Xlib and X Protocol Test Suite
X Version 11 Release 6.1

Programmers Guide for the X Test Suite

July 1992

Programmers Guide for the X Test Suite

1. Introduction

This document is a Programmers Guide to the X Test Suite.

Instructions for installing and running the X Test Suite are contained in the document "User Guide for the X Test Suite". It is not necessary to read the Programmers Guide in order to install and run the test suite.

2. Purpose of this guide

The information in this section is designed to be used by a programmer intending to review the source code in the revised X Test Suite. It is also intended to be used by an experienced programmer, familiar with the X Window System, to modify or extend the X Test Suite to add additional tests.

Before reading this document, it is necessary to read the document "User Guide for the X Test Suite". This is because the nomenclature used in this document is explained in the "User Guide". Appendix F of that document is a glossary, which explains the meaning of some terms which may not be in common usage.

The directory structure used within the X Test Suite is described further in "Appendix A - Contents of X Version 11 Release 6.1" in the "User Guide". You should be familiar with that appendix before reading further.

3. Contents of this guide

The test set source files in the revised X Test Suite have been developed in the format of a simple language, specially produced as part of the test suite development project. Files in the test suite which use this language have the suffix ".m" and are known as dot-m files.

The syntax of this language is described in the next section of this document entitled "Source file syntax".

During the stage one review of the development project, it was determined that there were some advantages in methods of automatically generating source code for the tests from templates. The dot-m file may be seen as a template for the tests. At the same time, it was important that any utilities associated with the code generation should be commonly available or provided within the test suite.

For this reason, a utility `mc` has been provided to convert dot-m files into C source code and produce Makefiles automatically for the test set. The file formats which may be produced are described in the section entitled "Source file formats". Summaries of usage of the utilies are given in appendices D-F.

As part of the test suite development, a number of conventions were established to define how the syntax of this language should be used in writing test sets. It is useful to understand these conventions in order to understand the structure of the existing tests. You are recommended to use the existing tests as a model, and follow the same structure when modifying or extending the X Test Suite. This is described in the section entitled "Source file structure".

The test set structure has deliberately been kept as simple as possible, and common functions have been developed in libraries. The contents of these libraries is described in the section entitled "Source file libraries".

3.1 Typographical conventions used in this document

The following conventions have been used in this document:

1. Items appearing in angle brackets <> should be substituted with a suitable value.
2. Items appearing in square brackets [] are optional.

4. *Source file syntax*

It is a design requirement that the test code for each test purpose has an associated description of what is being tested (an *assertion*) and a description of the procedure used to test it (a *test strategy*). All of these items are contained in dot-m files, and programs are used to extract the relevant parts. The utilies developed for the purpose are outlined in appendices D-F, and the format of the files they output is described in the section entitled "Source file formats.

The format of a dot-m file consists of a number of sections introduced by keywords. The sections of a dot-m file are as follows:

1. A copyright notice.
2. A section introduced by the >>TITLE keyword. This section defines the name of the function being tested and its arguments.

   The >>TITLE keyword, and the declaration and arguments which follows it, are together known as the "title section".
3. Optionally, there may be a section introduced by the >>MAKE keyword. This section defines additional rules for `make` beyond the default rules.

   The >>MAKE keyword, and the text which follows it, are together known as a "make section".
4. Optionally, there may be a section introduced by the >>CFILES keyword. This section defines additional C source files source files that should be compiled and linked (together with the C source files produced by `mc`) when building a test set.
5. Optionally, there may be a section introduced by the >>EXTERN keyword. This section defines C source code which will be in scope for all test purposes in the test set.

   The >>EXTERN keyword, and the source code which follows it, are together known as an "extern section".
6. For each test purpose in the test set, there will be a section introduced by an >>ASSERTION keyword. These sections each contain the text of an assertion for the function being tested.

   The >>ASSERTION keyword, and the text which follows it, are together known as an "assertion section".
7. An assertion section is normally followed by a section introduced by a >>STRATEGY keyword, which is followed by the strategy. This is a description of how the assertion is tested.

   The >>STRATEGY keyword, and the strategy which follows it, are together known as a "strategy section".
8. The strategy is always followed by a section introduced by a >>CODE keyword, which is followed by a section of C source code which will test whether the assertion is true or false on the system being tested.

   The >>CODE keyword, and the C source code which follows it, are together known as a "code section".

9. Optionally, the >>INCLUDE keyword includes, from a file, one or more of the above sections, at that point in the dot-m file. It terminates the previous section.

The keywords introducing the sections are defined in detail below.

There are also optional keywords which may appear anywhere in a dot-m file:

1. The >>SET keyword sets options which apply in the dot-m file.
2. The >># keyword introduces a comment in a dot-m file.

The optional keywords are also defined in detail below.

4.1 Title section - >>TITLE

4.1.1 Description

```
>>TITLE <function> <section>
```

This keyword must be used at the start of a test purpose immediately after the copyright message.

It may be followed by the declaration of the data type returned and the function arguments. These lines may be omitted if XCALL is never used in any code section in the test set.

1. The line after >>TITLE should specify the data type returned.
2. The next line should specify the function call with any arguments. This is no longer mandatory, since the information is not used in the current version of `mc`. So the second line may be left completely blank.
3. The following lines should specify the arguments and may set them to any expression (not necessarily a constant expression). Usually arguments are set to the return value of a function call, a global variable inserted by the `mc` utility, or a global variable declared in an >>EXTERN section.

   Static variables will be created to match these arguments, and will be reset as specified automatically before the start of each test purpose, by the function `setargs()` (described in the section entitled "Source file formats").

4.1.2 Arguments

function
: This is the name of the function in the X Window System to be tested. In the X Protocol tests it should be the name of an X Protocol request or an X event. In the Xlib tests it should be the name of an Xlib function or an X event.

  The XCALL macro will invoke the named function or macro (which should be in Xlib) with the arguments specified on the lines following >>TITLE.

section
: This is the section of the X Test Suite in which this particular test set is stored. It should be the name of a directory in `$TET_ROOT/tset`. The section name is used in formatting the assertions in the test set, and for output to the journal file for reporting purposes. The section name does not affect the building and execution of the tests.

4.2 Make section - >>MAKE

4.2.1 Description

```
>>MAKE
```

This keyword may be used anywhere in the dot-m file after the end of a section.

It should be followed by lines which will be copied into the Makefile by the `mc` utility, when the Makefile is remade using the `-m` option.

The lines specified will be joined together and placed in order before the first rule in the Makefile. In this way, the lines copied may contain both initialisation of Make variables, and additional rules.

This keyword should be used sparingly, since the additional Makefile lines must be consistent with the rest of the Makefile (see later section entitled "Makefile").

These lines are designed to allow auxiliary programs to be made by `make` (in addition to the default target which is `Test`). For example, if test purposes in the test set execute programs `Test1` and `Test2`, which must also be built, the >>MAKE keyword can be followed by lines which specify rules for building these executable programs. This can be done as follows:

```
>>MAKE
AUXFILES=Test1 Test2
AUXCLEAN=Test1.o Test1 Test2.o Test2

all: Test

Test1 : Test1.o $(LIBS) $(TCMCHILD)
      $(CC) $(LDFLAGS) -o $@ Test1.o $(TCMCHILD) $(LIBLOCAL) $(LIBS) $(SYSLIBS)

Test2 : Test2.o $(LIBS) $(TCMCHILD)
      $(CC) $(LDFLAGS) -o $@ Test2.o $(TCMCHILD) $(LIBLOCAL) $(LIBS) $(SYSLIBS)
```

Notice that the default target `Test` is still made by the new default Make rule. Also, notice that `Test1` and `Test2` are not dependencies for the `all` target. They are dependencies for `Test`. (See the later section entitled "Makefile".)

4.3 Additional source files - >>CFILES

4.3.1 Description

```
>>CFILES <filename> ...
```

This keyword may be used anywhere in the dot-m file after the end of a section.

The list of files following the keyword are taken as the names of C source files that should be compiled and linked (together with the C source files produced by `mc`) when building a test set. This allows code to be split among several files.

The only effect is to alter the Makefile that is produced by `mmkf`.

4.4 Extern section - >>EXTERN

4.4.1 Description

```
>>EXTERN
```

This keyword may be used anywhere in the dot-m file after the end of a section.

It should be followed by lines of C source code which will be copied into the C source file by the `mc` utility, when the C source code is remade.

These lines will be copied unaltered into the C source file before the source code for the first test purpose.

This section is useful for including three types of source code:

1. static variables declarations which are used by a number of test purposes in the test set.
2. static functions which are used by a number of test purposes in the test set.
3. header file inclusions which are needed in addition to the default header file inclusions in the C source code.

4.5 Assertion section - >>ASSERTION

4.5.1 Description

```
>>ASSERTION <test-type> [ <category> [<reason>] ]
```

This keyword is used at the start of each test purpose.

It should be followed by the text of the assertion which is a description of what is tested by this particular test purpose.

The text should not contain troff font change commands (or any other nroff/troff commands). This is because the majority of nroff/troff commands will not be understood by the `ma` utility. However, various macros can be used to enable mapping of the format of special text onto similar fonts to those used in the X Window System documentation. These are described in appendix B.

The keyword xname in the assertion text will be replaced by `mc` with the name of the function under test obtained from the >>TITLE keyword.

Unless the `category` argument specifies an extended assertion, or the `test-type` is `gc` or `def`, the assertion text must be followed by the >>STRATEGY keyword, strategy section, >>CODE keyword and code section. Refer to the description of the `category` argument.

If the >>CODE keyword is missing, following the text of an assertion which is not an extended assertion, the `mc` utility will insert code to produce a result code UNREPORTED for this test purpose when the test set is executed.

4.5.2 Arguments

test-type

Good

This is a "good" test. The function under test is expected to give a successful result.

By convention these assertions appear in the dot-m file before all assertions with test-type Bad.

Bad

This is a "bad" test. The function under test is expected to give an unsuccessful result under the conditions that the test imposes.

By convention these assertions appear in the dot-m file after all assertions with test-type Good.

The assertion text for many of these assertions is included via the .ER keyword, described below.

gc

The assertion text states which gc components affect the function under test. In this case the remaining arguments are unused, and `mc` inserts into the C source file a series of assertions, strategies and test code corresponding to the gc components listed in the assertion text via the macro

```
.M gc-comp ,
```

or

```
.M gc-comp .
```

The assertions, strategies and test code are included, from files in the directory `$TET_ROOT/xtest/lib/gc`.

The .M macro is used, since the gc components correspond to structure members in a gc structure.

def

The assertion text is tested in the test for another assertion. These assertions are often definitions of terms, which cannot be tested in isolation, hence the abbreviation "def". The remaining arguments are unused, and `mc` inserts code into the C file to issue the result code NOTINUSE, and issues a message stating that the assertion is tested elsewhere.

category

This is the assertion category, modelled on the corresponding codes in the document[1] POSIX.3 entitled "Test Methods for Measuring Conformance to POSIX". It is either A, B, C or D.

---

1. Obtainable from Publication Sales, IEEE Service Center, P.O. Box 1331, 445 Hoes Lane, Piscataway, NJ 08854-1331, (201) 981-0060

If the assertion tests a conditional feature, it is categorised as type C or D, otherwise it is categorised as type A or B.

If the assertion is classified as an "extended assertion" it is categorised as type B or D. Otherwise it is categorised as type A or C and is known as a "base assertion".

	Base Assertion	Extended Assertion
Required Feature	A	B
Conditional Feature	C	D

Tests are always required for base assertions. Tests are not required for extended assertions, but should be provided if possible. Extended assertions are used to describe features that may be difficult to test.

In some cases partial testing may be performed for extended assertions. An example is that it may be possible to test that some specific common faults are not present. In this the result code would be FAIL if an error is detected, or UNTESTED if no failure is detected, but the assertion is still not fully tested.

For this reason, the strategy and code sections are optional for extended assertions. If they are not supplied, `mc` will automatically generate source code to put out a result code UNTESTED, with a message which describes the reason. If they are supplied, they will override the automatically generated sections.

Since there is not yet an equivalent document to POSIX.3 for the X Window System then these codes are subject to change. For example, an assertion classified as an "extended assertion" (type B) might become a "base assertion" (type A) if a test method is later identified.

The following table lists the allowed test result codes for each category.

Category	Allowed Result Codes
A	PASS, FAIL, UNRESOLVED
B	PASS, FAIL, UNTESTED, UNRESOLVED
C	PASS, FAIL, UNSUPPORTED, UNRESOLVED
D	PASS, FAIL, UNSUPPORTED, UNTESTED, UNRESOLVED

reason

In the case of extended assertions (category B or D) a reason code must be supplied. These are the same as in POSIX.3. A list of the reason codes, and the corresponding text of the reason, are shown in appendix A.

4.6 Strategy section - >>STRATEGY

4.6.1 Description

```
>>STRATEGY
```

If the category of an assertion is A or C, this keyword must be used immediately after the assertion section.

If the category of an assertion is B or D, this keyword may be optionally be used immediately after the assertion section.

It should be followed by the strategy, which is a description of how the assertion is to be tested.

The text of the strategy is in free format sentences. It may contain the xname keyword, which is an abbreviation for the name of the function under test.

The use of the XCALL keyword in the strategy section as an abbreviation for "Call xname" is discontinued, although in this release, some occurrences remain.

4.7 Code section - >>CODE

4.7.1 Description

```
>>CODE [<BadThing>]
```

This keyword must be used immediately after the strategy section.

It should be followed by the C source code which will test the assertion.

The C source code will be converted by `mc` into a format suitable for the use by the TET API. The way in which this is done is described in the section entitled "Source file formats".

A blank line must separate the declarations of automatic variables in the test function from the first executable statement. There must be no other blank lines within these declarations.

The utility `mc` also expands the XCALL macro to call the function under test, and to call library functions, before and after the function under test, to install and deinstall error handlers and flush pending requests to the X server. The way in which this is done is described in the section entitled "Source file formats".

4.7.2 Arguments

BadThing

This is an optional argument to the >>CODE macro.

If it is omitted, the XCALL macro will cause code to be inserted to ensure that the function under test produces no X Protocol error, and issues a result code FAIL if an error is detected.

If it is set to the symbolic value of an X Window System error code, the XCALL macro will cause code to be inserted to ensure that the function under test produces that X Protocol error, and issues a result code FAIL if the wrong error, or no error, is detected.

4.8 Included section - >>INCLUDE

4.8.1 Description

```
>>INCLUDE <filename>
```

This keyword includes the contents of `filename`, which must be a file containing one or more sections in the dot-m file format, optionally containing a copyright header. The sections in the included file should produce a valid dot-m file, if they were included directly at the point of inclusion. The >>INCLUDE keyword terminates the preceding section - it cannot be used in the middle of a section.

The included sections are processed at the point of inclusion when the C source code is generated by the `mc` utility.

The >>INCLUDE keyword is usually used when including test purposes which are common to more than one test set. The >>INCLUDE mechanism allows an entire test purpose (including assertion, strategy and code sections) to be included.

The >>INCLUDE keyword should not be used for merely including common functions called by a number of test purposes. If the functions are common to one dot-m file, they should be placed in an extern section. If the functions are common to many test sets, they should be placed in one of the X Test Suite libraries.

4.9 Included errors - .ER

4.9.1 Description

```
.ER [Bad]Access grab
.ER [Bad]Access colormap-free
.ER [Bad]Access colormap-store
.ER [Bad]Access acl
.ER [Bad]Access select
.ER [Bad]Alloc
.ER [Bad]Atom [val1] [val2] ... †
.ER [Bad]Color
.ER [Bad]Cursor [val1] [val2] ... †
.ER [Bad]Drawable [val1] [val2] ... †
.ER [Bad]Font bad-font
.ER [Bad]Font bad-fontable
.ER [Bad]GC
.ER [Bad]Match inputonly
.ER [Bad]Match gc-drawable-depth
.ER [Bad]Match gc-drawable-screen
.ER [Bad]Match wininputonly
.ER [Bad]Name font
.ER [Bad]Name colour
.ER [Bad]Pixmap [val1] [val2] ... †
.ER [Bad]Value <arg> [mask] <val1> [val2] ... ‡
.ER [Bad]Window [val1] [val2] ... †
```

† - these arguments are optional.

‡ - the <arg> and at least <val1> argument must be supplied. The mask argument, and additional arguments, are optional.

Note - the `Bad` prefix is in each case optional.

This keyword causes `mc` to insert into the C source file the text of an assertion, and in some cases default strategy and default test code to test for the generation of a particular X Protocol error by an Xlib function.

In some cases there is no strategy and code in the included file, because only the assertion is common - the strategy and code sections are specific to each test purpose, and must be provided immediately after the .ER keyword.

The default strategy and code sections (if included) may be overridden by sections in the dot-m file immediately after the .ER keyword.

Note that this keyword does not insert the >>ASSERTION keyword - this must appear on the line before .ER is invoked. Thus the keyword does not include the entire assertion section.

The assertion text, strategy and test code are included from files in the directory `$TET_ROOT/xtest/lib/error`.

The names of these files, and the assertion text in each file, is shown in appendix C.

4.10 Set options - >>SET

4.10.1 Description

```
>>SET startup <func_startup>
>>SET cleanup <func_cleanup>
>>SET tpstartup <func_tpstartup>
>>SET tpcleanup <func_tpcleanup>
>>SET need-gc-flush
>>SET fail-return
>>SET fail-no-return
>>SET return-value <return_value>
>>SET no-error-status-check
>>SET macro [ <macroname> ]
>>SET begin-function
>>SET end-function
```

These options control how the `mc` utility converts the dot-m file into a C source file.

Except where specifically stated, they may appear anywhere in the dot-m file and apply from that point on, unless reset by a further >>SET keyword with the same first argument.

4.10.2 Arguments

startup
: The name of the function called before all test purposes is to be set to func_startup (rather than the default, startup()).

cleanup
: The name of the function called after all test purposes is to be set to func_cleanup (rather than the default, cleanup()).

tpstartup
: The name of the function called before each test purpose is to be set to func_tpstartup (rather than the default, tpstartup()).

tpcleanup
: The name of the function called after each test purpose is to be set to func_tpcleanup (rather than the default, tpcleanup()).

need-gc-flush
: When the XCALL macro is expanded, code to call the X Test Suite library function gcflush(display, gc) will be inserted after the code to call the function under test.

fail-return
: When the XCALL macro is expanded, code to end the test purpose will be inserted where an error is reported (the default is to continue after an error is reported).

fail-no-return
: When the XCALL macro is expanded, no code to end the test purpose will be inserted where an error is reported (this reverses the effect earlier using

>>SET fail-return).

return-value

When the XCALL macro is expanded, and the Xlib function call has return type `Status`, and the return value of XCALL is not saved for testing in the calling code, code will be inserted to report an error if the function under test does not return <return_value>. (By default, when the Xlib function call has return type `Status`, an error is reported for assertions with test-type Good if the return value is zero, and for assertions with test-type Bad if the return value is non-zero).

no-error-status-check

When the XCALL macro is expanded, the default code to check for X Protocol errors will not be inserted. The test purpose can perform alternative checking after invoking XCALL. This setting only applies up to the end of the current section.

macro

There is a macro in an X Window System header file for which a test set source file will be produced which uses identical test purposes to the function under test. This is used to automatically generate test purposes for the Display and Screen information macros, which are identical to those for the corresponding Xlib functions.

The macro name is set to `<macroname>` - the default is the `function` argument in the >>TITLE keyword, with the leading letter ‘X’ removed.

This option must be specified **before** the title section of the dot-m file.

Note - this option may not be used for macros which have no arguments.

begin-function

The name of an additional function called before each test purpose, after tpstartup() and global function arguments are initialised.

end-function

The name of an additional function called after each test purpose, before tpcleanup().

4.11 Comment lines - >>#

4.11.1 Description

```
>># <Comment text>
```

This keyword specifies a one line comment. These are not intended to replace code comments in the code section - in fact the `mc` utility does not copy dot-m format comments to the C source file. Comments in the dot-m file are used for higher level comments rather than detailed comments. For example, comments are used to record the history of the development of the dot-m file, to preserve previous versions of assertions where necessary, and to draw attention to unresolved problem areas.

5. Source file formats

This section describes the output from various utilities which may be used to format the contents of a dot-m file.

The code-maker utility `mc` builds C source files from a dot-m file. Appendix D gives a usage summary.

The Makefile utility `mmkf` builds Makefiles from a dot-m file. Appendix E gives a usage summary.

The assertion utility `ma` produces a list of assertions from a dot-m file. Appendix F gives a usage summary.

Instructions for building and installing the `mc` utility are given in the "User Guide". When the utility `mc` is built and installed, the utilities `mmkf` and `ma`, which are links to the same program `mc` differing only in name, are automatically installed as well.

5.1 C files for standalone executable - `Test.c`

The command

```
mc −o Test.c stclpmsk.m
```

produces a C file named `Test.c`. This may be compiled to produce a standalone executable file named `Test`. Instructions on building and executing the tests are given in the "User Guide". The `Test.c` files are not provided as part of this release, but are built automatically when building the X Test Suite.

The C file contains all of the interface code required to invoke the test purposes from the TET and a description of the assertion being tested is placed as a comment above the code for each test purpose to make it easy to understand what is being tested.

The remaining parts of this section describe the format of the `Test.c` files in more detail. The descriptions are in the order in which the text is inserted into the `Test.c` file.

Some parts of the `Test.c` file are constructed by copying in template files specifically written to work with `mc`. These files are all located in the directory `$TET_ROOT/xtest/lib/mc`.

5.1.1 Copyright header

A copyright header is inserted as a C source comment block. This will contain lines showing the SCCS versions of the dot-m file and any included files.

5.1.2 SYNOPSIS section

A synopsis defining the arguments of the function being tested is inserted as a C source comment block. This is constructed from the lines following the >>TITLE keyword in the dot-m file. It includes the data type returned and any arguments to the function.

The synopsis section is omitted if there are no lines following the the >>TITLE keyword.

For example:

```
/*
 * SYNOPSIS:
 *   void
 *   XSetClipMask(display, gc, pixmap)
 *   Display *display;
 *   GC gc;
 *   Pixmap pixmap;
 */
```

5.1.3 Include files

For the Xlib tests, when the `section` argument to the >>TITLE keyword is other than `XPROTO`, the contents of the file `mcinclude.mc` are then included.

In this release the contents of this file are as follows:

```
#include    <stdlib.h>
#include    "xtest.h"
#include    "Xlib.h"
#include    "Xutil.h"
#include    "Xresource.h"
#include    "tet_api.h"
#include    "xtestlib.h"
#include    "pixval.h"
```

For the X Protocol tests, when the `section` argument to the >>TITLE keyword is `XPROTO` the contents of the file `mcxpinc.mc` are then included.

In this release the contents of this file are as follows:

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

5.1.4 External variables

For the Xlib tests, when the `section` argument to the >>TITLE keyword is other than `XPROTO`, the contents of the file `mcextern.mc` are then included.

In this release the contents of this file are as follows:

```
extern      Display  *Dsp;
extern      Window   Win;

extern      Window   ErrdefWindow;
extern      Drawable ErrdefDrawable;
extern      GC       ErrdefGC;
extern      Colormap ErrdefColormap;
extern      Pixmap   ErrdefPixmap;
extern      Atom     ErrdefAtom;
extern      Cursor   ErrdefCursor;
extern      Font     ErrdefFont;
```

The external variables are defined in the file startup.c or (when linking a program executed via tet_exec()) in the file ex_startup.c.

For the X Protocol tests, when the section argument to the >>TITLE keyword is XPROTO the contents of the file mcxpext.mc are then included.

In this release this file is empty.

5.1.5 Test set symbol and name

A symbol is defined indicating the function under test.

For example:

```
#define T_XSetClipMask  1
```

You may use this in a #ifdef control line, to distinguish special cases for particular functions in code sections of a dot-m file included via the >>INCLUDE keyword.

The global variable TestName is initialised to the name of the function under test, which is the function argument given to the >>TITLE keyword.

For example:

```
char    *TestName = "XSetClipMask";
```

You can use this within a code section of a dot-m file to obtain the name of the function under test.

5.1.6 Definitions for arguments

Symbols are defined to correspond with any arguments to the function specified in the >>TITLE keyword.

These correspond to the lines following the >>TITLE keyword in the dot-m file.

For example:

```
/*
 * Defines for different argument types
 */
#define A_DISPLAY display
#define A_GC gc
#define A_PIXMAP pixmap
#define A_DRAWABLE pixmap
```

These are used by the code in various included files, to substitute a symbol representing a particular argument type with the actual variable used as the argument by the function under test.

5.1.7 Static variables

Static variables are defined to correspond with any arguments to the function specified in the >>TITLE keyword.

These correspond to the lines following the >>TITLE keyword in the dot-m file.

For example:

```
/*
 * Arguments to the XSetClipMask function
 */
static Display *display;
static GC gc;
static Pixmap pixmap;
```

These are the arguments that will be passed to the function under test when the XCALL macro is expanded.

You can initialise these in a code section of a dot-m file as required prior to invoking the macro XCALL.

These variables will be initialised at the start of each test purpose using the function `setargs()` described below in the section entitled "Initialising arguments".

5.1.8 Test purpose number

You can use this within a code section of a dot-m file to obtain the number of the current test purpose.

```
int     tet_thistest;
```

5.1.9 Initialising arguments

A function `setargs()` is defined to initialise the arguments to the function under test.

Code to call this function is inserted at the start of each test purpose before the code you put in the code section.

The arguments are initialised to have the value of the expression you specified in the title section. This does not have to be a constant expression - for example, it my be a return value of a function in a library or extern section. By default arguments are initialised to zero values.

For example:

```
/*
 * Called at the beginning of each test purpose to reset the
 * arguments to their initial values
 */
static void
setargs()
{
      display = Dsp;
      gc = 0;
      pixmap = 0;
}
```

5.1.10 Initialising arguments when test-type is Bad

A function `seterrdef()` is defined to initialise some of the arguments to the function under test to values which are suitable for conducting the included error tests.

This is called in some of the included error tests to initialise the arguments to known good values.

For example:

```
/*
 * Set arguments to default values for error tests
 */
static void
seterrdef()
{
	gc = ErrdefGC;
	pixmap = ErrdefPixmap;
}
```

5.1.11 Code sections

The code sections in the dot-m file are converted into a sequence of functions named tnnn(), where nnn is a three digit number which is filled with leading zeros if necessary, the first code section being named t001(). This is known as a "test function".

Each of the test functions is preceded by the corresponding assertion for the test purpose, in a C source code comment, labelled with the test purpose number.

Code to call the library function `tpstartup()` is inserted at the start of the test function, immediately after any automatic variables declared in your code section. This function performs initialisation required for each test purpose, including setting error handlers to trap unexpected errors.

Code to call the automatically generated test-set specific function `setargs()` is then inserted. This function is described further in the previous section entitled "Initialising arguments".

The contents of your code section are then inserted.

The macro XCALL in a dot-m file is expanded to call the function under test with arguments corresponding to the lines following the >>TITLE keyword.

For example:

```
		XCALL;
```

is by default expanded to

```
		startcall(display);
		if (isdeleted())
			return;
		XSetClipMask(display, gc, pixmap);
		endcall(display);
		if (geterr() != Success) {
			report("Got %s, Expecting Success", errorname(geterr()));
			FAIL;
		}
```

The stages in this expansion are as follows:

1. The library function `startcall()` is called to check for any outstanding unexpected X protocol errors, which might have been generated, for example, during the setup part of the test. A call to `XSync()` is made to achieve this.
2. The library function `startcall()` installs a test error handler in place of the unexpected X protocol error handler.
3. The library function `isdeleted()` returns `True` if the test purpose has already issued a result code UNRESOLVED due to an earlier call to `delete()`. (This must be done after calling `startcall()` in case `XSync()` flushed an unexpected X protocol error.)
4. The function under test is then called with the arguments listed in the title section.
5. The library function `endcall()` is called to check for any X protocol errors caused by the function under test. A call to `XSync()` is made to achieve this.
6. The library function `endcall()` installs the unexpected X protocol error handler.
7. The test error handler saves the number of the most recent X protocol error. It is accessed by calling the function `geterr()` and the value is checked. The value is expected to be `Success` by default, or `BadThing` if the code section was introduced in the dot-m file by

   ```
   >>CODE BadThing
   ```

   If it is desirable to skip checking the error status at this point, the option

   ```
   >>SET no-error-status-check
   ```

   may be inserted in the dot-m file in the current code section before the XCALL. This setting only applies up to the end of the current section.

5.1.12 TET initialisation code

The `mc` utility adds a reference to the function into an array of functions which can be invoked via the TET API.

For example:

```
struct tet_testlist tet_testlist[] = {
        t001, 1,
        t002, 2,
        t003, 3,
        NULL, 0
};
```

Code to calculate the number of test purposes in the test set is inserted as follows:

```
int     ntests = sizeof(tet_testlist)/sizeof(struct tet_testlist)-1;
```

Finally, the names of the startup and cleanup functions are used to initialise variables used by the TET API. These functions are called before the first test purpose and after the last test purpose. The functions called can be overridden using the options >>SET startup and >>SET cleanup.

The default library functions perform initialisation including reading the TET configuration variables and opening a default client.

Should you override the default startup and cleanup functions, you are recommended to call `startup()` as the first line of your startup function and `cleanup()` as the last line of your cleanup function.

```
void  (*tet_startup)() = startup;
void  (*tet_cleanup)() = cleanup;
```

5.2 C files for standalone executable in macro tests - `MTest.c`

When the dot-m file contains the line

```
>>SET macro
```

the command

```
mc -m -o MTest.c scrncnt.m
```

produces a C file named `MTest.c`. This may be compiled to produce a standalone executable file named `MTest`. Instructions on building and executing the tests are given in the "User Guide". The `MTest.c` files are not provided as part of this release, but are built automatically when building the X Test Suite.

The file `MTest.c` is identical to the file `Test.c` except that a macro (which is expected to be made visible by including the file Xlib.h) is tested instead of the Xlib function named in the title section of the dot-m file.

The macro name is set to the `<macroname>` argument of the >>SET macro option - if there is no >>SET macro option in the file, or no argument specified, the default is the `function` argument in the >>TITLE keyword, with the leading letter 'X' removed.

5.3 C files for linked executable - `link.c`

The command

```
mc -l -o link.c stclpmsk.m
```

produces a C file named `link.c`. This is identical to the `Test.c` file with the exception of the initialisation code which enables the source code to be compiled and linked into a space-saving executable file. This is an executable file which may invoke any of the test purposes in the various link.c files, thereby reducing the number of executable files required, and saving space. The `link.c` files are not provided as part of this release, but are built automatically when building the X Test Suite.

The remaining parts of this section describe the differences in format of the `link.c` and `Test.c` files.

The differences are associated with the TET initialisation code being in a separate source file named `linktbl.c`, rather than in the test set source file.

A `linktbl.c` file is provided for each section of the X Test Suite in each subdirectory of `$TET_ROOT/xtest/tset`. This file contains a pointer to an array of `linkinfo` structures, one for each test set in the section. Each `linkinfo` structure contains the following items:

`name`	a unique name for that test set (the name of the test set directory).
`testname`	the actual Xlib function tested by the test set.
`ntests`	the number of test purposes in the test set.
`testlist`	a pointer to the array of test functions constructed for that test set from the contents of the link.c file.
`localstartup`	a pointer to the startup function specific to that test set.
`localcleanup`	a pointer to the cleanup function specific to that test set.

Later in this section there are example values of these structure members.

When the space-saving executable is executed, the TET initialisation code in the library function `linkstart.c` determines which test set is required. This is done by matching `argv[0]` with a `name` element in the array of `linkinfo` structures. The test functions specified by the corresponding `testlist` element of the `linkinfo` structure are then executed, preceded and followed by the corresponding startup and cleanup function respectively.

5.3.1 Test set symbol and name

The global variable `TestName` is made static.

```
static char    *TestName = "XSetClipMask";
```

5.3.2 Test purpose number

This is defined in `linktbl.c`, and is made available via the following code:

```
extern int    tet_thistest;
```

5.3.3 TET initialisation code

The global variable `tet_testlist` is made static.

For example:

```
static struct tet_testlist tet_testlist[] = {
      t001, 1,
      t002, 2,
      t003, 3,
      NULL, 0
};
```

The global variable `ntests` is made static.

```
static int    ntests = sizeof(tet_testlist)/sizeof(struct tet_testlist)-1;
```

The `linkinfo` structure specific to this test set is defined.

For example:

```
struct linkinfo EXStClpMsk = {
        "stclpmsk",
        "XSetClipMask",
        &ntests,
        tet_testlist,
        0,
        0,
};
```

The TET variables for the startup and cleanup functions are defined in `linktbl.c`, and are made available via the following code:

```
extern void    (*tet_startup)();
extern void    (*tet_cleanup)();
```

5.4 C files for linked executable in macro tests - `mlink.c`

When the dot-m file contains the line

```
>>SET macro
```

the command

```
mc -m -l -o mlink.c scrncnt.m
```

produces a C file named `mlink.c`. The `mlink.c` files are not provided as part of this release, but are built automatically when building the X Test Suite.

The file `mlink.c` is identical to the file `link.c`, except that a macro (which is expected to be made visible by including the file Xlib.h) is tested instead of the Xlib function named in the title section of the dot-m file.

The macro name is set to the `<macroname>` argument of the >>SET macro option - if there is no >>SET macro option in the file, or no argument specified, the default is the `function` argument in the >>TITLE keyword, with the leading letter ‘X’ removed.

5.5 Makefile

The command

```
mmkf -o Makefile scrncnt.m
```

produces a Makefile which can be used to build all the C source files described in the previous sections and to build the test executables from the C files.

Further instructions appear in the "User Guide" in the section entitled "Building, executing and reporting tests without using the TET".

The Makefiles produced by `mc` are portable in that they use symbolic names to describe commands and parameters which may vary from system to system. The values of these symbolic names are all obtained by a utility `pmake` from the build configuration file, which is described in the "User Guide" in the section entitled "Configuring the X Test Suite".

The targets in the Makefile which can be invoked by `pmake` are as follows:

`pmake Test`
: Builds standalone executable version of the test set.

`pmake Test.c`
: Builds `Test.c` using `mc` with the format described in the earlier section entitled "C files for standalone executable - `Test.c`".

`pmake MTest`
: Builds standalone executable version of the test set to test the macro version of the function.

`pmake MTest.c`
: Builds `MTest.c` using `mc` with the format described in the earlier section entitled "C files for standalone executable - `MTest.c`".

`pmake linkexec`
: Builds the object files and links which can be used to produce a linked executable file. These targets are used when building space-saving executables as described in the "User Guide".

`pmake link.c`
: Builds `link.c` using `mc` with the format described in the earlier section entitled "C files for linked executable - `link.c`".

`pmake mlink.c`
: Builds `mlink.c` using `mc` with the format described in the earlier section entitled "C files for linked executable - `mlink.c`".

`pmake clean`
: This removes object files and temporary files from the test set directory.

`pmake clobber`
: This removes object files and temporary files from the test set directory and additionally removes all the source files which `mc` can remake.

The remaining parts of this section describe the format of the Makefiles in more detail.

Refer to the section entitled "Make section - >>MAKE" for examples of how the variables AUXFILES and AUXCLEAN may be set.

5.5.1 Copyright header

A copyright header is inserted as a comment block. This will contain lines showing the SCCS versions of the dot-m file and any included files.

5.5.2 Make variables

A series of `make` variables are initialised to represent the names of the source, object and executable files.

```
SOURCES=scrncnt.m
CFILES=Test.c
OFILES=Test.o
MOFILES=MTest.o
LOFILES=link.o mlink.o
LINKOBJ=scrncnt.o
LINKEXEC=scrncnt
```

5.5.3 *Targets for X Protocol tests*

For the X Protocol tests, when the `section` argument to the >>TITLE keyword is `XPROTO` the contents of the file `mmxpinit.mc` are then included.

This file initialises various `make` variables to specific values for the X Protocol tests.

In this release the contents of this file are as follows:

```
#
# X Protocol tests.
#

# CFLAGS - Compilation flags specific to the X Protocol tests.
#
CFLAGS=$(XP_CFLAGS)
SYSLIBS=$(XP_SYSLIBS)
LIBS=$(XP_LIBS)

# LINTFLAGS - Flags for lint specific to the X Protocol tests.
#
LINTFLAGS=$(XP_LINTFLAGS)
LINTLIBS=$(XP_LINTLIBS)
```

5.5.4 *Targets for standalone executable* - `Test`

The contents of the file `mmsa.mc` are included. These are the targets to create the standalone executable file `Test`.

```
#
# Build a standalone version of the test case.
#
Test: $(OFILES) $(LIBS) $(TCM) $(AUXFILES)
	$(CC) $(LDFLAGS) -o $@ $(OFILES) $(TCM) $(LIBLOCAL) $(LIBS) $(SYSLIBS)

Test.c: $(SOURCES)
	$(CODEMAKER) -o Test.c $(SOURCES)
```

5.5.5 *Targets for standalone executable* - `MTest`

If the dot-m file contains the >>SET macro option, the contents of the file `mmmsa.mc` are included. These are the targets to create the standalone executable file `MTest` for the macro version of the specified Xlib function.

```
#
# Build a standalone version of the test case using the macro version
# of the function.
#
MTest: $(MOFILES) $(LIBS) $(TCM) $(AUXFILES)
	$(CC) $(LDFLAGS) -o $@ $(MOFILES) $(TCM) $(LIBLOCAL) $(LIBS) $(SYSLIBS)

MTest.c: $(SOURCES)
	$(CODEMAKER) -m -o MTest.c $(SOURCES)
```

5.5.6 Targets for linked executable

The contents of the file `mmlink.mc` are included. These are the targets to create object files and links which can be used to produce a linked executable file. These targets are used when building space-saving executables as described in the "User Guide".

```
#
# A version of the test that can be combined together with
# all the other tests to make one executable.  This will save a
# fair amount of disk space especially if the system does not
# have shared libraries.  Different names are used so that
# there is no possibility of confusion.
#
link.c: $(SOURCES)
	$(CODEMAKER) -l -o link.c $(SOURCES)

# Link the objects into one large object.
#
$(LINKOBJ): $(LOFILES)
	$(LD) $(LINKOBJOPTS) $(LOFILES) -o $(LINKOBJ)

# Link the object file into the parent directory.
#
../$(LINKOBJ): $(LINKOBJ)
	$(RM) ../$(LINKOBJ)
	$(LN) $(LINKOBJ) ..

# Make a link to the combined executable.
#
$(LINKEXEC): ../Tests
	$(RM) $(LINKEXEC)
	$(LN) ../Tests $(LINKEXEC)

../Tests: ../$(LINKOBJ)

linkexec:: $(LINKEXEC) $(AUXFILES) ;
```

5.5.7 Targets for linked executable - macro version

If the dot-m file contains the >>SET macro option, the contents of the file `mmmlink.mc` are included. These are the targets to create object files and links for the macro version of the specified Xlib function which can be used to produce a linked executable file. These targets are used when building space-saving executables as described in the "User Guide".

```
# A version of the test that can be combined with all the other tests for
# the macro version of the function.
#
mlink.c: $(SOURCES)
	$(CODEMAKER) -m -l -o mlink.c $(SOURCES)

linkexec:: m$(LINKEXEC) $(AUXFILES) ;

m$(LINKEXEC): ../Tests
	$(RM) m$(LINKEXEC)
	$(LN) ../Tests m$(LINKEXEC)
```

5.5.8 Targets for libraries

For the Xlib tests, when the `section` argument to the >>TITLE keyword is other than `XPROTO`, the contents of the file `mmlib.mc` are then included.

In this release the contents of this file are as follows:

```
#
# This part of the makefile checks for the existance of the libraries
# and creates them if necessary.
#

# The xtestlib is made if it doesn't exist
#
$(XTESTLIB):
	cd $(XTESTROOT)/src/lib; $(TET_BUILD_TOOL) install

# The fontlib is made if it doesn't exist
#
$(XTESTFONTLIB):
	cd $(XTESTROOT)/fonts; $(TET_BUILD_TOOL) install
```

For the X Protocol tests, when the `section` argument to the >>TITLE keyword is `XPROTO` the contents of the file `mmxplib.mc` are then included. This file is identical to `mmlib.mc` except for the following additional lines:

```
# The X Protocol test library is made if it doesn't exist
#
$(XSTLIB):
	cd $(XTESTROOT)/src/libproto; $(TET_BUILD_TOOL) install
```

5.5.9 Targets for cleaning and linting

The contents of the file `mmmisc.mc` are then included.

This file includes a `clean` target to remove object files and temporary files, and a `clobber` target which additionally removes all the source files which `mc` can remake.

There is also a `LINT` target which enables the C source files to be checked against lint libraries specified in the build configuration file.

```
#
# Miscellaneous housekeeping functions.
#

# clean up object and junk files.
#
clean:
	$(RM) Test $(OFILES) $(LOFILES) $(LINKOBJ) $(LINKEXEC) core\
		MTest m$(LINKEXEC) $(MOFILES) CONFIG Makefile.bak $(AUXCLEAN)

# clobber - clean up and remove remakable sources.
#
clobber: clean
	$(RM) MTest.c Test.c mlink.c link.c Makefile

# Lint makerules
#
lint: $(CFILES)
	$(LINT) $(LINTFLAGS) $(CFILES) $(LINTTCM) $(LINTLIBS)

LINT:lint
```

5.5.10 Targets for building known good image files

The contents of the file `mmpgen.mc` are then included.

These include targets which enable the test set to be built so that it generates known good image files.

These are not intended to be used outside the development environment at UniSoft.

```
#
# Pixel generation makerules for generating the reference
# known good image files.
#

PVOFILES=pvtest.o

pvgen: $(PVOFILES) $(PVLIBS) $(TCM)
	$(CC) $(LDFLAGS) -o $@ $(PVOFILES) $(TCM) \
	$(PVLIBS) $(SYSLIBS) $(SYSMATHLIB)

pvtest.o: pvtest.c
	cc -c -DGENERATE_PIXMAPS $(CFLAGS) pvtest.c

pvtest.c: Test.c
	$(RM) pvtest.c; \
	$(LN) Test.c pvtest.c
```

5.5.11 Targets for included files

Rules are included to specify the dependency of the C source files on any included files.

For example:

```
Test.c link.c: $(XTESTLIBDIR)/error/EAll.mc
Test.c link.c: $(XTESTLIBDIR)/error/EGC.mc
Test.c link.c: $(XTESTLIBDIR)/error/EPix.mc
```

5.6 Formatting assertions

The command

```
ma -o stclpmsk.a -h -m stclpmsk.m
```

produces in the file `stclpmsk.a` a list of the assertions from the assertion sections of the specified dot-m file. The assertions are output in nroff format. All macros used in the assertion text can be obtained using the -h and -s options as described below.

The remaining parts of this section describe the output format in more detail.

5.6.1 Copyright header

A copyright header is output as an nroff comment block. This will contain lines showing the SCCS versions of the dot-m file and any included files.

5.6.2 Macro definitions

If the -h option was specified, macros that are later used in the assertion text will be output from the file `maheader.mc`.

5.6.3 Title

The line

```
.TH <function> <section>
```

is output, where `<function>` and `<section>` are obtained from the title section of the dot-m file.

The default macro definition for .TH in `maheader.mc` causes the section and function name to be printed at the top of each page.

5.6.4 Assertions

For each assertion section, the line

```
.TI <category> \" <function>-n
```

is output, where `<category>` is obtained from the second argument of the >>ASSERTION keyword and `<function>` is obtained from the title section of the dot-m file, and `n` is the number of the assertion in the test set.

This is followed by the assertion text in which `xname` is converted to `<function>`. For example:

```
.TI A \" XSetClipMask-1
A call to
.F XSetClipMask
sets the
.M clip_mask
component of the specified GC to the value of the
.A pixmap
argument.
```

The other macros used in the assertion text to control font changes are described in appendix B.

The default macro definition for .TH in `maheader.mc` causes the example assertion to be printed as follows:

Assertion XSetClipMask-1(A).
A call to `XSetClipMask` sets the *clip_mask* component of the specified GC to the value of the *`pixmap`* argument.

6. *Source file structure*

This section describes the C source coding style and conventions which which have been used in the development of the revised X Test Suite. These conventions apply to the structure of the code sections of the dot-m files, whose overall structure is defined in previous sections of the Programmers Guide. In some cases (particularly in the structure of the X Protocol tests) the style and conventions have been deleloped from the earlier T7 X Test Suite.

You are advised to study the contents of this section before attempting to modify or extend the X Test Suite. The contents of this section will give you guidelines on how to structure the test code so that it is easy to follow, gives correct and reliable information when the tests are executed, and is written as compactly as possible.

Libraries of common functions have been used and further developed in the revised X Test Suite in order to keep the source code in the test sets as compact as possible. The rest of this section describes recommendations on how particular library functions should be used. It does not describe the contents of the libraries in detail. A complete list of library contents is provided in the section entitled "Source file libraries".

During the development of the Xlib tests, a library of support functions has been developed. This library includes functions for performing common operations required when testing the X Window System, as well as performing common reporting operations. This library includes a small number of functions developed for the Xlib tests within the T7 X Test Suite. This library is known as the "X test suite library" in this document, and the source of the library is in the directory `$TET_ROOT/xtest/src/lib`.

Calls to any function in this library may be made by any test set in the X Test Suite.

6.1 *Structure of the Xlib tests*

This section describes the structure of the code sections of the Xlib tests.

The Xlib tests are the tests for sections 2 to 10 of the X11R4 Xlib specifications. They are stored in subdirectories of the directories `CH02` to `CH10` (which are to be found in the directory `$TET_ROOT/xtest/tset`). There is a subdirectory for each Xlib function containing a dot-m file which includes all the test purposes provided for that Xlib function. The naming scheme which is used for these directories is described in appendix B of the "User Guide".

6.1.1 *Result code macros*

It is good practice where possible to structure the test so that only one test result code is assigned before the code section returns or ends.

The significance of the various test result codes that may be assigned are described more fully in appendix D of the "User Guide".

The following macros may be used to assign the test result code. These macros call the function `tet_result()` which is part of the TET API.

PASS

This assigns test result code PASS.

FAIL
: This assigns test result code FAIL.

UNRESOLVED
: This assigns test result code UNRESOLVED.

UNSUPPORTED
: This assigns test result code UNSUPPORTED.

UNTESTED
: This assigns test result code UNTESTED.

NOTINUSE
: This assigns test result code NOTINUSE.

WARNING
: This assigns test result code WARNING.

FIP
: This assigns test result code FIP.

Note that there are two other test result codes which may not be assigned directly within a test purpose.

The result code UNINITIATED will be assigned to a test purpose from within the TET when the function `tet_delete()` has been called in an earlier test purpose or startup function. This is useful to prevent initiation of later test purposes when it is not possible to continue executing test purposes in the test set.

The result code NORESULT will be assigned to a test purpose from within the TET if the test purpose is initiated but no result code has been output by the time control returns from the test purpose to the TET.

The FAIL macro also increments a failure counter which is used to prevent a result code being assigned in a later call to CHECKPASS (see below).

6.1.2 Result code functions

There are a series of convenience functions which output a particular test result code preceded by a test information message of type REPORT. (See "Outputting test information messages", below).

In each case the arguments are exactly like those for *printf(3)*.

These are as follows:

untested()
: This function may be used for an extended assertion to output the test result code UNTESTED, preceded by a message.

unsupported()
: This function may be used for a conditional assertion to output the test result code UNSUPPORTED, preceded by a message.

notinuse()
: This function may be used to output the test result code NOTINUSE, preceded by a message.

delete()
: This function may be used to output the test result code UNRESOLVED, preceded by a message.

6.1.3 Assigning result codes

The code should be structured such that a PASS result code is only assigned if there is no doubt that the assertion being tested has been determined to be positively true on the system being tested. Absence of failure should not be taken as proof of success. For this reason, there should if possible be just one place where a PASS result may be assigned, whilst there may be many code paths which report other result codes.

The result code FAIL should not be called until the function under test has been called.

During execution of the test purpose, it may not be possible to setup the conditions for the assertion to be conclusively tested. In this case the result code UNRESOLVED should be assigned rather than FAIL.

For example:

```
>>CODE

        if (setup()) {
              delete("setup() failed; the test could not be completed");
              return;
        }

        ret = XCALL;

        if (ret == 0)
              PASS;
        else
              fAIL;
```

6.1.4 Assigning result codes for extended assertions

Extended assertions are described in more detail as part of the >>ASSERTION keyword.

In some cases partial testing may be done for extended assertions. In this case, the result code would be FAIL if an error is detected, or UNTESTED if no failure is detected but the assertion is still not fully tested.

For example:

```
>>CODE

        ret = XCALL;

        if (ret == 0)
              PASS;
        else
              untested("The assertion could not be completely tested");
```

6.1.5 Assigning result codes for conditional assertions

Conditional assertions are described in more detail as part of the >>ASSERTION keyword.

It is usual to determine at the beginning of the test purpose whether the conditional feature described in the assertion is supported.

For example:

```
>>CODE

	if (!feature_supported) {
		unsupported("The required feature is not supported");
		return;
	}

	ret = XCALL;

	if (ret == 0)
		PASS;
	else
		FAIL;
```

6.1.6 Assigning result codes for multi-part tests

It is often the case that the test strategy for an assertion requires a number of separate checks to be performed, all of which must pass before the test purpose can be assigned a PASS result code.

In order to ensure that all relevant checks have been performed, a macro CHECK is provided which increments a pass counter. At the end of the test, a further macro CHECKPASS checks that the counter has reached the required value before assigning a PASS result. (The expected value of the pass counter is normally a constant, but may be a function of a loop counter if the test involves calling CHECK in a loop.)

The macro CHECK uses `trace()` to print the pass counter and line number in the TET journal file. The format of the TET journal file is described further in appendix C of the "User Guide".

CHECKPASS also ensures that the pass counter is not zero, and that the fail counter is zero.

For example:

```
>>CODE

        n_ret = -1;
        ret = XCALL;

        if (ret == 0)
              CHECK;
        else
              FAIL;

        if (n_ret == expected_number)
              CHECK;
        else
              FAIL;

        CHECKPASS(2);
```

In the case of extended assertions, the macro CHECKUNTESTED may be called, which is identical to CHECKPASS, except that the final result code assigned will be UNTESTED.

6.1.7 Outputting test information messages

Test information messages are normally output to describe the reason for any test result codes which are other than PASS, and for other purposes, as described in this section.

Appendix D of the "User Guide" describes the four different categories of test information messages which may appear in the TET journal file. This section describes how these messages are output from the test purpose.

The functions described in this section call the function `tet_infoline()` which is part of the TET API.

REPORT

A test information message with type REPORT is used to report the reason for any test result code which is other than PASS. A warning message is printed by the report writer `rpt` if a test information message of type REPORT is given in a test purpose which produced a test result code PASS.

This is output using the function `report()`, which takes arguments exactly like those for *printf(3)*.

CHECK

A test information message with type CHECK should not be output directly - this should only be done via the CHECK macro.

TRACE

A test information message with type TRACE is used to describe the state of the test being executed.

This is not output to the TET journal file if the execution configuration parameter XT_OPTION_NO_TRACE is set to Yes.

This is output using the function `trace()`, which takes arguments exactly like those for *printf(3)*.

DEBUG

A test information message with type DEBUG is a debug message inserted during the development of the test.

This is only output to the TET journal file if the value of the execution configuration parameter XT_DEBUG is greater than or equal to the level of the debug message.

This is output using the function `debug()`, which takes arguments exactly like those for *printf(3)*, except that the *printf(3)* arguments are preceded by a single argument which is the debug level. The debug level should be between 1 and 3.

For example:

```
>>CODE

      debug(1, "about to call %s", TestName);
      ret = XCALL;

      if (ret == 0)
            trace("%s returned %d", TestName, ret);
            PASS;
      } else {
            report("%s returned %d instead of 0", TestName, ret);
            FAIL;
      }
```

6.1.8 Creating new test purposes

You can create new test purposes within an existing dot-m file using the guidelines in this section.

It is expected that in doing this you will be primarily aiming to produce new test purposes for a particular Xlib function. You should add the new test purpose to the dot-m file containing the test purposes for that Xlib function.

6.1.8.1 Creating new sections in the dot-m file

You are advised to create an assertion section and strategy section at the end of the file, using as a template one of the existing sections in the dot-m file.

You should then create a code section commencing with the >>CODE keyword. Since there are many different styles of Xlib functions which may be tested, there are few additional guidelines that can be given beyond those contained in earlier parts of this guide.

6.1.8.2 Creating test purposes which use pixmap verification

If you have not done so yet, refer to the section entitled "Examining image files" in the "User Guide". This explains some background to the pixmap verification scheme, and in particular how to view image files produced when running the X Test Suite.

A number of test purposes supplied in the X Test Suite use a scheme known as pixmap verification, to compare the image produced by the X server with a known good image which is stored in a known good image file.

All the required known good image files for the test programs in the X Test Suite (as supplied) have been created in advance. The known good image files for each test program are supplied in the X Test Suite in the test set directory in which the dot-m file is supplied. They are named `annn`.dat, where `nnn` is the number of the test purpose for which the known good image file was generated.

The known good image files are generated as follows. The X Test Suite is compiled with the additional compilation flag -DGENERATE_PIXMAPS, and linked with a replacement Xlib which determines analytically the expected X server display contents at any point. At the points where pixmap verification is going to be performed, the expected image is instead written to a data file, which is the known good image file.

It is not possible to generate further known good image files in this way, because the replacement Xlib is not part of the X Test Suite.

However, it is possible to write a server-specific image file containing the contents of the X server display at points where pixmap verification is going to be performed. This may be useful for the purposes of validation and regression testing against a known server. This may be done by working through the following stages:

1. Create the test purpose with a call to the macro PIXCHECK at the point where you want to validate the image displayed by the X server. Note that the macro PIXCHECK calls the macros CHECK or FAIL depending on whether the image displayed by the X server matches that in the image file. The code invoked by the macro PIXCHECK(display, drawable) is as follows:

```
if (verifyimage(display, drawable, (struct area *)0))
      CHECK;
else
      FAIL;
```

   The function `verifyimage()` is described in more detail in the section entitled "Source file libraries".

2. Build and execute the test without using the TCC, (refer to the "User Guide") and check that the newly created test purpose gives result code UNRESOLVED due to the absence of a known good image file as follows:

```
pmake
pt
prp
```

3. Rerun the test, saving the image produced by the X server as follows:

```
pt -v XT_SAVE_SERVER_IMAGE=Yes
```

4. This should create a file named `annn`.sav, where `nnn` is the name of the newly created test purpose. This is a server-specific image file. Rename this file to the name used for known good image files as follows:

```
mv annn.sav annn.dat
```

5. Check that the process has worked by executing the test without using the TCC, and enabling pixmap verification against the server-specific image file as follows:

   ```
   pt
   prp
   ```

   The newly created test purpose should give a result code of PASS.

It is particularly important that new test purposes are added at the end of the file if an earlier test purpose calls the macro `PIXCHECK`. This is because inserting a test purpose before another test purpose will cause the later test purpose to be renumbered. As well as causing unnecessary confusion in other ways, this will cause the later test purpose to now look for the wrong known good image file.

6.2 Structure of the X Protocol tests

This section describes the structure of the code sections of the X Protocol tests.

The X Protocol tests are the touch tests for the X Protocol (version 11). They are stored in subdirectories of the directory `XPROTO` (which is to be found in the directory `$TET_ROOT/xtest/tset`). There is a subdirectory for each X Protocol request containing a dot-m file which includes all the test purposes provided for that X Protocol request. The naming scheme which is used for these directories is described in appendix B of the "User Guide".

During the development of the X Protocol tests, extensive use has also been made of a library of support functions developed in the earlier T7 X Test Suite. This library is known as the "X Protocol library" in this document, and the source of the library is in the directory `$TET_ROOT/xtest/src/libproto`.

Calls to any function in this library may be made by any of the X Protocol tests in the X Test Suite.

6.2.1 Structure of the code sections

In the T7 release of the X Test Suite, each of the X Protocol tests consisted of a `main()` function which called library functions to send an X Protocol request to the X server, and checked for the correct response (reply, error or nothing).

In the revised X Test Suite, the test code originally in the `main()` function has been moved to a function called `tester()` which is located in an >>EXTERN section in each dot-m file, so that it can be called from each test purpose as described below. The test function `tester()` is in turn called from a library function `testfunc()`.

For example:

```
>>CODE

        test_type = GOOD;

        /* Call a library function to exercise the test code */
        testfunc(tester);
```

By default, the library function `testfunc()` calls `tester()` for each byte orientation. The test function `tester()` is called in a sub-process via the TET API function `tet_fork()`, and returns the exit status of the test process to `testfunc()`.

If required, the execution configuration parameter XT_DEBUG_BYTE_SEX may be set to NATIVE, REVERSE, MSB or LSB to call `tester()` just once with the required byte orientation.

Each client has a test type, which is initialised when the client is created. The test type determines whether X Protocol requests sent by the client are to be good requests or invalid requests (expecting an X Protocol error to be returned). The test type may be modified during the lifetime of the client by invoking the macro `Set_Test_Type()`. In many tests, this is done by setting the test type to one of the following values before calling `Send_Req()`, then setting it to GOOD immediately afterwards for subsequent library calls:

GOOD
: The request sent will be a known good X Protocol request (unless otherwise modified in `tester()` before calling `Send_Req()`).

BAD_LENGTH
: The request sent will have length field less than the minimum needed to contain the request.

JUST_TOO_LONG
: The request sent will have length field greater than the minimum needed to contain the request (and, for requests where the length is used to determine the number of fields in the request, the length is also not the minimum length plus a multiple of the field size).

TOO_LONG
: The request sent will have a length field which is greater than that accepted by the X server under test.

BAD_IDCHOICE1
: The request sent will have a resource ID that is already in use (it is the responsibility of the function `tester()` to ensure the resource ID is in use before calling `Send_Req()`).

BAD_IDCHOICE2
: The request sent will have a resource ID that is out of range (it is the responsibility of the function `tester()` to ensure the resource ID is out of range before calling `Send_Req()`).

BAD_VALUE
: The request sent will have an invalid mask bit set (it is the responsibility of the function `tester()` to ensure the mask field contains an invalid bit before calling `Send_Req()`).

OPEN_DISPLAY
: A special value used only in the test for OpenDisplay for testing the connection setup protocol.

SETUP
: The initial test type of a client, which will cause errors during test setup to produce result code UNRESOLVED rather than FAIL.

6.2.2 Outputting test information and result code

Errors may be detected and reported both within the test function `tester()` and within library functions. When an error is detected, the function `Log_Err()` should be called. This increments an error count and uses `report()` to output a test information message of type REPORT to the TET journal file.

If no error is detected, the function Log_Trace() may be called to record that the expected response was received. This uses `trace()` to output a test information message of type TRACE to the TET journal file.

You can also use the function Log_Debug() to output more detailed test information such as the contents of request, reply and event structures. This uses `debug()` to output a test information message of type DEBUG at level one to the TET journal file.

The function `Exit()` should be called at any point after an error has occurred, which will assign a test result code FAIL and print the error count (or UNRESOLVED if the error counter is zero). The exit status will be EXIT_FAIL in this case.

If `tester()` performs all checks and the results are correct, the function `Exit_OK()` should be called. The exit status will be EXIT_SUCCESS in this case.

A result code of PASS is only assigned to a test purpose in the library function `testfunc()` if all calls to `tester()` give exit status EXIT_SUCCESS. It should not be assigned anywhere else.

6.2.3 Creating new test purposes

You can create new test purposes within an existing dot-m file using the guidelines in this section.

It is expected that in doing this you will be primarily aiming to produce new test purposes for a particular X Protocol request. You should add the new test purpose to the dot-m file containing the test purposes for that X Protocol request.

6.2.3.1 Creating new sections in the dot-m file

You are advised to create an assertion section and strategy section at the end of the file, using as a template one of the existing sections in the dot-m file.

You should then create a code section which passes a function `my_test()` you are about to create to the library function `testfunc()`.

For example:

```
>>CODE

        test_type = GOOD;

        /* Call a library function to exercise the test code */
        testfunc(my_test);
```

6.2.3.2 Creating a new test function

You should create a function `my_test()` in an >>EXTERN section in the dot-m file using the guidelines in this section.

A client is a connection to the X server under test. Each X Protocol request is sent from a particular client. You can create a client numbered `client` using `Create_Client(client)`. Normally a single client is created, but it is possible to create more than one client. This will be necessary when testing the effect on the server of multiple clients.

The client data structure `Xst_clients` is used to store the information about each client you have created. This includes resource ID's and a display structure which is filled in when the client is created. The client data structure is documented in more detail in the header file in which it is defined (`$TET_ROOT/xtest/include/Xstlib.h`).

Next you will need to create a request structure. The function `Make_Req(client, req_type)` should be called to create a request of a specified type `req_type` for a specified client `client` and return a pointer to the request structure. The request structure will be filled in with defaults which may be suitable for the test purpose you are creating. The file `MakeReq.c` in the X Protocol library fills in the default values.

Should you want to change the defaults you can do this at any point between creating the request structure and sending it to the X server. It may be modified by accessing the structure members. The format of the request structures is exactly as defined in your X Protocol header file (normally `/usr/include/X11/Xproto.h`). You can alter value list items using the following functions:

`Add_Masked_Value()`

`Del_Masked_Value()`

`Clear_Masked_Value()`

`Add_Counted_Value()`

`Clear_Counted_Value()`

`Add_Counted_Bytes()`

When you have the request structure you wish to pass to the X server, call the function `Send_Req(client)`. This sends the request `req` from client `client` to the X server, and handles byte swapping and request packing as necessary. If you wish the X Protocol library to further modify the request structure to send an invalid protocol request, set the test type of the client before calling `Send_Req(client)` using the macro `Set_Test_Type(client)`. The possible test types are listed in the earlier section entitled "Structure of the code sections".

To check that the X server has reacted correctly to the request sent, you will need to call the function `Expect()`. For convenience, a number of macros and functions have been created to call `Expect()` depending on the outcome you are expecting. These are as follows:

`Expect_Event(client, event_type)`

> This expects an event of type `event_type` to be sent back from the X server to client `client`.

`Expect_Reply(client, req_type)`
: This expects a reply to the X Protocol request of type `req_type` to be sent back from the X server to client `client`.

`Expect_Error(client, error_type)`
: This expects an error of type `error_type` to be sent back from the X server to client `client`.

`Expect_BadLength(client)`
: This expects a BadLength error to be sent back from the X server to client `client`.

`Expect_BadIDChoice(client)`
: This expects a BadIDChoice error to be sent back from the X server to client `client`.

`Expect_Nothing(client)`
: This expects neither an error, event or reply to be sent back from the X server to client `client`.

The `Expect()` function will check that the response from the X server is of the correct type and has the correct length. It will be byte swapped and unpacked as necessary into an event or reply structure, to which a pointer will be returned.

It is recommended that one of these functions be called immediately after sending an X Protocol request to the X server. This causes any pending response from the X server to be flushed out, and checked. This makes it easier to locate wrong responses from the X server. This is effectively designing the test to run synchronously.

Once an error, event or reply has been returned, it can be examined directly.

Since the structures allocated for requests, replies and events are allocated dynamically, it is wise to free the structure after use. this may be done using the functions `Free_Req()`, `Free_Reply()` and `Free_Event()`.

When the outcome of sending the X Protocol request has been assessed, you will want to either report an error or output a trace message indicating that the expected response was received. Refer to the earlier section entitled "Outputting test information and result code".

You should end the test purpose if every part of the test purpose has succeeded by calling `Exit_OK()`. This should only be done once, because it is the means of passing back to the library function `testfunc()` the fact that the test purpose passed. If at an earlier part of the test purpose an error occurs and it is desired to exit, call `Exit()`.

6.2.3.3 Creating test purposes to test X Protocol extensions

The nature of the extension mechanism in X makes it difficult to just add support in the switch statements throughout the X Protocol library to support protocol extensions.

The reason for this is that you do not know the value of the event types and reply types until you have queried the X server.

For this reason, you are recommended to review the scope of the work that would be required in modifying the supplied X Protocol library before attempting to test X Protocol extensions. You can use the supplied X Protocol library as a framework, and develop new versions of routines which handle events and replies.

7. *Source file libraries*

This sections lists the contents of the principal libraries of source files used by many tests in the X Test Suite.

7.1 *The X Test Suite library*

A library of common subroutines for the X Test Suite has source in `$TET_ROOT/xtest/src/lib`. This is built automatically when building tests in the X Test Suite. Should it be required to build it separately for any reason run the command.

```
cd $TET_ROOT/xtest/src/lib
pmake install
```

The list of source files in this library, with a brief description of the contents of each file, is as follows:

XTestExt.c

If XTESTEXTENSION is defined, this file contains routines to access the XTEST extension in order to simulate device events and obtain information on the cursor attributes of windows.

If XTESTEXTENSION is not defined, dummy routines are used instead.

If XTESTEXTENSION is not defined, client-side functions previously in file XTestLib.c (now available in the XTEST extension library) are still included. These are XTestDiscard() (to discard current request in request buffer) and XTestSetGContextOfGC() and XTestSetVisualIDOfVisual() (to set values in opaque Xlib data structures). These functions require access to data structures now in the internal Xlib header file `Xlibint.h`.

badcmap.c

Create an invalid colourmap ID by creating a readonly colourmap of the default visual type.

badfont.c

Return a bad font ID by loading a font and then unloading it.

badgc.c

Return a bad GC id on display disp by creating a GC and invalidating it using the XTEST extension library function XTestSetGContextOfGC.

badpixm.c

Return a bad pixmap id on display disp by creating a pixmap and freeing it.

badvis.c

Make a visual bad by using the XTEST extension library function XTestSetVisualIDOfVisual.

badwin.c

Return a bad window id on display disp by creating a window and destroying it.

bitcount.c

Handle bits in words.

block.c
: Check whether process blocks when testing event handling functions.

buildtree.c
: Build a tree of windows specified by a list which determines position, size and parentage of each window.

checkarea.c
: Check pixels inside and/or outside an area of a drawable are set to given values.

checkevent.c
: Check two arbitrary events to see if they match, report an error if they don't.

checkfont.c
: Check returned font characteristics, properties, text extents and widths against those expected in the supplied test fonts.

checkgc.c
: Check GC components against expected values.

checkimg.c
: Check pixels inside and/or outside an area of an image are set to given values.

checkpixel.c
: Check specified pixels of a drawable are set to given values.

checktile.c
: Check that an area of a drawable is filled with a specified tile.

config.c
: Initialise the config structure by getting all the execution parameters.

crechild.c
: Create a mapped child window for a parent window, and wait for the child window to become viewable.

cursor.c
: Routines for accessing cursor information. This includes convenience functions for checking the cursor defined for a given window. These routines call those in XTestExt.c to use the XTEST extension to access the cursor information.

delete.c
: Set the test result code for the current test purpose to UNRESOLVED.

devcntl.c
: Routines for input device control. This includes convenience functions for pressing keys and buttons and remembering those pressed. These routines call those in XTestExt.c to use the XTEST extension to simulate the required device events.

dset.c
: Set every pixel in a drawable to a specified value.

dumpimage.c

Dump the contents of an image to a file.

environ.c

Contains a test suite specific version of putenv() (which may not be available on POSIX.1 systems). This is required to set up the environment before some calls to tet_exec().

err.c

Test error handler (installed when calling the function under test). Unexpected error handler (installed at all other times). I/O error handler (installed at all times). Obtain the error code and resource ID saved by the test error handler.

events.c

Handle the serial fields of incoming requests.

ex_startup.c

Generic startup routines required before executing the first test purpose and after executing the last test purpose. The routines `exec_startup()` and `exec_cleanup()` in this file should be called at the start and end of the `main()` function of each program executed via the TET function `tet_exec()`.

exposechk.c

Check that either enough expose events were received to restore the window, or that the window has been restored from backing store.

extenavail.c

If XTESTEXTENSION is defined, the function IsExtTestAvailable() returns True if the server extension XTEST is available, otherwise it returns False.

If XTESTEXTENSION is not defined, the function IsExtTestAvailable() always returns False.

gcflush.c

Flush the GC cache.

gcinclude.c

Functions which are called from the code included to test the correctness of use of GC components by the drawing functions.

The only function included at present is `setfuncpixel()`, which finds the first pixel set in a drawable (this will vary depending on the drawing function).

getevent.c

Check if there are events on the queue and if so return the first one.

getsize.c

Get the size of a drawable. Just uses XGetGeometry but avoids all the other information that you get with that.

gettime.c

Get the current server time. Use a property attached to the root window of the display called XT_TIMESTAMP and replace it with 42 (32-bits). The

PropertyNotify event that is generated supplies the time stamp returned.

iponlywin.c

Create an input only window.

issuppvis.c

The function `issuppvis()` takes a visual class as argument and returns true if such a class is supported by the server under test. This function uses the XGetVisualInfo() function rather than the user-supplied XT_VISUAL_CLASSES parameter.

The function `visualsupported()` takes a mask indicating a set of visuals, and returns a mask indicating the subset that is supported. If the mask is 0L then the mask shows all supported visuals.

The function `resetsupvis()` takes a mask indicating a set of visuals. Subsequent calls to `nextsupvis()` will return the next supported visual specified in the mask and increment a counter. The function `nsupvis()` returns this counter.

linkstart.c

Define global variables used by the TET which are required when linking test programs to produce a space-saving executable.

When the space-saving executable is executed, the TET initialisation code in the library function `linkstart.c` determines which test set is required. This is done by matching `argv[0]` with the `name` elements in the array of `linkinfo` structures. The corresponding test functions specified by the `testlist` element of the `linkinfo` structure are then executed.

lookupname.c

Convert symbolic values from X Window System header files to appropriate names.

makecolmap.c

Make a colourmap for the screen associated with the default root window.

makecur.c

Create a cursor that can be used within the test suite. The cursor is created using XCreateFontCursor. The shape chosen can be controlled through the configuration variable XT_FONTCURSOR_GOOD.

makegc.c

Make a GC suitable for use with the given drawable.

makeimg.c

Creates a general purpose image that can be used within the test suite. The image is cleared to W_BG.

makepixm.c

Creates a general purpose pixmap that can be used within the test suite. The pixmap is cleared to W_BG.

makeregion.c
: Creates a general purpose region that can be used within the test suite.

makewin.c
: Creates a general purpose windows that can be used within the test suite.

makewin2.c
: Creates windows corresponding to a particular area.

maxsize.c
: Obtain the number of cells in a colourmap.

nextvclass.c
: Functions to cycle through all the visual classes that are supposed to supported by the display/screen that is being tested. Note that these functions are only used in the tests for XMatchVisualInfo and XGetVisualInfo.

  The function `initvclass()` initialises the visual class table. The visual classes that are supported are supplied by the test suite user in the variable XT_VISUAL_CLASSES, together with the depths at which they are supported.

  The function `resetvclass()` resets the visual class table. Subsequent calls to `nextvclass()` obtain the next visual class and depth. The function `nvclass()` returns the size of the visual class table.

nextvinf.c
: Functions to cycle through all the visuals that are supported on the screen under test. These functions use the XGetVisualInfo() function rather than the user-supplied XT_VISUAL_CLASSES parameter. If the parameter XT_DEBUG_VISUAL_IDS is set to a non-empty string, only the visuals ID's in the string are used.

  The function `resetvinf()` obtains a list of all visuals supported for a particular screen. Subsequent calls to `nextvinf()` obtain the next visual. The function `nvinf()` returns the number of visuals.

nondpth1pix.c
: Obtain a pixmap of depth other than 1 if such a pixmap is supported.

notmember.c
: Returns a list of numbers that are not members of given list. (This is used to test assertions of the form "When an argument is other than X or Y, then a BadValue error occurs".

opendisp.c
: Open a connection to the server under test.

openfonts.c
: Open the xtest fonts, and place their ID's in the fonts array.

pattern.c
: Draw a pattern consisting of vertical bands on the specified drawable.

pfcount.c

Functions which may take arguments which are set to the pass and fail counters in test set code created by `mc`. Calls to the `pfcount` functions are inserted in order to use the counters at least once, and so prevent `lint` reporting unwanted errors.

pointer.c

Routines to move the pointer, and determine if the pointer has been moved.

regid.c

Routines are provided to register resources created during a test purpose. Wherever possible, library functions register resources, and test purposes may do so directly if desired. Registered resources are then destroyed at the end of the test purpose.

report.c

Reporting functions, which output test information messages to the TET journal file. These all use the TET reporting function `tet_infoline()`.

rpt.c

Reporting functions, which output test information messages to the TET journal file, and additionally assign a test result code. These all use the TET reporting function `tet_infoline()`.

savimage.c

The function `savimage()` returns a pointer to a saved image on a drawable using `XGetImage`.

The function `compsavimage()` checks that the image currently on the drawable matches a saved image.

The function `diffsavimage()` checks that the image currently on the drawable differs from a saved image.

These functions are used where the precise pixels drawn cannot be determined in advance, but the test result may still be infered by image comparisons.

setline.c

Convenience functions to set line width, cap style, line style and join style in a GC, using `XChangeGC()`.

settimeout.c

The function `settimeout()` sets a timeout which causes the process to exit after a timeout. This should be done only in a child process of a test purpose created by `tet_fork()`.

The function `cleartimeout()` clears a previously set timeout.

stackorder.c

The function `stackorder()` uses `XQueryTree()` to determine the position of a window in the stacking order.

startcall.c

The function `startcall()` checks for any outstanding unexpected X protocol errors, which might have been generated, for example, during the setup part of the test. A call to `XSync()` is made to achieve this.

The function `startcall()` installs a test error handler in place of the unexpected X protocol error handler.

The function `endcall()` checks for any X protocol errors caused by the function under test. A call to `XSync()` is made to achieve this.

The function `endcall()` installs the unexpected X protocol error handler.

startup.c

Generic startup routines called by TET before executing the first test purpose and after executing the last test purpose.

tpstartup.c

Generic startup routines called by TET before executing each test purpose and after executing each test purpose.

verimage.c

The function `verifyimage()` uses `XGetImage()` to obtain the contents of the specified drawable. This is then compared with the contents of a "known good image file". If there is a discrepancy, the image produced by the server is dumped to a file using `dumpimage()` together with the known good image. The image produced by the server and the known good image may be examined as described in the section in the "User Guide" entitled "Examining image files".

If the execution configuration parameter XT_DEBUG_NO_PIXCHECK is set to `Yes`, the image checking is skipped in `verifyimage()`.

If the execution configuration parameter XT_SAVE_SERVER_IMAGE is set to `Yes`, the image produced by the server is dumped to a file using `dumpimage()` (regardless of whether it matches the "known good image file").

For more background on pixmap verification see the earlier section entitled "Creating test purposes which use pixmap verification".

winh.c

Build a tree of windows to test event generation, propogation and delivery.

xthost.c

Specifies operating system dependent data used by the access control list functions. This includes arrays of `XHostAddress` structures. These should be checked and if necessary edited refering to the section in the "User Guide" entitled "System dependent source files".

xtestlib.h

This file contains definitions which are common to many of the source files in the X Test Suite library, and it is included in those source files.

xtlibproto.h

This file contains declarations and (if required by an ANSI Standard-C compiler) function prototypes for all the functions in the source files in the X Test Suite library.

7.2 The X Protocol library

A library of common subroutines for the X Protocol tests in the X Test Suite has source in `$TET_ROOT/xtest/src/libproto`. This is built automatically when building tests in the X Test Suite. Should it be required to build it separately for any reason run the command.

```
cd $TET_ROOT/xtest/src/libproto
pmake install
```

The list of source files in this library, with a brief description of the contents of each file, is as follows:

ClientMng.c

Having established a client connection to the X server using the functions in ConnectMng.c, allocate a client data structure and fill in its display structure.

ConnectMng.c

Establish a client connection to the X server.

DataMove.c

Convert individual fields into format for sending to the X server.

DfltVals.c

Obtain reasonable default values for contents of request structures.

Expect.c

Check for the expected response (error, event, reply, or nothing) from the X server.

JustALink.c

This file is a link to one of the files XlibXtst.c, XlibOpaque.c, or XlibNoXtst.c. The link is created when the X Protocol library is built, and the file used depends on the configuration parameter XP_OPEN_DIS.

Log.c

Log test results.

MakeReq.c

Construct a request structure using the functions in DfltVals.c, which has reasonable default values so that it may be immediately sent to the X server using the functions in SendReq.c.

RcvErr.c

RcvEvt.c

RcvRep.c

Unpack the response from the server into a structure (RcvErr.c for errors, RcvEvt.c for events, RcvRep.c for replies; these all use DataMove.c to do the unpacking).

ResMng.c
: Create a resource (e.g. atom, window) and store its resource ID in the client data structure.

SendEvt.c
: Pack an event structure into a request structure (only used by SendEvent protocol request).

SendReq.c
: Pack a request structure in correct format using the functions in DataMove.c and send to the X server.

SendSup.c
: Support routines for packing request structure.

ShowErr.c

ShowEvt.c

ShowRep.c

ShowReq.c

ShowSup.c
: Display contents of structures in nice human-readable form (ShowErr.c for errors, ShowEvt.c for events, ShowRep.c for replies, and ShowReq.c for requests, all of which call ShowSup.c support routines).

TestMng.c
: Manage the setup and closedown of the tests. This file includes definitions and initialisation of global variables (including TET configuration variables) and assigning test result codes.

TestSup.c
: Support routines for handling mapping state and event masks of windows.

Timer.c
: Set up a timer that will execute a certain routine on completion.

Utils.c
: Utilities for isolating operating system dependencies.

Validate.c
: Routines to check whether the server under test supports the feature being tested (eg. writable colour cells).

ValListMng.c
: Modify the value lists at the ends of request structures.

XlibNoXtst.c
: This file contains functions which emulate the post R5 enhanced connection setup scheme. A connection can be established in client native or byte-swapped orientations, and (when testing XOpenDisplay) both valid and invalid byte orderings may be sent to the X server. The connection is made using operating system specific procedures which were developed in 4.2BSD environment, and may need modifications to work on other systems.

XlibOpaque.c

This file contains portable functions to handle connection setup where the Xlib implementation does not support the post R5 enhanced connection setup scheme. The Xlib functions XOpenDisplay and ConnectionNumber are called here to obtain a connection using the client native byte orientation, and subsequent X Protocol requests are made using this connection.

XlibXtst.c

This file contains portable functions to handle connection setup where the Xlib implementation supports the post R5 enhanced connection setup scheme. The enhancement involves using additional parameters to the Xlib function _XConnectDisplay() which allow a byte swapped connection to be established. Details of operating system specific connection setup procedures including networking are thus not needed in the X Protocol library.

XstIO.c

Routines to handle protocol packet transmission and reception including fatal I/O errors.

delete.c

Set the test result code for the current test purpose to UNRESOLVED.

linkstart.c

Define global variables used by the TET which are required when linking test programs to produce a space-saving executable.

When the space-saving executable is executed, the TET initialisation code in the library function `linkstart.c` determines which test set is required. This is done by matching `argv[0]` with the `name` elements in the array of `linkinfo` structures. The corresponding test functions specified by the `testlist` element of the `linkinfo` structure are then executed.

startup.c

Generic startup routines called by TET before executing the first test purpose.

tpstartup.c

Generic startup routines called by TET before executing each test purpose.

DataMove.h

This file contains the macros for byte swapping and word alignment.

XstlibInt.h

This file contains definitions which are common to many of the source files in the X Protocol library, and it is included in those source files.

XstosInt.h

This file contains definitions related to operating system functions which are common to many of the source files in the X Protocol library, and it is included in those source files.

7.3 The X test fonts library

A library of common subroutines defining the characteristics of the test fonts for the X Test Suite has source in `$TET_ROOT/xtest/fonts`. This is built automatically when building tests in the X Test Suite. Should it be required to build it separately for any reason run the command.

```
cd $TET_ROOT/xtest/fonts
pmake install
```

The source files `xtfont0.c` to `xtfont6.c` contain definitions of `XFontStruct` structures named `xtfont0` to `xtfont6` which define the characteristics of the test fonts used by many of the text drawing functions.

8. Appendix A - reason codes for extended assertions

The reason code is a number between 1 and 6 (currently) and is used if and only if the category is B or D. This number corresponds to a reason from the following table which is coded into `mc`.

The text of the reason will be printed with a result code UNTESTED if there is no >>CODE.

1 - "There is no known portable test method for this assertion",
2 - "The statement in the X11 specification is not specific enough to write a test",
3 - "There is no known reliable test method for this assertion",
4 - "Testing the assertion would require setup procedures that involve an unreasonable amount of effort by the user of the test suite.",
5 - "Testing the assertion would require an unreasonable amount of time or resources on most systems",
6 - "Creating a test would require an unreasonable amount of test development time."

9. *Appendix B - commands for fonts and symbols in assertions*

In the text of assertions there should be no in-line nroff font changes. This is because the font names may need to be changed on some systems.

As an alternative, a number of macros have been defined which are understood by the utilities developed during stage two of the project. The definition of these macros uses appropriate fonts to correspond closely with those used by the X Window System documentation.

1. Arguments to a function should be written:

   .A window

2. Function names should be written:

   .F XAllocColorCells

   (When the special symbol xname is used it can be left as it is, so the .F form only needs using when refering to some other function. We have avoided cross references to other functions where possible).

3. Structure members should be written:

   .M override_redirect

4. Symbols should be written:

   .S InputOutput

   This is used for everything that is in the courier font in the X Window System documentation and which is not a function name or structure member. This includes the #define constants in the headers and typedef'ed names.

```
Eg.
      BadColor
      IsViewable
      DirectColor
      Visual
      Display
      MotionNotifyEvent
```

Punctuation separated by white space from the argument will be in the original font, as in mm.

.A InputOutput ,
.A InputOnly .

— There is a .SM macro, as in mm. Any word that is uppercase only should use it to obtain a reduced point size.

  .SM DEBUG
  .SM MIT

10. *Appendix C - Included error assertions*

The .ER keyword is described in the section entitled "Included errors - .ER".

This appendix gives the names the files which are included when this keyword is used with the supported arguments, and shows the text of the assertions in those files.

All the files from which included tests are stored are located in the directory `$TET_ROOT/xtest/lib/error`.

The names of the files which are included, and the text of the assertion contained in the file, are specified in the following list:

Access grab
File included: EAcc1.mc
Assertion text:
```
When an attempt to grab  a  key/button  combination  already
grabbed  by  another  client is made, then a BadAccess error
occurs.
```

Access colormap-free
File included: EAcc2.mc
Assertion text:
```
When an attempt to free a colormap entry  not  allocated  by
the client is made, then a BadAccess error occurs.
```

Access colormap-store
File included: EAcc3.mc
Assertion text:
```
When an attempt to store into a read-only or an  unallocated
colormap entry is made, then a BadAccess error occurs.
```

Access acl
File included: EAcc4.mc
Assertion text:
```
When an attempt is made to modify the  access  control  list
from  a  client that is not authorised in a server-dependent
way to do so, then a BadAccess error occurs.
```

Access select
File included: EAcc5.mc
Assertion text:
```
When an attempt to select an event type is  made,  which  at
most  one  client can select, and another client has already
selected it then a BadAccess error occurs.
```

Alloc
File included: EAll.mc
Assertion text:
```
When the server fails to allocate a required resource,  then
a BadAlloc error occurs.
```

Atom [ARG1] [ARG2] ...
File included: EAto.mc
Assertion text:

```
When an atom argument does not name a valid  Atom [, ARG1]  [or
ARG2], then a BadAtom error occurs.
```

Color

File included: ECol.mc

Assertion text:

```
When a colourmap argument does not name a  valid  colourmap,
then a BadColor error occurs.
```

Cursor [ARG1] [ARG2] ...

File included: ECur.mc

Assertion text:

```
When a cursor argument does not name a valid Cursor [, ARG1] [or
ARG2], then a BadCursor error occurs.
```

Drawable [ARG1] [ARG2] ...

File included: EDra.mc

Assertion text:

```
When a drawable argument does not  name  a  valid  Drawable,
[ARG1] [or ARG2], then a BadDrawable error occurs.
```

Font bad-font

File included: EFon1.mc

Assertion text:

```
When a font argument does not name  a  valid  font,  then  a
BadFont error occurs.
```

Font bad-fontable

File included: EFon2.mc

Assertion text:

```
When the font argument does not name  a  valid  GContext  or
font resource, then a BadFont error occurs.
```

GC

File included: EGC.mc

Assertion text:

```
When the GC argument does not name  a  defined  GC,  then  a
BadGC error occurs.
```

Match inputonly

File included: EMat1.mc

Assertion text:

```
When a drawable argument  is  an  InputOnly  window  then  a
BadMatch error occurs.
```

Match gc-drawable-depth

File included: EMat2.mc

Assertion text:

```
When the graphics context and the drawable do not  have  the
same depth, then a BadMatch error occurs.
```

Match gc-drawable-screen

File included: EMat3.mc

Assertion text:

```
When the graphics context and the drawable were not  created
for the same root, then a BadMatch error occurs.
```

Match wininputonly

File included: EMat4.mc

Assertion text:

```
When the window argument  is  an  InputOnly  window  then  a
BadMatch error occurs.
```

Name font

File included: ENam1.mc

Assertion text:

```
When the specified font does not exist, then a BadName error
occurs.
```

Name colour

File included: ENam2.mc

Assertion text:

```
When the specified colour does not  exist,  then  a  BadName
error occurs.
```

Pixmap [ARG1] [ARG2] ...

File included: EPix.mc

Assertion text:

```
When a pixmap argument does not name a valid Pixmap [, ARG1] [or
ARG2], then a BadPixmap error occurs.
```

Value ARG1 VAL1 [VAL2] ...

File included: EVal.mc †

Assertion text:

```
When the value of ARG1 is other than VAL1 [or  VAL2],  then  a
BadValue error occurs.
```

† - the assertion text is not in the included file, but is inserted directly by `mc`.

Value ARG1 mask VAL1 [VAL2] ...

File included: EVal.mc †

Assertion text:

```
When the value of ARG1 is not a bitwise combination of  VAL1
[or VAL2], then a BadValue error occurs.
```

† - the assertion text is not in the included file, but is inserted directly by `mc`.

Window [ARG1] [ARG2] ...

File included: EWin.mc

Assertion text:

```
When a window argument does not name a valid Window [, ARG1] [or
ARG2], then a BadWindow error occurs.
```

11. Appendix D - `mc` *utility*

Usage

```
mc [-a a_list] [-o <output-file>] [-l] [-m] [-s] [-p] [<input-file>]
```

The `mc` utility outputs a C source file containing tests specified in the input file `<input-file>`, which must be a dot-m file which has the format specified in the section entitled "Source file syntax".

If no `<input-file>` is specified, the input is taken from standard input. Multiple input files can be processed by the utility, but the overall syntax must still conform to that defined in the section entitled "Source file syntax". A consequence of this is that you cannot specify another title section for a different function and expect to output tests for more than one function simultaneously. Limited diagnostics are given if the file does not have the required syntax. By default, the C source file is written to the standard output stream.

More details of the formats of the C source files produced by `mc` are given in the section entitled "Source file formats".

Options

`-a a_list`
: This permits the specification of a list of assertions of the form `n1-m1,n2-m2,...` to be output. Test code will only be output corresponding to the tests in the specified ranges.

`-o output-file`
: This sends the output to the file `<output-file>` instead of the standard output stream.

`-l`
: This option outputs a C source file containing tests with modified startup code which allows the source code to be compiled and linked into a space-saving executable file. The format of these files is described in the section entitled "C files for linked executable - `link.c`".

`-m`
: This option outputs a C source file containing tests for the macro version of the function specified in the title section of the dot-m file. The format of these files is described in the section entitled "C files for standalone executable in macro tests - `MTest.c`".

: The macro name is set to the `<macroname>` argument of the >>SET macro option - if there is no >>SET macro option in the file, or no argument specified, the default is the `function` argument in the >>TITLE keyword, with the leading letter ‘X’ removed.

`-s`
: This option outputs a test strategy from the dot-m file as a C source code comment block between the assertion and the code. The test strategy is derived from the corresponding strategy section in the dot-m file.

`-p`

This causes additional output including indicators of line number in the original dot-m file (where possible). This means that any diagnostics produced by *cc(1)* or *lint(1)* will refer to the line numbers in the original dot-m file rather than the C source file.

12. Appendix E - `mmkf` *utility*

Usage

```
mmkf [-o <output-file>] [-s sections] [<input_file>]
```

The `mmkf` utility outputs a Makefile corresponding to the specified input file `<input-file>`, which must be a dot-m file which has the format specified in the section entitled "Source file syntax". The Makefile can build all the C source files that can be output by `mc` from the input file `<input-file>`.

If no `<input-file>` is specified, the input is taken from standard input. Multiple input files can be processed by the utility, but the overall syntax must still conform to that defined in the section entitled "Source file syntax". A consequence of this is that you cannot specify another title section for a different function and expect to output Makefiles for more than one function simultaneously. Limited diagnostics are given if the file does not have the required syntax. By default, the Makefile is written to the standard output stream.

More details of the formats of the Makefiles produced by `mmkf` are given in the sub-section entitled "Makefile" in the section entitled "Source file formats".

Options

`-o output-file`

This sends the output to the file `<output-file>` instead of the standard output stream.

`-s sections`

This option enables output of certain optional sections of the Makefile. By default, output of all these sections is enabled. There is no reason why you should need to use this option with the current version of the X Test Suite.

The `sections` argument is a character string which may contain the key letters `l`, `L`, `m` and `p`. If these characters are included, the specified sections of the Makefile are then output.

Key letter	Optional section
l	Targets for linked executable
L	Targets for libraries
m	Targets for linting and cleaning
p	Targets for building known good image files

13. Appendix F - `ma` *utility*

Usage

```
ma [-a a_list] [-o <output-file>] [-h] [-s] [-p] [-m] [<input-file>]
```

The `ma` utility outputs a file containing a list of assertions in *nroff(1)* format (requiring no macros other than those supplied in file `maheader.mc`). The assertions are specified in the input file `<input-file>`, which must be a dot-m file which has the format specified in the section entitled "Source file syntax".

If no `<input-file>` is specified, the input is taken from standard input. Multiple input files can be processed by the utility, but the overall syntax must still conform to that defined in the section entitled "Source file syntax". A consequence of this is that you cannot specify another title section for a different function and expect to output assertions for more than one function simultaneously. Limited diagnostics are given if the file does not have the required syntax. By default, the assertion list is written to the standard output stream.

More details of the format of the assertion list produced by `ma` are given in the subsection entitled "Formatting assertions" in the section entitled "Source file formats".

Options

`-a a_list`
: This permits the specification of a list of assertions of the form `n1-m1,n2-m2,...` to be output. Assertions will only be output corresponding to the tests in the specified ranges.

`-o output-file`
: This sends the output to the file `<output-file>` instead of the standard output stream.

`-h`
: The macros required for formatting the assertions are included at the start of the output stream. These are copied from the file `maheader.mc`.

: By default, the macros are not copied to the output stream.

`-s`
: If this option is specified, and the `-h` option is specified, the line

```
.so head.t
```

: will be output at the start of the output stream.

: This option is not intended for general use - it was used when distributing assertions in compact form for external review.

`-p`
: The macros `.NS` and `.NE` will be output before and after each line in the dot-m file which is a comment (commencing with >>#). By default, dot-m file comments are not output. The macros `.NS` and `.NE` are defined in `maheader.mc`; they cause the dot-m file comment lines to be printed in italic font by *nroff(1)*.

: This option is not intended for general use - it was used when reviewing assertions before delivery.

`−m`

This option outputs assertions for the macro version of the function specified in the title section of the dot-m file.

The macro name is set to the `<macroname>` argument of the >>SET macro option - if there is no >>SET macro option in the file, or no argument specified, the default is the `function` argument in the >>TITLE keyword, with the leading letter ‘X’ removed.

CONTENTS