event in progress.
An attempt was made to modify the system ID list associated with an event after the event had started, or IC or TP start or end requests arrived in the wrong order, or a CONFIG request violated the configuration variable

exchange protocol.

`ER_INTERN` Internal server error.
A server request processing function neglected to supply a message reply code. This usually indicates an internal program error of some kind.

`ER_INVAL` Invalid request parameter.
The value of one of the request parameters was not valid for this request type. This error may be raised both in the client's server function interface code and in the server's request processing code.

`ER_LOGON` Logon protocol error.
A client issued a server request before having first logged on to the server, or attempted to log on to a server more than once. This error may be raised both in the client's server function interface code and in the server's request processing code. This usually indicates an internal program error of some kind.

`ER_MAGIC` Bad magic number.
An interprocess message header contained an invalid magic number. This can occur if the client and server data streams become out of sync or when the client and server are incompatible (e.g., because they are derived from different dTET2 releases).

`ER_NOENT` No such file or directory.
A file specified in an `OP_EXEC` request could not be found, or a saved files directory could not be accessed.

`ER_PERM` No permission.
A request was received from a client that was not authorised to send it, or the server did not have the permissions necessary to perform the requested action.

`ER_PID` Invalid process ID.
The process ID specified in an `OP_WAIT` or `OP_KILL` request did not refer to a process started by the requesting system.

`ER_RCVERR` Message receive error.
A server could not process a request because an I/O error occurred while the message was being received.

`ER_REQ` Invalid request.
A request was attempted that was not valid for this server type. This error may be raised both in the client's server function interface code and in the server's request processing code. This usually indicates an internal program error of some kind.

`ER_SIGNUM` Invalid signal number.
The signal number specified in an `OP_KILL` request was not valid on the target system.

`ER_SNID` Bad sync ID.
A request to `tetsyncd` specified an invalid sync identifier. This usually indicates an internal program error of some kind.

`ER_SYNCERR` Synchronisation was unsuccessful.
A sync event completed unsuccessfully because one or more of the participants did not synchronise, submitted a `no` sync vote, timed out or terminated unexpectedly.

`ER_SYSID` Bad system ID.
A client running on a system which is not participating in a distributed test purpose tried to participate in an event relating to that test purpose, or no process from a system that was supposed to be participating in an event was logged on to the server.

`ER_TIMEDOUT` Request timed out.
The timeout specified with the request expired before the request could complete.

`ER_TRACE` Tracing not configured.
An `OP_TRACE` request was sent to a server which had been compiled with tracing disabled.

`ER_WAIT` No un-waited-for children to report.
An `OP_WAIT` request with a zero timeout found that the specified process had not yet exited.

`ER_XRID` Bad execution results file ID.
A request to `tetxresd` specified an invalid execution results file identifier. This usually indicates an internal program error of some kind.

E. Mnemonics used in dTET2 diagnostics

E.1 Introduction

This appendix describes some of the mnemonics that may be used in diagnostic and trace messages generated by dTET2 utility programs.

E.2 Process types

Each type of process is assigned a process type identifier.

The following table lists dTET2 process types and their meanings:

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

E.3 Process states

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

The following table lists dTET2 process states and their meanings:

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

E.4 Server request codes

Each request sent by a client to a server process includes a request code. Some requests are implemented in all dTET2 servers, whereas others are specific to a particular server.

The following table lists dTET2 server request codes and their meanings:

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

Request code	Processed by	Description
OP_TSFILES	tccd	Transfer save files
OP_TSINFO	tccd	Send transport-specific information
OP_UNLINK	tccd	Unlink a file
OP_USYNC	tetsyncd	User sync request
OP_WAIT	tccd	Wait for a process
OP_XRES	tetxresd	Write data to an execution results file
OP_XRCLOSE	tetxresd	Close an execution results file
OP_XROPEN	tetxresd	Open an execution results file
OP_XRSYS	tetxresd	Send system name list

E.5 Server reply codes

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

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

E.6 Sync states

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

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

E.7 Execution result states

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

The following table lists execution results states and their meanings:

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

F. Trace and debugging facilities

F.1 Introduction

Each dTET2 utility program contains certain facilities which may be used by an experienced software engineer to trace its operation and for debugging purposes.

When tracing facilities are enabled, messages describing the processing being undertaken are printed to the standard error stream. Although an explanation of the meanings of these messages is beyond the scope of a User Guide, the information in this appendix is presented in the belief that it may be of some use to experienced dTET2 users.

There are two tracing mechanisms implemented in dTET2. One is used only in `tcc` to trace the execution of code derived from the TET. The other is used in the dTET2-specific code to trace the execution of client and server functions.

F.2 Caveats

Trace messages are intended to be interpreted by an experienced software engineer who is familiar with the internal operation of dTET2 processes. In most cases, they need to be interpreted in conjunction with the dTET2 source code.

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

F.3 Local tracing in `tcc`

Local tracing is enabled when `tcc` is invoked with the `-d` option, thus:

`tcc -d` *n other-options* . . .

where *n* is a number whose value determines which kind of tracing is to be performed.

Different types of trace messages are enabled when different bits in *n* are set. The following table lists which types of message are controlled by each bit:

Bit position	Value	Trace message type
0	1	Function calls
1	2	Actions within functions
2	4	Tracing of file descriptors
3	8	Table of un-freed memory
4	16	Memory table including permanent values
5	32	Memory allocation tracing
6	64	Unused
7	128	Scenario syntax check output
8	256	Debug statements within child processes
9	512	Utility functions
10	1024	Signal handling
11	2048	Running resume or rerun option
12	4096	Miscellaneous warning messages

For example:

`tcc -d 3` *other-options* . . .

enables tracing of function calls and actions within functions.

Local tracing in `tcc` is performed by calls to the `DBUG()` macro.

F.4 Client/server function tracing

F.4.1 Description

In each dTET2 process there are several trace flags that can be used to control the emission of debug output from client and server functions. Each trace flag has a value associated with it; generally speaking, the higher the value, the greater the output produced.

The following table lists these flags and the trace message types controlled by them:

Trace flag name	Trace message type
`Tbuf`	memory allocation
`Tio`	message i/o operations
`Tloop`	start and end of the client and server loops
`Tserv`	generic server operation
`Tsyncd`	`tetsyncd` functions
`Ttcc`	`tcc` functions
`Ttccd`	`tccd` functions
`Ttrace`	trace subsystem operation
`Txresd`	`tetxresd` functions

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

At present, trace flags are propagated as follows:

- `tcc` passes trace flags to `tetsyncd` and `tetxresd` on the command line, to `tccd` in an `OP_TRACE` message, and to TCM processes in `TET_TIARGS`.

- `tccd` passes trace flags to TCM processes in `TET_TIARGS`.
- TCM processes pass trace flags to `tccd` in an `OP_TRACE` message.

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

Each global value has a set of bits associated with it which determines to which other processes it will be propagated.

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

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

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

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

The following process indicators are understood:

`M`	`tcc`
`S`	`tccd`
`C`	master TCM/API
`D`	slave TCM/API
`X`	`tetxresd`
`Y`	`tetsyncd`

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

The following trace flag indicators are understood:

`b`	`Tbuf`
`e`	`Tserv`
`i`	`Tio`
`l`	`Tloop`
`m`	`Ttcc`
`s`	`Ttccd`
`t`	`Ttrace`
`x`	`Txresd`
`y`	`Tsyncd`

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

Client/server function tracing is performed by calls to the `TRACE`*n*`()`, `TDUMP()` and `BUFCHK()` macros.

F.4.2 Examples

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

`tcc -TS,s4` ... Set the global `Ttccd` flag to 4 and pass it to `tccd`, which copies the global flag to its local flag.

`tcc -TXY,i6` ... Set the global `Tio` flag to 6 and pass it to `tetsyncd` and `tetxresd`, each of which copy it to their local flag.

`tcc -Ts2` ... Set both the local and global `Tserv` flags to 2; the global flag is passed to all processes, each of which copies it to its local flag.

`tcc -Tall10` ... Set both the global and local versions of each flag to 10. Each global flag is passed to all processes, each of which copies it to its local flag.

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

G. dTET2 manual pages

This appendix contains manual pages for dTET2 utility programs.

CONTENTS