MIN Test Framework
User's Guide

Change history:

Version	Date	Status	Comments
1.0	17.12.2008	Draft	Initial draft
1.1	20.01.2009	Draft	Added chapter 17
1.2	09.02.2009	Draft	Added information about scripter internal variables and keywords: if, else, endif and breaklloop.

Table of contents:

1. Document control

1.1 Documentation conventions

Code is written with the `Courier New` font.

1.2 Abbreviations

DLL	Dynamic Link Library
CUI	Console User Interface

1.3 Definitions

IPC	Inter-Process Communication, a set of techniques for the exchange of data among two or more processes.
MIN Console UI	The Console application of MIN Test Framework.
MIN Logger	A utility of MIN Test Framework that offers logging services.
MIN Parser	A utility, which extracts information from text files.
MIN, MIN Test Framework	The name of the developed MIN Test Framework test tool.
Test Case	A unique test, which either passes or fails.
Test Engine	A module of MIN Test Framework.
Test module	Contains test cases. (A test module type can be, for example, hard-coded.)
Test Module Template Wizard	A tool of MIN for automatically creating a test module template (e.g. MINUnit.)
TMC	Test Module Controller
Test Set	A collection of test cases, currently this is implemented in MIN Console UI.

2. Introduction

This document describes how to use MIN Test Framework (MIN).

Chapter 3 describes what MIN is and how its program modules are placed in the MIN architecture.

Chapter 4 describes how to set up, build and configure MIN.

Chapter 5 describes how to use MIN for executing test cases.

Chapter 6 describes how to use MIN for test case implementation.

Chapter 7 gives instructions on how to run MIN tests as part of a build.

Chapter 8 describes usage of MIN Parser.

Chapters 9 and 10 deal with MIN Scripter.

Chapter 11 shows how to use Lua to create MIN test cases.

Chapter 12 deals with Python test cases.

Chapter 13 describes usage of MIN Logger.

Chapter 14 describes usage of Event System.

3. Introduction to MIN Test Framework

This chapter describes what the MIN Test Framework (MIN) is, and what are the program modules that MIN consist of.

MIN is a test harness for testing Linux non-UI components. This test framework can be used for both test case implementation and test cases execution.

MIN separates the actual test cases from the test case execution environment. It provides different user interfaces and a common reporting mechanism, and allows executing test cases simultaneously. This test framework contains user interfaces on the interface layer, MIN Engine on the engine layer and the test modules and tools on the test layer.

The console UI or an interface to a test automation tool can be used to start and monitor test case execution.

Test Engine is responsible for loading test modules and for executing the test cases.

Test cases are collected to separate modules, known as test modules, which are easy to implement.

MIN is not an automatic test case creator but allows you to concentrate on the actual test case implementation.

3.1 MIN architecture overview

Figure 3-1 shows an overview of the MIN architecture. The architecture is divided into three main layers: the interface layer, the engine layer and the test layer. The arrows show the direction of the data flow.

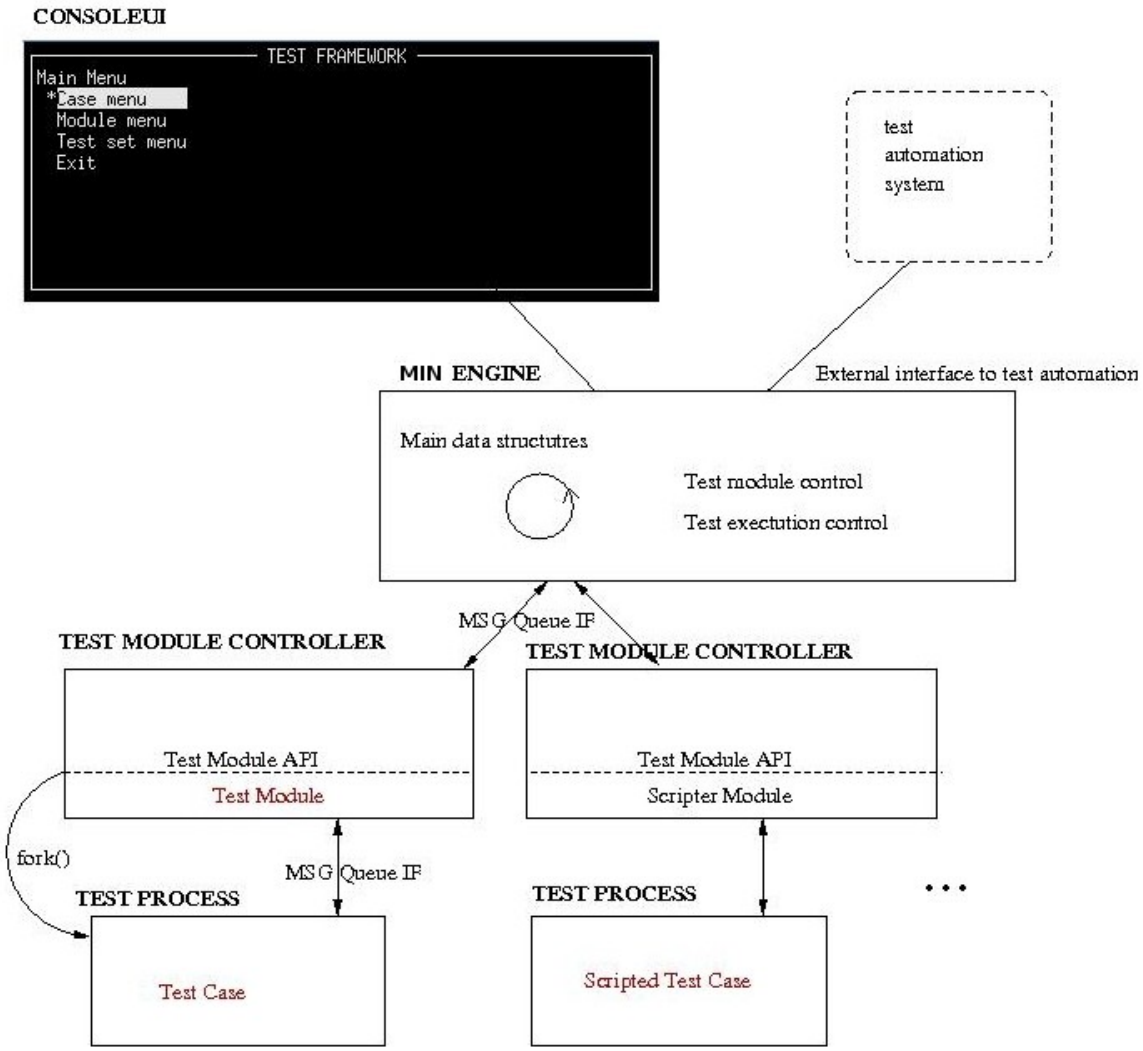

Figure 3-1 MIN architecture

The architecture of MIN is modular. These main modules are listed in Table 3-1. The interface between the modules is either a function call or IPC interface. There are no cyclic dependencies between the modules. The user interface module uses MIN Engine, MIN Engine uses Test Module Controller(s) and Test Module Controller uses test module.

Table 3-1 Modules

Module	Type	Multiplicity	Implementation
User Interface	Executable	One	Executable
EXT interface	Executable	One	Executable
Engine	Library	One	Static linked library
Test Module Controller	Library	Several instances	Executable
Test Modules	Library	Several separate libraries and several instances	Library (.so)

The MIN modules on the interface layer are described in Section 3.2. The MIN Engine module on the engine layer is described in Section 3.3 and the MIN modules on the test layer are described in Section 3.4.

3.2 MIN modules on the interface layer

Currently, three interfaces are supported in MIN:

- MIN Console UI
- MIN Command line interface (limited)
- MIN EXT interface

The UI is used to start the test case execution. The UI can also show information about the test case progress to the user.

The interface modules have access to the engine functionality. There exists several asynchronous functions in Test Engine, and therefore the user interface has to handle several ongoing asynchronous requests.

The MIN Console UI is used to manually start the test case execution.

The command line interface can be used, for example, to execute all the test cases of all configured test module(s).

3.2.1 MIN Console UI

MIN Console UI is a menu-based console application, which can be used to manually execute test cases. A module under development can be tested on target before the actual UI is usable.
A screen shot of MIN Console UI can be seen in Figure 3-2.

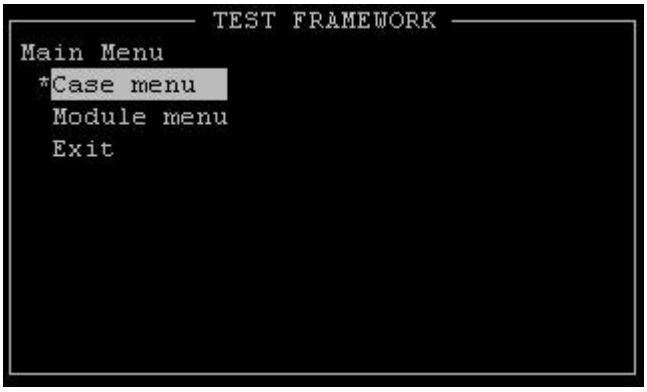

Figure 3-2 MIN Console UI screen shot

3.2.2 MIN EXT interface

The MIN EXT interface is used to start test case execution from for example, a test automation system.

3.3 MIN module on the engine layer

Test Engine is the main module of MIN. It is responsible for loading test modules and for executing the test cases.

3.3.1 Test Engine

Test Engine provides an API for user (console UI or EXT interface) for executing test cases and loading the test modules. Test Engine can load several test modules and execute different test cases from separate test modules at the same time. Engine provides user (CUI or EXT – in the following chapter, both will be referred to as User) with functionalities for control of test execution and test result collection.

Engine and User interface run in same process, divided into threads servicing user's requests and commands, and communication with "test module" layer of the system. This process also spawns, by means of fork( ) system call, new processes for each needed Test Module Controller. Later on, it controls those processes and communicates with them, to facilitate test execution and result reception.

Main interface for communication between Engine and Test Module Controller processes is POSIX message queue. One thread in Engine is polling queue continuously for messages addressed to engine (from Test Module Controllers). Functions that send messages to queue run also in other Engine's thread, enabling control of test execution by User.

Communication with User is realized by function interface, since physically engine is statically linked library, compiled into one executable with User interface.

Test Engine uses the initialization file of MIN for its initial parameters. This initialization file also contains a preloading list of the test modules that are used to automatically load and initialize the test modules on the MIN startup.

Test Engine generates its own test report that can be used to analyze the test case progress after the test cases have been executed. Test Engine also creates a log file that contains detailed information about the operations that are performed in a MIN Engine.

Test Engine keeps a list of test modules, as well as a list of available test cases from the modules. There is also data for every run of each test case, to facilitate test results logging/display and report generation.

3.4 MIN modules on the test layer

The test layer consists of the following modules:

- Test Module Controller(s)
- Test module(s)
- Tools
    1. MIN Parser
    2. MIN Logger
    3. MIN Event System

Figure 3-3 shows how the MIN test layer modules are placed in the MIN architecture. Boxes with bolded font indicate the MIN modules on the test layer.

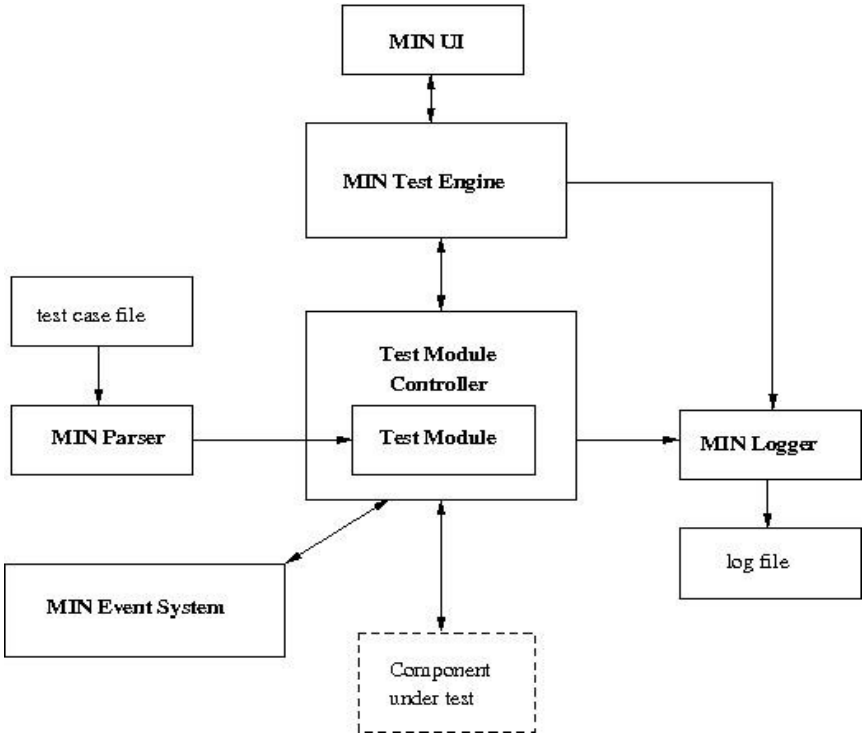

Figure 3-3 MIN modules on the test layer

3.4.1 Test Module controller

For each test case, there exists one Test Module Controller that gets requests from MIN Engine and either handles the requests itself or calls the corresponding function from the test module. For example, a request to execute a test case causes a function call to the test module and a request to stop the test case is handled inside Test Module Controller.

3.4.2 Test module

Test cases are collected to separate modules, known as test modules, which are easy to implement.

There can be several test cases implemented in one test module, and test cases can be defined for example, by hard coding test cases to the test module. In addition, MIN supports running different test cases simultaneously from several test modules.

A test module contains the actual test case implementation. The test module is a library, which implements the MIN Test Module API. The most important function in that interface is `int tm_run_test_case( unsigned int id, const char *cfg_file, TestCaseResult *result)`, which executes the test case and returns the test case result.

The MIN services interface from the test module to MIN provides synchronous operations for the test module. For example, printing from the test module is implemented inside a synchronous function that causes a synchronous request to be completed inside Test Module Controller.

Currently, three different test module types are supported in MIN:

- Hardcoded
- Normal
- MINUnit

MIN Test Module Template Wizard provides test module templates for test module creation. The possible test module template types of MIN are:

- Hardcoded (for a Hardcoded test module)
- Normal
- MINUnit

3.4.2.1 Hardcoded test module

There is a test module template for Hardcoded module (min/TestModuleTemplates/ HardCodedTestModuleXXX). It contains some ready made test cases for MIN try-out purposes. Figure 3-4 shows how Hardcoded test modules are situated in MIN. Boxes with bolded font indicate the MIN modules on the test layer. The Hardcoded test module is underlined in the figure.

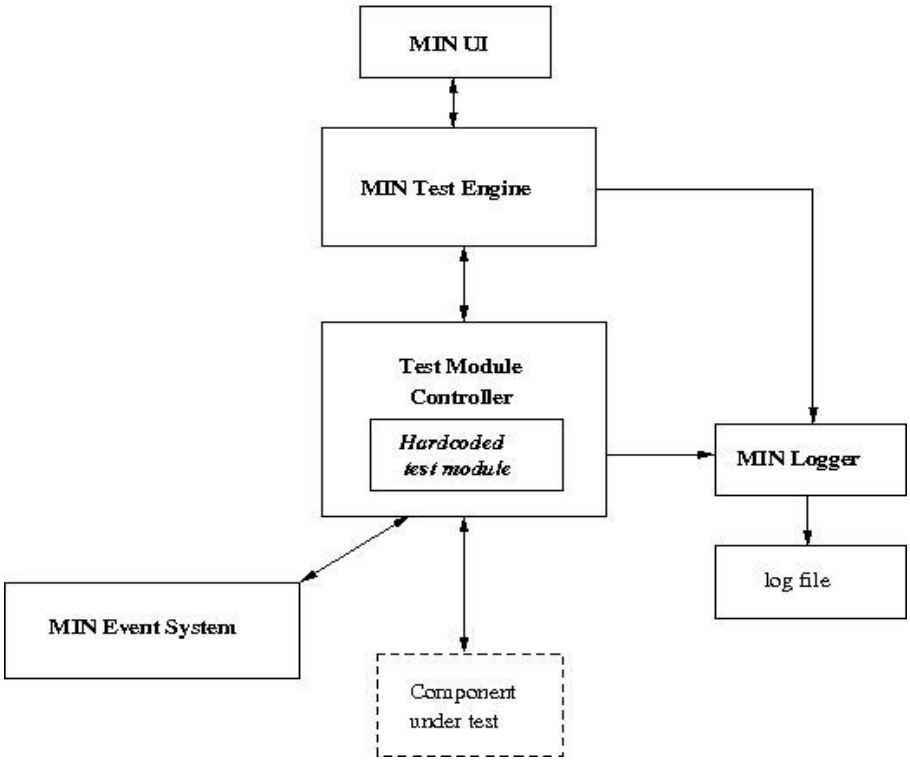

Figure 3-4 Hardcoded test module on the test layer

3.4.2.2 Normal test module

When using the normal test module template type, an empty test module is created. Test cases for a normal type of test module can be freely implemented, and this template enables the implementing of unique type of test cases and the use of old test cases as the basis. Figue 3-5 shows where the normal test module is situated in MIN. Boxes with bold font indicate the MIN modules on the test layer. The normal test module and its test case file are underlined in the figure.

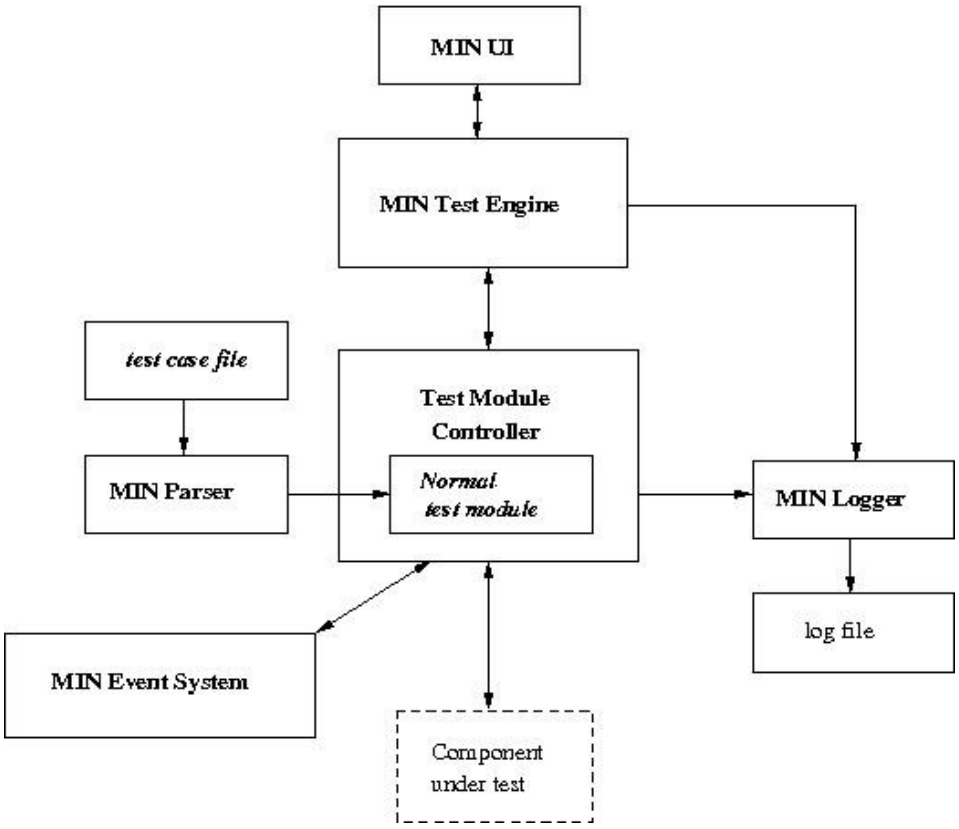

Figure 3-5 The Normal test module on the test layer

3.4.2.3 MINUnit test module

The MINUnit test module differs from Hardcoded test module mainly in the syntax of test case definition. MINUnit is an implementation of XUnit testing framework (see http://en.wikipedia.org/wiki/XUnit). The MINUnit test module supports assert macros as well as test fixtures setup and cleanup. See chapter 6.6 for instructions on how to create a test case for MINUnit test module.

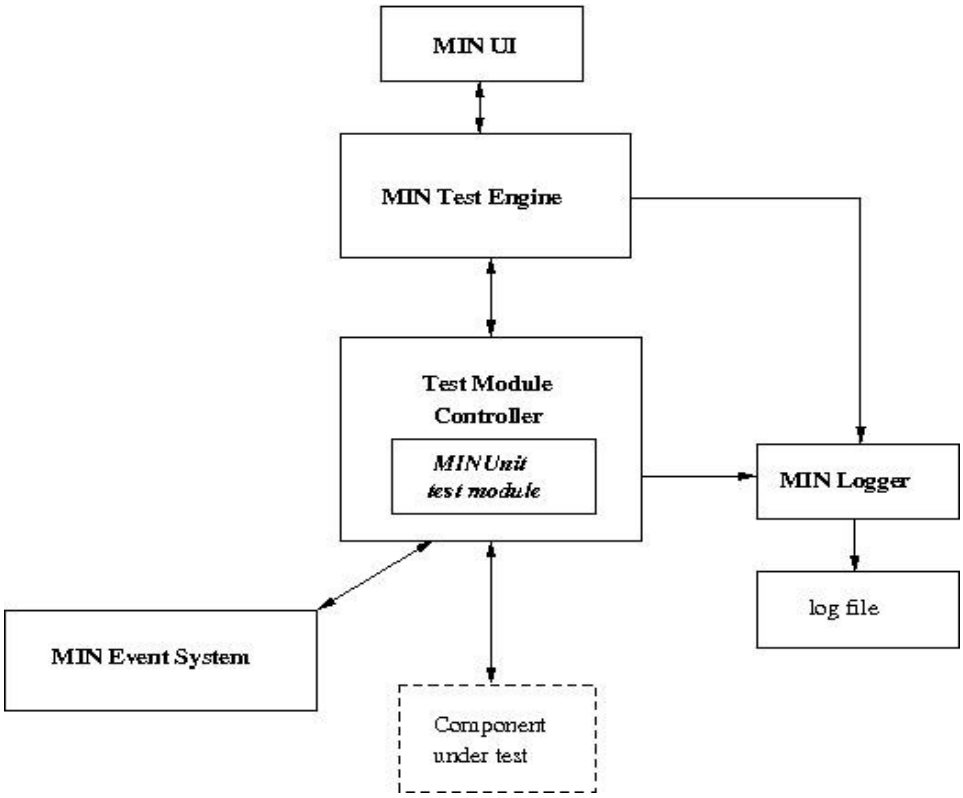

Figure 3-6 The MINUnit module on the test layer

3.4.3 MIN Parser

MIN Parser can be used when there is a need to parse a test module's test case files or a test module's initialization files. By providing a common parser in MIN, it can obtain better control of the test case behavior. For more information on MIN Parser, see Chapter 8.

3.4.4 MIN Logger

The purpose of MIN Logger is to get information from the modules in order to write different log files or to send information to another data store via, for example, Bluetooth. By offering a common logger in MIN, logging can be controlled.

Test modules can also use their own logging mechanisms, but it is preferred that the new test code use MIN Logger.

For more information about MIN Logger, see Chapter 13.

3.4.5 MIN Event System

MIN Event System enables synchronization between the test cases/modules through convenient interface. The MIN Event System has two types of events: state events and indication events. State events are used to indicate that some state is active / inactive. Test process can set and unset a state event. Indication events are used to indicate that some event happened; an indication event can only be set.

MIN Event System can be used to synchronize test case execution. The high-level part of MIN Event System is implemented inside Test Engine and the low-level part inside Test Module Controller.

For more information about MIN Event System, see Chapter 14.

3.4.6 MIN Text interface

MIN provides generic Text interface to manipulate strings without knowledge about the amount of memory needed to store particular string. Text interface will grow or shrink, depending on situation.

For more details check the MIN Text interface chapter.

3.5 MIN features

The following features are currently supported:

- Test module implements test cases
- Error and exception handling
- Concurrent test case execution
- Test Module Template Wizard (*createtestmodule*)
- File parsing with MIN Parser
- Logging facilities for test modules with MIN Logger (currently only limited set of features)
- Test case synchronization with MIN Event System
- Write once, test everywhere, that is, test cases made during the development phase can be used in system testing, automatic release testing, and so on
- Easy to use
- Multiple test cases can be executed concurrently
- All execution errors and exceptions are handled properly and reported to teste.
- Support for test automation

4. Setting up MIN

This chapter describes how MIN can be taken into use.

4.1 Building MIN

Unzip the MIN package.

$ unzip MIN<year>w<week>.zip

Go to the package newly created directory.

$ cd MIN<year>w<week>/min

To build MIN, type:

$ sh build.sh

To Install MIN, type:

$ sudo make install

After these steps you should have MIN and tmc binaries in the /usr/bin directory and a default min.conf in /etc.

To start MIN, type:

$ min (uses the default min.conf from /etc)

When starting MIN, the tool searches min.conf file from following locations:
1. Current working directory
2. /home/<user name>/.min
3. /etc

Use Test Module Wizard to generate new Test Module, as instructed in chapter 6.3. When building generated module, it is automatically added to ~/.min/min.conf file.

4.2 Creating MIN Debian packages

MIN release includes metadata for creating Debian packages. The min and min-dev packages can be created with dpkg-buildpackage in the release min/ directory.

$ cd tags/<YYYY>w<WW>

$ dpkg-buildpackage -rfakeroot

After running the command, min and min-dev packages should appear in the parent directory.

$ ls ../.deb*

../min_2008w50_i386.deb ../min-dev_2008w50_i386.deb

The MIN package install binaries (to /usr/bin), and a demo test library (to /usr/lib/min). The min-dev package contains Test Module Wizard and header and libraries needed to build MIN test modules.

After installing MIN package there should be min and tmc binaries in /usr/bin directory and a default min.conf in /etc/.

Test module Debian packaging is also supported. This feature is available for modules created using Test Module Wizard.

4.2.1 Hardware builds

The recommended way is to create a debian package in Scratchbox and install it to hardware.

Alternatively MIN must be compiled in Scratchbox and following binaries have to be copied into the hardware:

- min/src/main/min.bin to /usr/bin/
- min/src/main/min to /usr/bin/
- min/src/tmc/tmc to /usr/bin/
- min/conf/min.conf to /etc/
- Your_test_library.so to /home/<user name>/.min/

Note: The location of these files must match the paths in min.conf file.

4.3 Configuring MIN via the MIN Test Framework initialization file

MIN has an initialization file that is used to set the MIN functionality. Currently initialization file contains paths for searching test modules and configuration files, defaults for MIN logger and the used test modules.

An example of the initialization file is shown in Example 4.1.

Example 4.1 MIN Test Framework initialization file

```
[Engine_Defaults]
ModSearchPath=/usr/lib/min
ModSearchPath=$HOME/.min
[End_Defaults]
[New_Module]
ModuleName=/home/user/.min/exampleModule.so
TestCaseFile=/home/user/.min/Example.cfg
[End_Module]

[Logger_Defaults]
EmulatorBasePath=/tmp
EmulatorFormat=TXT
EmulatorOutput=FILE SYSLOG
ThreadIdToLogFile=NO
[End_Logger_Defaults]
```

Table 4-2 shows the possible settings in the *min.conf* file.

Table 4-2 Possible settings in the min.conf file

Tag	Description	Optional/Required
TmcBinPath	Path of TMC (absolute path, detected automatically)	Optional
ModuleName	Path of test library/libraries (absolute path if the module is not in a location specified in ModSearchPaths)	Required
ConfigFile TestCaseFile	A configuration / test case file where the test cases are defined. There can be several configuration / test case files for one test module	Optional

Table 4-2 shows the possible logger settings in the *min.conf* file.

Table 4-3 Possible logger settings in the min.conf file

Tag	Description	Optional/Required
`EmulatorBasePath`	Directory that will be used for logger output.	Optional
`EmulatorFormat`	Output format, can be either TXT or HTML. Can be also both of them.	Optional
`EmulatorOutput`	Output facility, can be NULL either FILE or SYSLOG, or both of them.	Optional
`ThreadIdToLogFile`	Indicates if PID should be a part of output file name	Optional
`CreateLogDirectories`	Indicates if logger output directory should be created if not exists, possible values are YES or NO	Optional
`WithTimeStamp`	Indicates if time stamp should be added to the log messages, possible values are YES or NO	Optional
`WithLineBreak`	Indicates if line breaks should be added, possible values are YES or NO	Optional
`FileCreationMode`	File creation mode, possible values are APPEND or OVERWRITE	Optional
`LogLevel` •	LogLevel value, can be one of following: Trace, Debug, Notice, Info, Warning, Error, Fatal	Optional, default value is Info.

4.4 Configuring MIN Modules through /etc/min.d directory

MIN searches for module configurations (files containing [New_Module] … [End_Module] definitions) also from directory /etc/min.d. The idea is that if test modules are implemented for

example as debian packages, the min.conf does not need to be patched during the package (un)install.

5. Using MIN for test cases execution

This chapter describes how the MIN interface modules (MIN Console UI and the MIN EXT interface) can be used for test cases execution.

5.1 Working with MIN Console UI

MIN Console UI can be executed in normal text console.

Note: MIN Console UI requires the `Ncurses` library.

The MIN Console UI application provides a simple way to show test cases to the tester. It also shows to the tester the progress of ongoing test cases and the results of the executed test cases.

MIN Console UI is a menu-based application, which requires that at least two navigation keys are available. One key is used to activate an item from the menu, and the other key is used to change the selection. MIN Console UI also supports four-way navigation, which allows easier navigation in the menu system.

5.1.1 Startup parameters

MIN startup script has several command line parameters.

--version, -v prints version information.
--help, -h prints help screen with usage description.
--console, -c starts MIN without the console UI. All test cases defined in min.conf are executed sequentially (unless specified with –x switch)
-f forces MIN to kill already running MIN instance without prompting user.
--info, -i print info about given test module.
--execute, -x loads MIN with given test module (and optional configuration file(s)

5.1.2 Test module identification

All test module templates have their version and type compiled-in. It is possible, by using a command line switch, to have those data printed on the screen so that one can identify its test modules

```
maemo@maemo:~$ min --info /home/maemo/.min/SUnitTestModule.so
Module type: MINUnit
Module version: 200832
Build date: Jul 31 2008 13:43:24
```

Similar information can be also seen from the syslog output when the module is loaded

```
Aug  1 16:08:28 maemo TMC[27027]: tllib.c:136:tl_open():Module: MINUnit, Version: 200832, Build: Jul 31 2008 13:43:24
```

- Module type indicates the type of the module.
- Module version indicates the release on which the template has been modified.

- Module build denotes the date and time on which the module has been compiled.

5.1.3 Starting MIN Console UI

To start the MIN Console UI from the command line, type (note that MIN have to be installed):

```
min
```

When MIN starts it checks if:
1. there is already MIN running
2. there are TMC processes
3. there is message queue left behind.

When a MIN instance is found during startup, the default behavior is to ask the user if they want it to be killed (this behavior can be changed by using the –f parameter. See Section 5.1.1 for more information).

When starting MIN, it searches min.conf file from following locations:
1. Current working directory
2 ./home/<user name>/.min
3. /etc

5.1.4 Menu navigation

All menus are lists where pressing the Arrow down key changes the position down by one menu item and the Arrow up key changes the position up by one item. If the end of the list is reached, the position is changed to the beginning of the list, and vice versa.

A menu item is selected using the Arrow right key.

The Arrow left key is used to navigate back to the previous menu.

Selecting **Exit** in the main menu closes MIN Console UI.

5.1.5 Loading a test module

TBD

5.1.6 Starting a test case

To start a single test case, go to **Start new case** under the **Case** menu and select a particular case from the list. MIN Console UI starts to execute the chosen test case. A new test case can be started while another is still executed. When several cases are executed at the same time, they are executed in separate processes.

To start multiple test cases, go to **Run multiple tests** under the **Case** menu and select/mark test cases to be executed from the list and run selected test case either sequentially or in parallel.

5.1.7 Lists of started test cases

The **Case** menu offers a list of **Ongoing cases**, **Executed cases**, **Passed cases**, **Failed cases** and **Aborted/Crashed cases**. When selecting one of these, a list of the cases is shown on the display. The list is updated whenever the test case status is changed. For example, when a case is finished, it is moved from the Ongoing cases list to either the list of passed or failed cases, depending on the result of the test case.

5.1.8 Aborting, suspending and resuming a test case

When selecting a case from the Ongoing cases list, a new menu is shown that allows aborting or to suspend the test case execution, or resume the execution in case it was suspended.

5.1.9 Viewing the test case output

To view the test case output, select a case from any of the case lists. A new operation-specific menu is shown. By selecting **View Output**, the user will see the test case output.

By pressing any key, the focus is returned to the previous menu.

5.1.10 Test sets

The MIN Console UI application can be used to create a test set containing a set of test cases that can be run either sequentially or in parallel. MIN Console UI supports one test set at a time, but a different test set can be loaded via **Load test set** menu, which specifies the test set name.

Test set controlling is handled from the **Test set** menu that can be found from the main menu of MIN Console UI. The **Test set** menu provides the possibility to create or load a test set, or to control a test set (add/remove test cases, save, remove or execute test set).

The saved test name is created using time stamp date and time for example *2008-1-23,14:52.set*. The default path for test sets is */home/<username>/.min*. This is the location to which test sets are saved and from which they are loaded.

5.2 Working with the MIN command line interface

MIN can execute test cases also directly from the command line. User can tell MIN not to start the console UI with *–console* (or *-c*) command line switch. If *–c* is the only switch MIN executes all the test cases in all the modules configured in the min.conf file. MIN can also be instructed to omit the module definitions in min.conf and execute the cases from a test module specified on the command line with switch *–execute* (or *–x*).

```
maemo@maemo-desktop:~$min --console --
execute /usr/lib/min/scripter.so:/home/maemo/.min/script.cfg --
execute /home/maemo/.min/min84TestModule.so
Checking for active min processes...
Checking for active tmc processes...
Checking for MQ left behind...
Test case gathering...1
  2008-08-29 07:17:39
    ---------
  * Test Module /usr/lib/min/scripter.so
    ---------
    Test Case: scripted test
      o Test Result : success
```

```
    o Result Descr: Scripted test case passed
    o Message     : Passed
  ---------
* Test Module /home/maemo/.min/min84TestModule.so
  ---------
  Test Case: min84_fixed
    o Test Result : success
    o Result Descr: PASSED
    o Message     : Passed
  ---------
  Test Case: min84_2
    o Test Result : failure
    o Result Descr : AssertEquals failed
    o Message     : Failed
```

Note that it's possible to pass also one or more configuration files to the test module from command line (as in the example) above.

MIN does not then show the data that the test module wants to print on the screen. It only shows the test case that is running and the result of the test case when it is completed.

If the *–x* switch is used without the –c switch the console UI is started, but only the test module specified on the command line is visible in the menus.

6. Using MIN for test cases implementation

This chapter describes how MIN can be used for implementing test cases.

A test module contains the actual test case implementation. Test modules are implemented as separate libraries that MIN dynamically loads. Test modules can be freely implemented, as long as they use the required interface, which TMC can use. Test module can be Hardcoded, normal or MINUnit. These test module types can be seen in Figure 6-7. Test modules are indicated with bolded font.

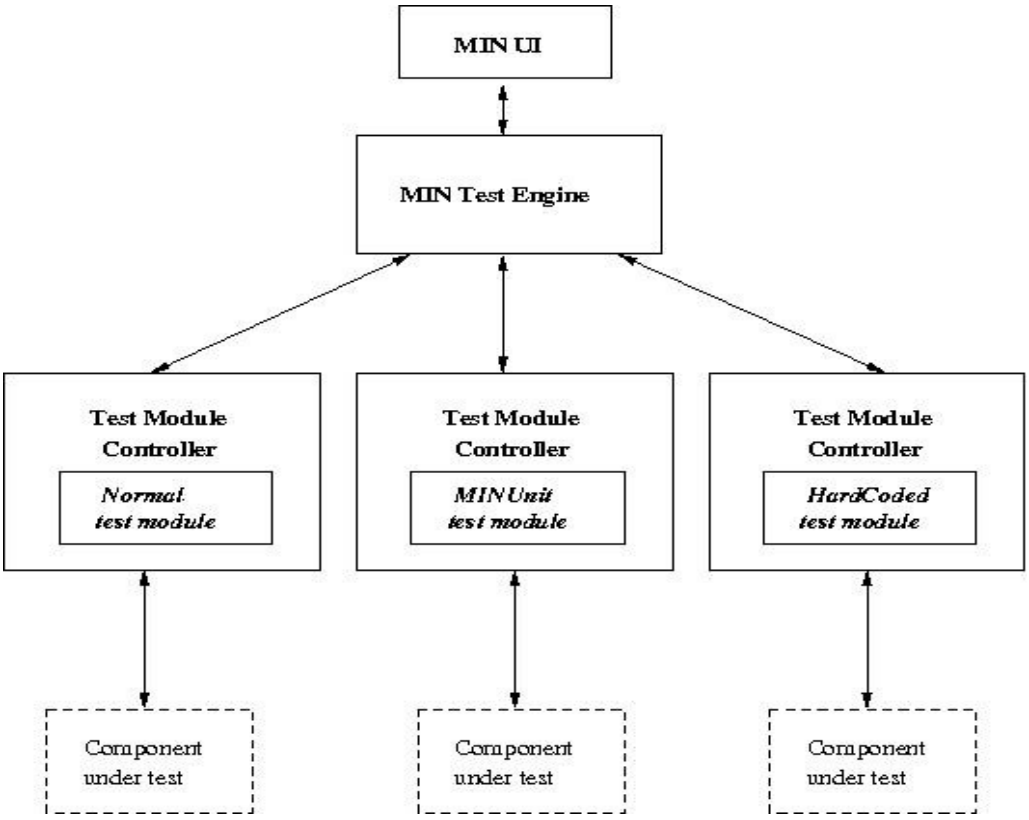

Figure 6-7 Test module types

6.1 Test Module API

Test Module API is a user-friendly API for test case execution

A test module has to provide the following interfaces to MIN:

- A list of the test cases
- Specified test case execution

These features are provided by the Test Module API that the test module has to implement in order fulfill MIN test module requirements.

In addition, there are some optional features that are offered by the MIN services API. MIN Services is interface from the test module to MIN Engine. For example, the interface offers UI independent printing functions, which can be used to show the test case progress on the screen, if there is a screen available (e.g. the console screen with the MIN Console UI application).

6.1.1 tm_get_test_cases

The `tm_get_test_cases` function is used to query test cases from a test module. The test case file is provided as a parameter (optional). This method is called for every test case file separately.

6.1.2 tm_run_test_case

The `tm_run_test_case` function is used to execute a specified test case. The test case file is provided as a parameter (optional). The method returns the result of the executed test case.

6.1.3 tm_initialize

The tm_initialize function is called once, when the test module is loaded. Function does not take arguments and does not return any results. It is not mandatory to implement it.

6.1.4 tm_finalize

The tm_finalize function is called once when module is closed. Function does not take any arguments and does not return results. It is not mandatory to implement it.

6.2 Test Module API features

The API functions provided to the test module provides optional features to the test module:

- `Printf` (output to UI)

6.2.1 tm_printf

The printing mechanism is based on asynchronous requests. Test process can send a printing request for TMC, which forwards request to UI.

6.3 Creating test module templates

Within MIN, there is a specific MIN Test Module Template Wizard for automatically creating a test module template, for example, a Hardcoded test module, based on the templates.

The test module template type can be `hardcoded, normal, testclass` or `minunit`. These test module template types can be seen in Figure 6-7. In the figure, the test module template types are indicated with *cursive* font. The figure does not contain detailed information for each test module; for example, the test case files are not shown.

Note: In order to use make, you need to have Makefile generated which is done by executing $sh build.sh in the test module directory (see Example 6.3).

MIN Test Module Template Wizard can be launched by running the *createtestmodule* script.

MIN Test Module Template Wizard takes the test module template, makes a new copy of it, and changes all the Test Module specific definitions (names) in the code files. MIN Test Module

Template Wizard asks from the user the test module template type and name, as well as the path where the new test module template is created.

The test module template type `hardcoded` creates a test module template where every test case is implemented as a separate function.

The test module template type `normal` creates an empty test module template where the test cases are implemented under the `tm_get_test_cases` and `tm_run_test_cases` methods.

The test module template type `minunit` creates MINUnit type of template.

An example of the test module creation process is shown in Example 6.2.

Example 6.2 Creating a test module using the createtestmodule script

```
maemo@maemo-desktop:~/MIN/TestModuleTemplates$ ./createtestmodule
Enter ModuleType:
        normal = creates an empty test module.
        hardcoded = creates test module that uses Hardcoded test cases.
        testclass = creates test class which is used with TestScripter.
        minunit = creates xUnit compatible test module
        exit = Exit.
Enter ModuleType: minunit
Enter ModuleName which has to be a valid C++ variable name.
Enter ModuleName (or exit): example_module
Enter path [default is drive root] (or exit): /tmp
Create test module of type minunit with name example_module to /tmp/
Starting module creation to /tmp/example_module/
Processing .
Processing CreateMINUnitModule
Processing inc
Processing MINUnitXXX.h
Processing min_unit_macros.h
Processing group
Processing AUTHORS
Processing Makefile.cvs
Processing NEWS
Processing MINUnitXXX_DoxyFile.txt
Processing build.sh
Processing README
Processing configure.in
Processing ChangeLog
Processing Makefile.am
Processing src
Processing MINUnitXXX.c
Processing MINUnitXXXCases.c
Module created to /tmp/example_module/
maemo@maemo-desktop:~/MIN/TestModuleTemplates$
```

An example of compiling the recently created module is shown in Example 6.3.

Example 6.3 Compiling the test module

```
maemo@maemo-desktop:/tmp/example_module$ sh build.sh
```

The build script defines your test library automatically to min.conf file (min.conf path is /home/<user name>/.min/).

It is also possible to make Debian package from your test library using command:

```
maemo@maemo-desktop:/tmp/example_module $make package
```

It makes min-testlib-<your library name>.deb into current directory. Verify/change the target in ../debian/control file before making the package.

When test module Debian package is installed, it automatically defines it self to /home/<user name>/.min/min.conf.

If you now start MIN newly created module should be visible on the console UI.

6.4 Implementing test cases for a Hardcoded test module

Implement a test case as a separate function to a *<own_module_name>.c* file using the following type of method:

```
int test_X( TestCaseResult* tcr )
```

Add your new function also into `int tm_get_test_cases( const char * cfg_file, DLList ** cases )` –function.

```
ENTRY(*cases,"My new test case", test_X);
```

See also examples from min/src/test_libraries/. For example testlibrary2.c shows an example of a test module that has all the test cases implemented inside a test module.

6.5 Implementing test cases for a normal test module

To add new test cases to test modules that have just been created, the user must modify the functions `tm_get_test_cases()` and `tm_run_test_case()`.

MIN calls the `tm_get_test_cases()` function to get the test cases from the test module.

MIN calls the `tm_run_test_case()` function to execute a test.

6.6 Implementing test cases for a MINUnit test module

Implement a test case as a separate function to a *<own_module_name>Cases.c* file using the following type of function:

```
MIN_TESTDEFINE(test_dllist_create)
{
          MIN_ASSERT_NOT_EQUALS( 1, 0 );
}
```

The file consists of two sections `TEST_VAR_DECLARATIONS` and `TEST_CASES`.

Variables that are common for all test cases are defined in the TEST_VAR_DECLARATIONS section.

The TEST_CASES section contains test fixtures (MIN_SETUP and MIN_TEARDOWN) and the test cases (MIN_TESTDEFINE). Code placed in the MIN_SETUP is executed for each test case before the test case execution; and MIN_TEARDOWN after each test case. The MIN_SETUP section should be used to initialize the common variables and possible startup routines. Similarly MIN_TEARDOWN can be used for clean up routines (e.g., freeing memory).

Implementation of test case itself is done by adding new MIN_TESTDEFINE section to TEST_CASES section. Name of the new test cases is defined as a parameter for the macro.Needed test case functionality must be placed in this section (apart from setup and teardown activities).

The result of each test case is determined by using of one or several MIN_ASSERT_ macros. If any of the MIN_ASSERT_ macros discovers that the test case result is other than expected, test case result will be marked as failed.

7. Integrating MIN tests to build environment

Often a source package contains test cases that are built as part of the package, and can be run to verify that the software is functioning properly, after it has been built. This chapter explains how MIN could be used for this purpose.

7.1 Support in Test Module Template Wizard

Makefiles for HardCoded, Normal and MINUnit test modules, created by the Test Module Template Wizard (createtestmodule), contain a make target for executing all the tests in the module with command *make check*.

```
maemo@maemo-desktop:~/modulecreatedbywizard$ make check
make  modulecreatedbywizardTestModule.so
make[1]: Entering directory `/home/maemo/modulecreatedbywizard'
make[1]: `modulecreatedbywizardTestModule.so' is up to date.
make[1]: Leaving directory `/home/maemo/modulecreatedbywizard'
make  check-TESTS
make[1]: Entering directory `/home/maemo/modulecreatedbywizard'
Checking for active min processes...
Checking for active tmc processes...
Checking for MQ left behind...
Checking for SHM left behind...
MIN Test Framework, © Nokia 2008, All rights reserved,
licensed under the Gnu General Public License version 2,
Contact: Antti Heimola, DG.MIN-Support@nokia.com

Test case gathering...1
  2008-10-13 04:18:10
    ---------
  * Test
Module /home/maemo/modulecreatedbywizard/./modulecreatedbywizardTestModule.so
    ---------
    Test Case: modulecreatedbywizard_1
      o Test Result: success
      o Result Desc: PASSED
      o Message    : Passed
    ---------
    Test Case: modulecreatedbywizard_2
      o Test Result: failure
      o Result Desc: AssertEquals failed
      o Message    : Failed
FAIL: modulecreatedbywizardTestModule.so
====================
1 of 1 tests failed
====================
make[1]: *** [check-TESTS] Error 1
make[1]: Leaving directory `/home/maemo/modulecreatedbywizard'
make: *** [check-am] Error 2
```

7.2 Test modules in one directory approach

If there are many test suites to be added to the build, creating them individually with *Test Module Template Wizard* may be undesired, since each module is generated to own directory, and contains files that could be shared between the modules. For MINUnit a new way of creating test

modules is introduced. MIN now installs from the folder min/shared a file (minunit.c), which can be used by all MINUnit modules, so that the file is compiled with a pre-processor flag stating the file containing the test cases. Example shows Makefile.am for two test modules used in this fashion.

```
check_PROGRAMS = testmodulea.so testmoduleb.so
TESTS = $(check_PROGRAMS)
TESTS_ENVIRONMENT = min --console --execute

testmodulea_so_CFLAGS = -DCASES_FILE='"a_testCases.c"'
testmodulea_so_LDFLAGS = -shared
testmodulea_so_SOURCES = ../shared/minunit.c
testmodulea_so_LDADD =  -lminutils -lmintmapi -lminevent

testmoduleb_so_CFLAGS = -DCASES_FILE='"b_testCases.c"'
testmoduleb_so_LDFLAGS = -shared
testmoduleb_so_SOURCES = ../shared/minunit.c b_test_utils.c
testmoduleb_so_LDADD =  -lminutils -lmintmapi -lminevent
```

MIN package contains unit tests, that can be executed to verify operation of MIN components, by executing command *make check* in directory *min/* after MIN has been built and installed. The traditional CHECK tests are in directory *min/tests,* and tests that are using MIN tool itself can be found from *min/mintests*.

8. Using MIN Parser for test data parsing

This chapter contains the MIN Parser API description for guidance on how to use MIN Parser for test data parsing.

shows how MIN Parser is involved in MIN. Test Engine uses MIN Parser for parsing data from the MIN configuration file (min.conf). MIN Parser can be used when parsing test data for test modules. Test data can be included in a test module's test case file, a test module's initialization file or a buffer.

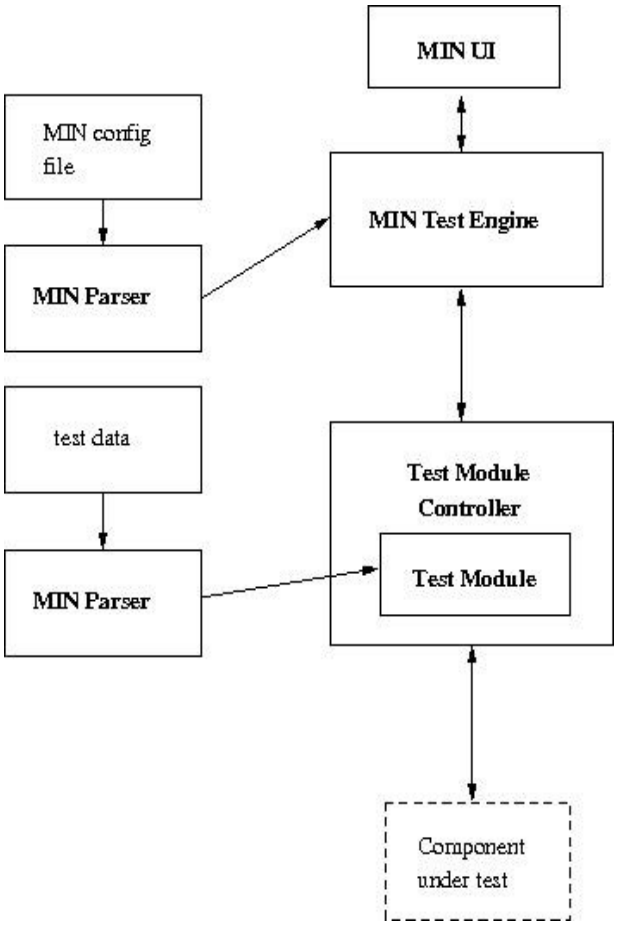

Figure 8-8 MIN Parser

8.1 MIN Parser API

MIN Parser is divided into three main parts: `MinParser`, `MinSectionParser` and `MinItemParser` (see Figure 8-9).

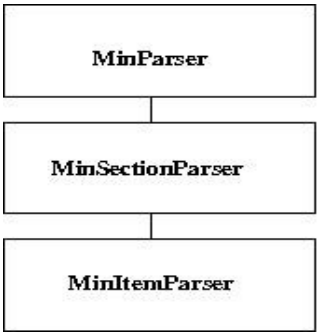

Figure 8-9 MIN Parser main parts

MIN Parser supports hierarchical parsing for:

- Sections
- Subsections
- Lines
- Strings
- Characters
- Integers

MIN Parser also supports INCLUDE command. See the following limitations of this feature:

- INCLUDE keyword must be written in capital letters and must start from the first column of the line.
- File name (with path and extension) must follow INCLUDE command. Rest of line would be ignored.
- All files included from Unicode file should also be in Unicode format (and vice versa).
- Loops in includes are not allowed (for example incorrect situation is when file A includes file B and file B includes file A). In that case, the second include will be ignored, and MIN will continue working).

8.1.1 MinParser

The `MinParser` component functionality opens and reads test data. The purpose of `MinParser` is to parse a required section of the test data. The section may be a whole test data file or some part of the data file.

The main functions of `MinParser` are:

- `mp_create` for creating a parser with path and file information (see Table 7.1)
- `mp_create_mem` (see Table 7.2)
- `mp_destroy` (see Table 7.3)
- `mp_section` (see Table 7.4)
- `mp_next_section` (see Table 7.5)

Table 8-1 create for creating a parser with path and file information

Method	**Description**
`mp_create`	Creates MinParser with path and file information. Possible errno return values: `ENOENT`: invalid path argument. `EACCES`: permission denied.
Parameters	**Description**
`const TPtrC* path`	The path to folder where config file located.
`const TPtrC* file`	The filename of config file.
`TCommentType comments`	Indication of the comment type. • `ENoComments`: Comments are included with parsing. • `ECStyleComments`: The user wants to parse sections without c-style comments.
Return value	**Description**
`MinParser*`	`MinParser struct object.`

Table 8-2 mp_create_mem for creating a parser with buffer information

Method	**Description**
`mp_create_mem`	Creating a parser with buffer information.
Parameters	**Description**
`const TPtrC* buffer`	The buffer to be parsed.
`TCommentType comments`	Indication of the comment type. • `ENoComments`: Comments are included with parsing. • `ECStyleComments`: The user wants to parse sections without c-style comments.
Return value	
`MinParser*`	`MinParser` struct object.

Table 8-3 mp_create_mem for creating a parser with buffer information

Method	**Description**
`mp_destroy`	Destroy MinParser struct component and free allocated memory and resources.
Parameters	**Description**
`MinParser** mp`	Pointer reference to used MinParser struct component. `INITPTR` pointer value is returned in mp pointer parameter if destroying operation completed successfully.
Return value	
`void`	None.

Table 8-4 mp_section

Method	Description
`mp_section`	Open and read configuration source and parses a required section. If start tag is empty the parsing starts begin of the configuration file. If end tag is empty the parsing goes end of configuration file. This function will parse next section after the earlier section if the sought parameter is set to 1. If configuration file includes several sections with both start and end tags so sought parameter seeks the required section. Possible errno return values: `EINVAL`: invalid value was passed to the function.
Parameters	**Description**
`MinParser *mp`	Pointer to MinParser instance.
`const TPtrC* start_tag`	Start tag name. If the start tag is empty, the parsing starts at the beginning of the file.
`const TPtrC* end_tag`	End tag name. If the end tag is empty, the parsing goes to the end of the file.
`int seeked`	Indicates the tag on which the parsing ends.
Return value	**Description**
`MinSectionParser*`	`MinSectionParser` struct object.

Table 8-5 mp_next_section

Method	**Description**
`mp_next_section`	Open and read configuration source and parses a required section. If start tag is empty the parsing starts begin of the configuration file. If end tag is empty the parsing goes end of configuration file. This method will parse next section after the earlier section if sought parameter is set to 1. If configuration file includes several sections with both start and end tags so sought parameter seeks the required section. Possible errno return values: `EINVAL`: Invalid value was passed to the function.
Parameters	**Description**
`MinParser *mp`	Pointer to MinParser instance.
`const TPtrC* start_tag`	Start tag name. If the start tag is empty, the parsing starts at the beginning of the file.
`const TPtrC* end_tag`	End tag name. If the end tag is empty, the parsing goes to the end of the file.
`int seeked`	A section indicator.
Return value	**Description**
`MinSectionParser*`	`MinSectionParser` struct object.

8.1.2 MinSectionParser

The purpose of `MinSectionParser` is to parse the required lines of the section to forward operations.

The main methods of `CMinSectionParser` are:

- `msp_create` (see Table 7.6)
- `msp_destory` (see Table 7.7)
- `msp_get_item_line` (see Table 7.8)
- `msp_get_next_item_line` (see Table 7.9)
- `msp_get_next_tagged_item_line` (see Table 7.10)
- `msp_sub_section` (see Table 7.11)
- `msp_next_sub_section` (see Table 7.12)
- `msp_next_sub_section` (see Table 7.13)
- `msp_get_line` (see Table 7.14)
- `msp_get_next_line` (see Table 7.15)
- `msp_get_next_tagged_line` (see Table 7.16)
- `msp_get_position` (see Table 7.17)
- `msp_set_position` (see Table 7.18)
- `msp_set_data` (see Table 7.19)
- `msp_des` (see Table 7.20)

Table 8-6 msp_create

Method	**Description**
`msp_create`	Creates a new MinSectionParser.
Parameters	**Description**
`unsigned int length`	The length of the section to be parsed. Possible errno return values: `ENOMEM: No sufficient memory to allocate new struct object.`
Return value	**Description**
`MinSectionParser*`	`MinSectionParser struct object.`

Table 8-7 msp_destroy

Method	**Description**
`msp_destroy`	Destroys previously created MinSectionParser.
Parameters	**Description**
`MinSectionParser** msp`	The reference pointer to MinSectionParser struct component. Returns address of MinItemParser or INITPTR in case of failure. Possible errno return values: `ENOMEM: No sufficient memory to allocate new struct object.`
Return value	**Description**
`void`	`None`

Table 8-8 msp_get_item_line

Method	**Description**
`msp_get_item_line`	Parses the next line for items parsing with a tag. Returns address of MinItemParser or INITPTR in case of failure.
Parameters	**Description**
`MinSectionParser* msp`	The reference pointer to MinSectionParser struct component.
`const TPtrC* tag`	Indicates the tag to start with. If start tag is empty the parsing starts begin of the section.
`TTagToReturmValue tag_indicator`	Idicates if tag will be included to the returned object.
Return value	**Description**
`MinItemParser*`	`MinItemParser` struct object.

Table 8-9 msp_get_next_item_line

Method	**Description**
`msp_get_next_item_line`	Parses a next line for items parsing. Returns pointer of MinItemParser.
Parameters	**Description**
`MinSectionParser* msp`	MinSectionParser entity to operate on.
Return value	**Description**
`MinItemParser*`	`MinSectionParser` construct object.

Table 8-10 msp_get_next_tagged_item_line

Method	**Description**
`msp_get_next_tagged_item_line`	Parses a next line for items parsing with a tag. Returns pointer to MinItemParser struct component.
Parameters	**Description**
`MinSectionParser* msp`	MinSectionParser entity to operate on.
`const TPtrC* tag`	Indicates the tag to start with. If start tag is empty the parsing starts begin of the section.
`TTagToReturnValue tag_indicator`	Indicates if tag will be included to the returned object.
Return value	**Description**
`MinItemParser*`	`MinItemParser` struct object.

Table 8-11 msp_sub_section

Method	**Description**
`msp_sub_section`	Parses a sub sections from the main section with a start and with an end tag. Returns pointer to MinSectionParser struct component.
Parameters	**Description**
`const TPtrC* start_tag`	Indicates the tag to start on.
`const TPtrC* end_tag`	Indicates the tag to end on.
`int seeked`	The sought parameter indicates subsection that will be parsed.

Return value	Description
`MinSectionParser*`	`MinSectionParser` struct object.

Table 8-12 msp_next_sub_section

Method	**Description**
`msp_next_sub_section`	Parses a next subsection from the main section with a start and with an end tag. Returns pointer of MinSectionParser or INITPTR in case of failure.
Parameters	**Description**
`MinSectionParser* msp`	MinSectionParser entity to operate on.
`const TPtrC* start_tag`	Indicates the tag to start on.
`const TPtrC* end_tag`	Indicates the tag to end on.
`int seeked`	The sought parameter indicates subsection that will be parsed.
Return value	**Description**
`MinSectionParser*`	`MinSectionParser` struct object.

Table 8-13 msp_next_sub_section

Method	**Description**
`msp_next_sub_section`	Parses a next subsection from the main section with a start and width an end tag.
Parameters	**Description**
`MinSectionParser* msp`	Pointer to the MinSectionParser struct component.
`const TPtrC* start_tag`	Pointer to start tag string.
`const TPtrC* end_tag`	Pointer to end tag string.
`int seeked`	The sought parameter indicates subsection that will be parsed.
Return value	**Description**
`MinSectionParser*`	Return pointer to MinSectionParser struct component or INITPTR pointer value in case of failure.

Table 8-14 msp_get_line

Method	Description
`msp_get_line`	Get a line from section with a tag. Possible errors: EINVAL errno value when invalid value was passed as a parameter.
Parameters	**Description**
`MinSectionParser* msp`	Pointer to the MinSectionParser struct component.
`const TPtrC* tag`	Indicates the tag to start with. If start tag is empty the parsing starts begin of the section.
`TPtrC** line`	Parsed line is returned through this variable.
`TTagToReturnValue tag_indicator`	Indicates if tag will be included to the returned line.
Return value	**Description**
`int`	Returns 0 in case of success, -1 in case of failure.

Table 8-15 msp_get_next_line

Method	Description
`msp_get_next_line`	Parses next line. Possible errors: EINVAL: when invalid value was passed as a parameter.
Parameters	**Description**
`MinSectionParser* msp`	Pointer to the MinSectionParser struct component.
`TPtrC** line`	Parsed line is returned through this variable.
Return value	**Description**
`int`	Returns value 0 in case of success, -1 in case of failure.

Table 8-16 msp_get_next_tagged_line

Method	Description
`msp_get_next_tagged_line`	Get next line with tag. Possible errors: EINVAL: when invalid value was passed as a parameter.
Parameters	**Description**
`MinSectionParser* msp`	Pointer to the MinSectionParser struct component.
`const TPtrC* tag`	Indicates the tag to start with. If start tag is empty the parsing starts begin of the section.
`TPtrC** line`	Parsed line is returned through this variable. If line is no found INITPTR pointer value is returned.
`TTagToReturnValue tag_indicator`	Indicates if tag will be included to the return line.
Return value	**Description**
`int`	Returns value 0 in case of success, -1 in case of failure.

Table 8-17 msp_get_position

Method	Description
`msp_get_position`	Get current position. Possible errors: EINVAL: when invalid value was passed as a parameter.
Parameters	**Description**
`MinSectionParser* msp`	Pointer to the MinSectionParser struct component.
Return value	**Description**
`int`	Returns current parsing position, or value -1 in case of failure.

Table 8-18 msp_set_position

Method	Description
`msp_set_position`	Set position. This function can be used to set parsing position, e.g. to rewind back to some old position retrieved with GetPosition. Possible errors: EINVAL: when invalid value was passed as a parameter.
Parameters	**Description**
`MinSectionParser* msp`	Pointer to the MinSectionParser struct component.
`unsigned int pos`	Indicates the position to which section parser should go.
Return value	**Description**
`int`	Returns error value in case of failure.

Table 8-19 msp_set_data

Method	Description
`msp_set_data`	Create a section.
Parameters	**Description**
`MinSectionParser* msp`	Pointer to the MinSectionParser struct component.
`const TPtrC* data`	String data which are to be parsed.
`TPtrC* start_pos`	Position from which parsing should start.
`unsigned int length`	Length of the data in bytes.
Return value	**Description**
`void`	Nothing returns.

Table 8-20 msp_des

Method	Description
`msp_des`	Returns current section.
Parameters	**Description**
`const MinSectionParser* msp`	Pointer to the MinSectionParser struct component.
Return value	**Description**
`const TPtrC*`	Return pointer to the data that parser is parsing.

8.1.3 MinItemParser

The purpose of `MinItemParser` is to parse strings, integers and characters.

The main methods of `MinSectionParser` are:

- `mip_create` (see Table 7.21)
- `mip_destroy` (see Table 7.22)
- `mip_parse_and_end_pos` (see Table 7.23)
- `mip_get_string` (see Table 7.24)
- `mip_get_next_string` (see Table 7.25)
- `mip_get_next_tagged_string` (see Table 7.26)
- `mip_get_int` (see Table 7.27)
- `mip_get_next_int` (see Table 7.28)
- `mip_get_next_tagged_int` (see Table 7.29)
- `mip_get_uint` (see Table 7.30)
- `mip_get_next_uint` (see Table 7.31)
- `mip_get_next_tagged_uint` (see Table 7.32)
- `mip_get_char` (see Table 7.33)
- `mip_get_next_char` (see Table 7.34)
- `mip_get_next_tagged_char` (see Table 7.35)
- `mip_get_remainder` (see Table 7.36)
- `mip_set_parsing_type` (see Table 7.37)
- `mip_get_parsing_type` (see Table 7.38)

Table 8-21 mip_create

Method	**Description**
`mip_create`	Creates MinItemParser struct component with given string data.
Parameters	**Description**
`TPtrC* section`	Pointer to section string data.
`int start_pos`	Start point in given section string by integer value.
`int length`	Length of given section string.
Return value	**Description**
`MinItemParser*`	Pointer to created new MinItemParser struct component.

Table 8-22 mip_destroy

Method	Description
`mip_destroy`	Frees allocated memory of MinItemParser struct component. Returns INITPTR if destroying operation complete successfully.
Parameters	**Description**
`MinItemParser** msp`	Pointer to MinItemParser struct component which to be destroyed.
Return value	**Description**
`void`	None.

Table 8-23 mip_parse_start_and_end_pos

Method	Description
`mip_parse_start_and_end_pos`	Parses MinItemParser section and returns results with start, end and extra end positions including length of parsed part of section. If start tag keyword is used then method searches string data after this given keyword. If start tag is not used then parsing starts at the beginning of section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC* start_tag`	A reference to the parsed string.
`TPtrC** ref_start_pos`	Reference pointer start position of parsed section part.
`TPtrC** ref_end_pos`	Reference pointer end position of parsed section part.
`TPtrC** ref_extra end_pos`	Reference pointer extra end position of parsed section. Extra end position is section string location after last quote.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully. or -1 if operation failed.

Table 8-24 mip_get_string

Method	**Description**
`mip_get_string`	Gets first string according to tag from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC* tag`	Pointer to tag string for searching section part string.
`TPrtC** string`	Reference string pointer for parsed string result.
Return value	**Description**
int	Returns error value 0 if parsing operation completed successfully. or -1 if operation failed.

Table 8-25 mip_get_next_string

Method	**Description**
`mip_get_next_string`	Gets next string according to previous section part string from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC** string`	Reference string pointer for parsed result string.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully. or -1 if operation failed.

Table 8-26 mip_get_next_tagged_string

Method	Description
`mip_get_next_tagged_string`	Gets next tagged string according to tag and earlier used StiItemParser section sting. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC* tag`	Tag string for search of parsing. If tag string is empty then parsing start at the beginning of section string.
`TPtrC** string`	Reference string pointer for parsed result string.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully. or -1 if operation failed.

Table 8-27 mip_get_int

Method	Description
`mip_get_int`	Gets first integer value according to tag from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC* tag`	Tag string for search of parsing. If tag string is empty then parsing starts the beginning of section string.
`int* value`	Reference string pointer for found integer value by parsing operation.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully. or -1 if operation failed.

Table 8-28 mip_get_next_int

Method	Description
`mip_get_next_int`	Gets next integer value according to tag and earlier used StiItemParser section sting. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`init* value`	Reference string pointer for parsed result string.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully. or -1 if operation failed.

Table 8-29 mip_get_next_tagged_int

Method	Description
`mip_get_next_tagged_int`	Gets next tagged integer value according to tag from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC* tag`	Tag string for search of parsing.
`int* value`	Reference string pointer for found integer value by parsing operation.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully. or -1 if operation failed.

Table 8-30 mip_get_uint

Method	Description
`mip_get_uint`	Gets first unsigned integer value according to tag from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC* tag`	Tag string for search of parsing. If tag string is empty then parsing starts at the beginning of section string.
`unsigned int* value`	Reference string pointer for found unsigned integer value by parsing operation.
Return value	**Description**
`Int`	Returns error value 0 if parsing operation completed successfully. or -1 if operation failed.

Table 8-31 mip_get_next_uint

Method	Description
`mip_get_next_uint`	Gets next unsigned integer value according to previous parsing operation by tag from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`unsigned int* value`	Reference string pointer for found unsigned integer value by parsing operation.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully. or -1 if operation failed.

Table 8-32 mip_get_next_tagged_uint

Method	Description
`mip_get_next_tagged_uint`	Gets next tagged unsigned integer value according to previous parsing operation by tag from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC* tag`	Tag string for search of next tagged unsigned integer data.
`unsigned int* value`	Reference string pointer for found unsigned integer value by parsing operation.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully. or -1 if searching operation failed.

Table 8-33 mip_get_char

Method	Description
`mip_get_char`	Gets first character by tag from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC* tag`	Tag string for search of next tagged unsigned integer data.
`TPtrC** chr`	Reference string pointer for found character data by parsing operation.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully. Possible error codes: -1 if section string length is zero. -2 if parsing operation failed by any reason.

Table 8-34 mip_get_next_char

Method	Description
`mip_get_next_char`	Gets next character according to previous parsing operation from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC** chr`	Reference string pointer for found character data by parsing operation.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully, or -1 if next character searching failed.

Table 8-35 mip_get_next_tagged_char

Method	Description
`mip_get_next_tagged_char`	Gets next tagged character by tag from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC* tag`	Tag string for search of next tagged unsigned integer data.
`TPtrC** chr`	Reference string pointer for found tagged character data by parsing operation.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully, or -1 if next character searching failed.

Table 8-36 mip_get_remainder

Method	Description
`mip_get_remainder`	Gets remainder string from MinItemParser section string. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TPtrC** string`	Reference string pointer for found remainder string by parsing operation.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully, or -1 if remainder string parsing failed.

Table 8-37 mip_set_parsing_type

Method	Description
`mip_set_parsing_type`	Sets parsing type value for MinItemParser struct component. Possible errors: `EINVAL` errno value when parsing operation failed.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.
`TParsingType type`	New parsing type for parsing operations.
Return value	**Description**
`int`	Returns error value 0 if parsing operation completed successfully, or -1 if MinItemParser struct component not available.

Table 8-38 mip_get_parsing_type

Method	Description
`mip_get_parsing_type`	Gets parsing type value from MinItemParser section string. Possible errors: `EINVAL` errno value when MinItemParser struct component not available.
Parameters	**Description**
`MinItemParser* mip`	Pointer to MinItemParser struct component.

Return value	Description
`TParsingType`	Returns parsing type value if parsing operation completed successfully, or -1 if operation failed.

Note 1: `ENormalParsing` type parsing is takes string after tag keyword and is separated result strings by white spaces. For example, if the section string is `"Tag1 Word1 Word2 Tag2 Word3 Word4 Tag3 Word5 Word6"` if is used tag `"Tag2"` then parsed result string is `"Word3"` by `mip_get_string()` method. If after this parsing will did new parsing by `mip_get_next_string()` then next parsed result string is `"Word4"`.

Note 2: `EQuoteStyleParsing` type parsing takes string inside of two quotes and this parsed may including also white spaces. For example, if only the start tag has been found from the section string then parsing will be returned rest of string after start tag.

Note 3: If section string including comments strings with two backslashes then parsing will be returned string after the first backslashes.

9. Using Test Scripter for creating scripted test cases

9.1 Test Scripter test case file

The Test Scripter test case file defines the commands to be executed in a test case described with a simple scripting language. The test case file may contain several test cases, the descriptions of which start with the tag `[Test]` and end with the tag `[Endtest]`. The test case is executed sequentially line by line. The test case itself is described with keywords, arguments and argument value pairs between the start and end tags. An example of a test case is described below.

```
[Test]
title Create, print, run example and delete
createx TestScriptClass testobj
print Call Example method
testobj Example pa ra me ters
print Example method called, delete instance
delete testobj
[Endtest]
```

The test case title must be given after the `title` keyword on the first line of the test case definition. In the example above, the `create` keyword creates a new instance of `TestScriptClass`, which is named `test`. The `print` keyword is used to print information to the UI. The fourth line of the example test case executes the `Example` method of the `test` object with four parameters: `pa`, `ra`, `me` and `ters`. The `delete` keyword is used to delete the `test` object.

Test Scripter's test case file and test module initialization file may contain macro definitions used in test cases. Macros are defined inside a define section started with `[Define]` tag ended with `[Enddefine]` tag, for example:

```
[Define]
ENOERR 0
[Enddefine]
```

9.2 Setting up the Scripter

Test Scripter is configured for use by adding Test Scripter as a test module to MIN. The test case file is added as a test case file for Test Scripter. An example of configuring Test Scripter as a test module can be seen in Example 4.2.

9.3 Creating a test class

A new test class can be easily created with MIN Test Module Template Wizard that comes with the MIN release

$*createtestmodule*

- Give `testclass` as `ModuleType`.
- Enter a name and path for the test class.

A new test class is now created with the given name to the given path. The next step is to create the building block methods to the test class.

To create the building block methods to the test class:

1. Implement the building blocks to the /*<testclassname>/src/<testclassname>Cases.c* file (there is one example method, `ExampleTest`, that can be copy-pasted).
2. Add entry lines for every method to `function ts_get_test_cases` (*<testclassname>Cases.c*):

   ENTRY(*list,"Example",ExampleTest);
3. Add entry lines for every method to FORWARD DECLARATIONS:

   LOCAL int ExampleTest(MinItemParser* item);
4. Build the test class from the */<testclassname>/group* directory by using build.sh script. It will automatically copy test class library to /home/<user name>/.min.

Now the test class is created. The next step is to implement test cases to test the test case file used by Test Scripter. An example test case file can be found in */<testclassname>/group/Example.CFG*. The test case file is constructed using the script language defined in this document.

Copy the test case file to /home/<user name>/.min/

Now you should be able to execute your test class module by using console UI or EXT INTERFACE.

9.3.1.1 Accessing script variables from test class

Script local variables (see 9.3.2.4) can be retrieved from inside the test class as well as set to different value . The functions below are available for test class to set and get variable values.

```
int SetLocalValue (const char *varname, const char *varval);
/* ---------------------------------------------------------------------- */
/** Assign integer value to script variable
 * @param varname name of variable
 * @param varval value to assign
 * @return 0 on success, -1 on error
 */
int SetLocalValueInt (const char *varname, const long value);
/* ---------------------------------------------------------------------- */
/** Get value of script variable as an integer
 * @param varname name of variable
 * @param value [out] variable value
 * @return 0 on success, -1 on error
 */
int GetLocalValueInt (const char *varname, long *value);
/* ---------------------------------------------------------------------- */
/** Get value of script variable as a string
 * @param varname name of variable
 * @param value [out] variable value
 * @return 0 on success, -1 on error
 */
```

```
int GetLocalValue (const char *varname, char **value);
```

Note that test class can only get and set local script variables, it's not possible to declare variables in the tests class.

9.3.1.2 Scripter internal variables

Scirpter holds internal variables that are available to script or test class similar to local variables. The internal variables are counters shown in and can be used, for example, to bail out from a long test loop in case some error level is reached, or to call some special clean up function in case of crashed test method calls.

Table 9-4 Scripter internal variables.

Variable Name	Description
FAIL_COUNT	Number of failed method and combiner calls
CRASH_COUNT	Number of crashed method and combiner calls
TOUT_COUNT	Number of timed out method and combiner calls
ABORT_COUNT	Number of aborted method and combiner calls
ERROR_COUNT	Sum of all the of the above
TOTAL_COUNT	Total number of method and combiner calls

9.3.2 General keywords

9.3.2.1 title keyword

The `title` keyword is used to give a verbal description for a test case. The description is placed after the keyword. The `title` keyword is mandatory for every test case and must be placed as the first keyword in the test case description (see Section 9.1). For example:

```
title Create, print, run example and delete
```

9.3.2.2 timeout keyword

The `timeout` keyword is used to give a timeout value for a test case. The timeout value is given as an argument for the `timeout` keyword, as described in Table 9-5 below.

Table 9-5 timeout argument

Argument	Description
Timeout value	The timeout value in milliseconds.

The `timeout` keyword can be used, for example, in the following way (timeout 10 seconds):

```
timeout 10000
```

9.3.2.3 print keyword

The `print` keyword can be used to print, for example, progress information to the UI. The printed description is placed after the `print` keyword; see the example in Section 9.1

9.3.2.4 var keyword

The `var` keyword declares a variable for the scripter. A variable can be initialized during its declaration. The value of variable can be set in script or test class. When calling methods from a test object, the declared variable can be passed as an argument to this method. If variable name appears in text used with print keyword, its value gets printed.

The `var` keyword has one mandatory argument, described in Table 9-6 and one optional argument, described in Table 9-7.

Table 9-6 var mandatory argument

Argument	Description
variable name	The variable name.

Table 9-7 var optional argument

Argument	Description
variable value	The value which declared variable is to be initialized.

The `var` keyword can be used for example in the following way:

```
var variable2
var variable1 text
var variable2 1
sometestobj somemethod variable1 variable2
print variable2
```

9.3.3 Test Case control

9.3.3.1 createx keyword

The `createx` keyword is used to create a new instance of a test class. `createx` has two mandatory arguments, which are described in Table 9-8 below.

Table 9-8 createx arguments

Argument	Description
Test class name	The test class name for the new object.
Test object name	The name of the created new instance of the test class.

The `createx` keyword can be used, for example, in the following way:

```
createx TestScriptClass test
```

9.3.3.2 delete keyword

The `delete` keyword is used to delete an instance of a test class. `delete` has one mandatory argument, which is described in Table 9-9 below.

Table 9-9 delete argument

Argument	Description
Test object name	The name of the instance of the test class that is deleted.

The `delete` keyword can be used, for example, in the following way:

```
delete test
```

9.3.3.3 allownextresult keyword

The `allownextresult` keyword is used to add valid result values for a method and for asynchronous commands. The default value for the expected result is `0`, and if a value is set with `allownextresult`, `0` is removed from the expected values. A method may either return or leave with the specified result. Every method call removes all allowed results. That is, after every method call, the default value `0` is again the only expected result value. Either multiple `allownextresult` keywords can be placed before a method call or allownextresult keyword can have multiple parameters.

Table 9-10 allownextresult argument

Argument	Description
An error code	Error code, which is allowed from the next method. It can be custom value, or errno.

The `allownextresult` keyword can be used, for example, in the following way:

```
allownextresult -1
```

The `allownextresult` keyword can be used, for example, in the following way:

```
allownextresult -1 -2 -3 -4
```

9.3.3.4 allowerrorcodes keyword

The `allowerrorcodes` keyword is used to add valid result values for a method and for asynchronous commands. The default value for the expected result is `0`, and if a value is set with `allowerrorcodes`, `0` is removed from the expected values. A method may either return or leave with the specified result. Every method call removes all allowed results. That is, after every method call, the default value `0` is again the only expected result value. Either multiple `allowerrorcodes`

keywords can be placed before a method call or `allowerrorcodes` keyword can have multiple parameters.

Table 9-11 `allowerrorcodes` argument

Argument	Description
An error code	Error code, which is allowed from the next method. It can be custom value, or errno.

The `allowerrorcodes` keyword can be used, for example, in the following way:

```
allowerrorcodes -1
```

The `allowerrorcodes` keyword can be used, for example, in the following way:

```
allowerrorcodes -1 -2 -3 -4
```

9.3.3.5 sleep keyword

The `sleep` keyword is used to pause the execution of a test case for a specified timeout. `Sleep` has one mandatory argument, which is described in Table 9-12.

The `sleep` keyword only stops the test case line-runner active object for the specified period. All the other user active objects will continue to be serviced. That is, no further lines of the test case file will be executed during that delay, but the process is not halted; any user active objects may still be completed.

Table 9-12 `sleep` argument

Argument	Description
Timeout	The timeout for the pause, specified in milliseconds.

The `sleep` keyword can be used, for example, in the following way:

```
sleep 10000   // pause for 10 seconds
```

9.3.3.6 loop keyword

The `loop` keyword is used to repeat a section of the test case file for the specified number of iterations. The section to be repeated is enclosed with the `loop` and `endloop` keywords.

Table 9-13 loop argument

Argument	Description
Loop times	The loop count, that is, the number of times that the loop is executed.
`msec`	(optional) This keyword says that 'Loop times' argument stands for the time in milliseconds during which loop will be looped

The `loop` keyword can be used, for example, in the following way:

```
loop 5
// execute this 5 times
print pa ra me ter
endloop
```

9.3.3.7 breakloop keyword

The **`breakloop`** keyword is used to prematurely exit the current loop.

The `breakloop` keyword can be used, for example, in the following way:

```
loop 5
var exitcond
someclass somemethod someparam
if exitcond
 breakloop
endif
endloop
```

9.3.3.8 endloop keyword

The `endloop` keyword is used to specify the end of a looped section.

9.3.3.9 If, else and endif keywords

The if, else and endif keywords can be used for conditional execution in the script. The condition for if must be variable or value. Value 0 and string "false" (case insensitive) is interpreted as false, or other values as true. If block must always be closed with endif keyword. Nesting of if-blocks is supported.

```
var v
if v
  print is true
else
  print is false
endif
```

9.3.3.10 Object name

The test object name can be considered as a temporary keyword, which is valid between its creation with the `createx` keyword and its deletion with the `delete` keyword. The object name is

used to call methods from a test object. The method name is given as the first argument for the object name, and the method may have arguments, which are forwarded to the test class method.

For example: `TestObjectName MethodName` <method arguments 1 2 3>.

9.3.4 Event control

The keywords described in the following sections are used to control MIN Event System. For information about MIN Event System, see Chapter 8

9.3.4.1 request keyword

The `request` keyword is used to request an event. If someone wants to use an event, it must first be requested, and after that it can be waited. After the event is not used anymore, it must be released.

`request` has one mandatory argument, and one optional parameter. Both are described in Table 9-14 below.

Table 9-14 request mandatory argument

Argument	Description
Event	The event name.
State	Keyword "state" must be specified if request is for event of "state" type.

The `request` keyword can be used, for example, in the following ways:

```
request Event1
request Event2 state
```

9.3.4.2 wait keyword

The `wait` keyword is used to wait for an event. A request must be called before `wait`, and `wait` blocks until the requested event is set. `wait` may proceed immediately if the requested event is a state event and already pending (for example, a phone call is already active). `wait` has one mandatory argument, which is described in Table 9-15 below.

Table 9-15 wait mandatory argument

Argument	Description
Event	The event name.

The `wait` keyword can be used, for example, in the following way:

```
wait Event1
```

9.3.4.3 release keyword

The `release` keyword is used to release an event. Every requested event must be released explicitly when it is not used anymore. `release` has one mandatory argument, which is described in Table 9-16 below.

Table 9-16 release mandatory argument

Argument	Description
Event	The event name.

The `release` keyword can be used, for example, in the following way:

```
release Event1
```

9.3.4.4 set keyword

The `set` keyword is used to set an event. Every set state event must be explicitly unset.

`set` has one mandatory argument and also one optional argument, as described in Table 9-17 below.

Table 9-17 set arguments

Argument	Description
Event	The event name.
State	Optional. If a state is given, sets the state event, otherwise sets an indication event. A state event remains set until it is unset explicitly with the `unset` keyword. An indication event is set only once to every requester and implicitly unset after that.

The `set` keyword can be used, for example, in the following ways:

```
set Event1
set Event2 state
```

9.3.4.5 unset keyword

The `unset` keyword is used to unset a state event. Every set state event must be unset. Indication events cannot be unset. `unset` blocks until everyone who has requested the specified event has released the event.

`unset` has one mandatory argument, which is described in Table 8.18 below.

Table 8.18 unset mandatory argument

Argument	Description
Event	The event name.

The `unset` keyword can be used, for example, in the following way:

```
unset Event1
```

10. Using Test Scripter combiner feature

This chapter specifies Test Scripter Combiner feature of MIN. Test combiner feature is used for running test cases from different test modules and to generate new test cases by combining different test cases possibly from different test modules. Test combiner is controlled with a scripting language for which this document specifies the vocabulary.

10.1 Test combiner feature test case file

The Test Scripter test case file defines the commands to be executed in a test case described with a simple scripting language.

The test case file may contain several test cases for which the description starts with a `[Test]` tag and ends with a `[Endtest]` tag. The test case itself is described with keywords, arguments and argument value pairs between the start and end tag. For example:

```
[Test]
title Create net connection and send sms
timeout 10000
run netmodule net.cfg 4 ini=net.ini
print send sms
run smsmodule sms.cfg 1
[Endtest]
```

The test case is executed sequentially line by line. Some of the keywords may block execution, for example the `complete` keyword waits until the test case completes before the execution proceeds to the next line.

10.2 Setting up the Test Combiner

Test Combiner is configured for use by adding Test Combiner as a test module to MIN with the test case file specified.

10.3 Vocabulary

The Test combining feature vocabulary is composed of keywords and arguments. The keywords are used as the first word in a line of the test case description and they describe the main operation of the test case line. The keywords may have mandatory arguments and also optional arguments. The mandatory arguments are given as values and they have a specified sequence. The optional arguments can be given in any order and they are given with argument-value pairs. The Test Combiner keywords with their arguments are described in the following sections.

10.3.1 General

10.3.1.1 title keyword

The `title` keyword is used to give a verbal description for a test case. The description is placed after the keyword. The title keyword is mandatory for every test case and must be placed as the first keyword in the test case description.

10.3.1.2 timeout keyword

The `timeout` keyword is used to give a timeout value for a test case. The timeout value is given as an argument for the `timeout` keyword as is described in Table 10-19.

Table 10-19 Timeout argument

Argument	Description
Timeout value	The timeout value in milliseconds.

The `timeout` keyword can be used, for example, in the following way (a timeout of 10 seconds):

```
timeout 10000
```

10.3.1.3 print keyword

The `print` keyword can be used to print, for example, progress information to UI. The printed description is placed after the `print` keyword.

10.3.1.4 canceliferror keyword

The `canceliferror` keyword is used to cancel the execution of the remaining test cases if one of the executed test cases has failed. This keyword is normally used to stop the test case execution when some of the test cases are long running.

Table 10-20 canceliferror argument

Argument	Description
`canceliferror`	If this keyword is given and one of the executed test cases has failed, the execution of the remaining test cases is cancelled. This keyword is Test Combiner test case specific.

The example below shows how this keyword can be used. The first test case fails and the ongoing execution of the second test case is cancelled.

```
[Test]
title Simple test case with canceliferror keyword
canceliferror
run testmodule1 myConfig.cfg 1 // test case fails
run testmodule2 mySecondConfig.cfg 2 // long running test case
[Endtest]
```

10.3.1.5 pausecombiner keyword

The `pausecombiner` keyword is used to pause test combiner for a specified time. `pausecombiner` has one mandatory argument, as is described in Table 10-21

The `pausecombiner` keyword stops the test combiner for the specified period of time.

Table 10-21 pausecombiner argument

Argument	Description
Timeout	The timeout for the pause, specified in milliseconds.

The `pause` keyword can be used, for example, in the following way:

```
pausecombiner 10000   // pause for 10 seconds
```

10.3.2 Test Case control

10.3.2.1 run keyword

The `run` keyword is used to start a specified test case. It has several mandatory and optional arguments. The mandatory arguments are described in order in Table 10-22.

Table 10-22 run mandatory arguments

Argument	Description
`testmodule`	The test module name.
`configfile`	The test case configuration file.
Test case number	The test case number to be executed from `configfile`.

The optional arguments are described in Table 10-23. The possible default values, which can be changed with these arguments, are listed in brackets.

Table 10-23 run optional arguments

Argument	Description
`expect`	The expected result (`0 = ENOERR`).
`testid`	Test case identification, which is used by other keywords to identify the test case ().
`Ini`	The initialization file for test module (not supported yet).
`category`	The result category; either normal, leave, panic, exception or timeout.
`timeout`	Test case timeout.
`title`	Test case title. If given, following rules must be held: • if title is gives, test case number is ignored (however still must be provided); • if title contains space chars, then the whole title has to be given between quotation marks (e.g. title="My example with space"); • test case title must not contain quotation mark (“);

	• all normal modules must be 1-base indexed; • if module has more than one test case which match the title, first one will be run; • in master slave environment, if module does not have configuration file, *dummy.cfg* must be given for configfile argument.

The `run` keyword can be used for example in the following way:

```
run netmodule net.cfg 5 testid=test1 expect=3 ini=ini.txt
run netmodule net.cfg -1 testid=test1 title="My test case example"
```

10.3.2.2 cancel keyword

The `cancel` keyword is used to cancel a started test case. The test case is cancelled by immediately killing the thread that executes the test case. The `cancel` keyword has one mandatory argument as described in Table 10-24.

Table 10-24 cancel mandatory arguments

Argument	Description
`testid`	The test ID from the run command.

The `cancel` keyword can be used, for example, in the following way:

```
cancel test1
```

10.3.2.3 pause keyword

The `pause` keyword is used to pause a test case. The test case is paused by stopping the process that executes the test case. The `pause` keyword has one mandatory argument, described in Table 10-25 and one optional argument, described in Table 10-26.

Table 10-25 pause mandatory arguments

Argument	Description
`testid`	The test ID from the run command.

Table 10-26 pause optional arguments

Argument	Description
`Time`	Pause time in milliseconds. After this time, `resume` is called automatically (if not given, `resume` needs to be called explicitly).

The `pause` keyword can be used, for example, in the following way:

```
pause test1 time=10
```

10.3.2.4 resume keyword

The `resume` keyword is used to resume a paused test case. `resume` has one mandatory argument, described in Table 10-27.

Table 10-27 resume mandatory arguments

Argument	Description
`testid`	The test ID from the run command.

The `resume` keyword can be used for, example, in the following way:

```
resume test1
```

10.3.2.5 complete keyword

The `complete` keyword is used to have a started test case wait to complete. It blocks until the test case has finished. `complete` has one mandatory argument, described in Table 10-28.

Table 10-28 complete mandatory arguments

Argument	Description
`testid`	The test ID from the run command.

The `complete` keyword can be used, for example, in the following way:

```
complete test1
```

10.3.2.6 loop keyword

The `loop` keyword is used to repeat a section of the test case file for the specified number of iterations. The section to be repeated is enclosed with the `loop` and `endloop` keywords.

Table 10-29 loop argument

Argument	Description
Loop times	The loop count, that is, the number of times that the loop is executed.
`msec`	(optional) This keyword says that 'Loop times' argument stands for the time in milliseconds during which loop will be looped

The `loop` keyword can be used, for example, in the following way:

```
loop 5
```

```
// execute this 5 times
print pa ra me ter
endloop
```

10.3.2.7 breakloop keyword

The **breakloop** keyword is used to prematurely exit the current loop.

The breakloop keyword can be used, for example, in the following way:

```
loop 5
var exitcond
someclass somemethod someparam
if exitcond
 breakloop
endif
endloop
```

10.3.2.8 endloop keyword

The endloop keyword is used to specify the end of a looped section.

10.3.2.9 If, else and endif keywords

The if, else and endif keywords can be used for conditional execution in the script. The condition for if must be variable or value. Value 0 and string "false" (case insensitive) is interpreted as false, or other values as true. If block must always be closed with endif keyword. Nesting of if-blocks is supported.

```
var v
if v
  print is true
else
  print is false
endif
```

10.3.3 Remote test case control

Test Scripter can control and use slaves when running test cases. A slave can be used for running test cases in master control. Synchronization between test cases can be done using events.

Note: The master cannot see the slave's event system, and neither can slave see the master's event system. Master's and slave's events are visible only to Test Scripter, and therefore it must observe running tests and transfer events to other module(s) if synchronization is needed between master and slave module.

```
[Test]
title Events between master and slave
allocate phone slave
remote slave request event1
remote slave run scripter testcasefile.cfg 1
run scripter testcasefile2.cfg 2
remote slave wait event1
set event2
remote slave release event1
free slave
[Endtest]
```

In this example, Test scripter allocates the slave phone and starts the slave's Test Scripter type of test case. After that a local test case is started, and at some point it starts to wait for `event2`. Test Scripter also starts waiting for `event1` from the slave test module. When the slave test case sets `event1`, Test Scripter sets `event2` for local test case and it can then continue the execution.

10.3.3.1 allocate keyword

The `allocate` keyword is used to allocate a slave, for example for running a test case on a remote phone. It uses Remote Control Protocol (RPC). The slave must always be allocated first before it can be used.

The `allocate` keyword has two mandatory arguments, described in Table 10-30.

Table 10-30 allocate mandatory arguments

Argument	Description
Slave type	The type of the slave. MIN only supports slave `phone`. `phone` indicates that slave phone is also running MIN. Other types must be handled by the slave implementation, i.e. when implementing separate support for external network simulator.
Slave name	A unique name for the slave.

The `allocate` keyword can be used for example in the following way:

```
allocate phone MySlave
```

10.3.3.2 free keyword

Every allocated slave must be freed with `free` when it becomes unused.

The `free` keyword has one mandatory argument, described in Table 10-31.

Table 10-31 free mandatory argument

Argument	Description
slave name	The slave name, the same that was given for `allocate`.

`free` can be used for example in the following way:

```
free MySlave
```

10.3.3.3 var keyword

The `var` keyword declares a variable for the scripter. A variable can be initialized during its declaration. The value of variable can be acquired from another test case with `expect` keyword

and sent to another test case with `sendreceive` keyword. If variable name appears in text used with `print` keyword, its value gets printed.

The `var` keyword has one mandatory argument, described in Table 10-32 and one optional argument, described in Table 10-33.

Table 10-32 var mandatory argument

Argument	Description
variable name	The variable name.

Table 10-33 var optional argument

Argument	Description
variable value	The value which declared variable is to be initialized.

The `var` keyword can be used for example in the following way:

```
var variable2
expect variable2
print variable2
```

10.3.3.4 sendreceive keyword

The `sendreceive` keyword is used in slave script to send a variable value from slave to master. Sending a variable value from master to slave is described in Section 10.3.3.6.

The `sendreceive` keyword has two mandatory arguments, described in Table 10-34.

Table 10-34 sendreceive mandatory arguments

Argument	Description
variable name	The variable name.
variable value	The variable value.

The `sendreceive` keyword can be used for example in the following way:

```
sendreceive variable1=/tmp/file.txt
```

10.3.3.5 expect keyword

The `expect` keyword is used in slave script to expect a variable value from master. Variable must be declared by using `var` keyword. Expecting a variable value from slave to master is described in Section 10.3.3.6.

The `expect` keyword has one mandatory argument, described in Table 10-35.

Table 10-35 expect mandatory argument

Argument	Description
variable name	The name of the variable.

The `expect` keyword can be used for example in the following way:

```
expect variable1
```

10.3.3.6 remote keyword

The `remote` keyword is used to start the execution of a test case in a slave, to request and release events from the slave and sending a variable value to a slave or expecting a variable value from a slave. Other test case controlling for remote test cases is done with the same keywords as for the local test cases.

The `remote` keyword has two mandatory arguments, described in Table 10-36.

Table 10-36 remote mandatory arguments

Argument	Description
Slave name	The slave name, the same that was given for `allocate`.
Command name	The remote command name (supported: `run`, `request`, `wait`, `set`, `unset`, `release`, `expect`, `sendreceive`).

The `remote` keyword can be used for example in the following ways:

```
remote MySlave run netmodule net.cfg 5
remote MySlave request Event1
remote MySlave wait Event1
remote MySlave set Event1
remote MySlave unset Event1
remote MySlave release Event1
remote MySlave expect variable1
remote MySlave sendreceive variable1=/tmp/file.txt
```

The `remote` keyword with `sendreceive` command is used in the master script for sending a variable value to one of the slaves identified by slave name. This remote command supports the same parameters as the keyword described in Section 10.3.3.4.

The `remote` keyword with `expect` command is used in master script for expecting a variable value from one of the slaves identified by slave name. This remote command supports the same parameters as the keyword described in Section 10.3.3.5.

The other supported remote commands are `run`, `request`, `release`, `set`, `unset` and `wait`. They support the same parameters as the same keywords described in Sections 10.3.2.1, 9.3.4.1 and 9.3.4.3.

11. Using LuaScripter for creating scripted test cases

This chapter describes Lua Test Scripter feature of MIN. Lua Test Scripter feature is used for running scripted test cases where the script itself is written in Lua scripting language (www.lua.org).

11.1 Overview of Lua scripting language

Lua is a modern and flexible scripting language designed to be embedded in C or C++. It gives out of the box support for arrays, variables, control structures (if/else, while, for, repeat). It has functions and even threads.

For further details, please visit www.lua.org.

11.2 Lua Scripter test case file

The Lua Scripter uses scripts written in Lua as the test case file. Each test must be defined as a separate function which does not take any parameters and which is prefixed with *"case_"* statement. Test case file may contain several test functions, and some defined global variables as well. User can define its own functions inside of a test case file – every function not prefixed with *"case_"* will not be treated as test case.

The test case result is returned from test case function to the MIN by using return keyword.

Interaction with MIN is achieved by using MIN2Lua API which is covered in this chapter.

An example of a test case file is described below:

```
function case_foo() -- Test case title
            min.print("Hello World");
            return TP_PASSED;
end
```

Test case file may contain definitions. The definitions are global variables and can be defined as follows:

```
-- Begin of define section
MODULE = TestCaseModuleThatCanBeUsedWithLua
-- End of define section

function case_foo() -- Test case title
            min.print(string.format("Using: %s test module",MODULE));
end
```

11.3 Setting up the Lua Scripter

Lua Scripter is configured for use by adding Lua Scripter as a test module to MIN. The test case file is added as a test case file for Lua Scripter. An example:

```
[New_Module]
ModuleName=luascripter
TestCaseFile=script.lua
[End_Module]
```

Example above defines a new module for MIN (should be put in min.conf file) that uses Lua Scripter module with script.lua test case file.

11.4 Lua test class

In order to use user created test code with Lua Scripter it is necessary to use MIN test module which is compliant with Lua: the Lua test class module.

A new Lua test class can be easily created with MIN Test Module Template Wizard that comes with the MIN release

`$createtestmodule`

- Give `luatestclass` as `ModuleType`.
- Enter a name and path for the Lua test class.

A new Lua test class is now created with the given name to the given path. The next step is to create the building block methods to the test class.

To create the building block methods to the Lua test class:

1. Implement the building blocks to the `/<testclassname>/src/<testclassname>Cases.c` file (there is one example method, that can be copy-pasted).
2. Add entry lines for every method to function `ts_get_test_cases (<testclassname>Cases.c)`:
   `ENTRY(*list,"Example",ExampleTest);`
3. Add entry lines for every method to FORWARD DECLARATIONS:
   `LOCAL int ExampleTest( lua_State *l );`
4. Build the test class from `the /<testclassname>` directory by using `build.sh` script. It will automatically copy test class library to `/home/<user name>/.min`.

Now the Lua test class is created. The next step is to implement test cases to test the test case file used by Lua Scripter. An example test case file can be found in `/<testclassname>/Example.lua`. The test case file is constructed using the script language defined in this chapter.

Copy the test case file to /`home/<user name>/.min/`

Now you should be able to execute your test class module by using console UI or EXT INTERFACE.

11.5 General

This chapter describes the MIN2Lua API as well as it gives a general look on the Lua syntax.

11.5.1 Test case result

Test cases defined in Lua scripting language return their results by value (as they are functions). Two macros are provided: TP_PASSED and TP_FAILED.

Example:

```
function case_function() -- This is a title
            Return TP_PASSED;
```

```
end
```

```
function case_function() -- This is a title
            Return TP_FAILED;
end
```

11.5.2 Test case result description

Test case result description is returned as a second value from function. It is recommended but not needed to have proper test case.

Example:

```
function case_function() -- This is a title
            Return TP_FAILED,"Test Case result description";
end
```

11.5.3 Test case title

The title is used to give verbal id to the test case. If test case title is defined it will be seen on the Console UI, if not then function name will be seen instead. It is recommended to add title to each test case.

Test case title is defined in the same line on which function name is written, after "--"-marks. Please note that this -- is a comment in Lua.

```
function case_function() -- This is a title

end
```

The example above shows a test case with title defined, where following example shows test case without title:

```
function case_function()

end
```

Both of them will be shown on the Console UI, which differs from standard MIN Scripter (In standard MIN Scripter test case title is mandatory).

11.5.4 Calling test functions in test cases

Each test case is a method exported from Test Module. First of all it is a must to load Lua Test Class as a Test Module, then you can call each test function as follows:

```
function case_function() -- load test
            foo = min.load("Bar");
            foo.Example();
            min.unload(foo);
            return TP_PASSED;
end
```

Note that the following call:

```
foo.Example();
```

is referring to the test function name given in ENTRY macro. It is not the real test function name.

11.6 MIN2Lua API

Cooperation between MIN and script written in Lua is done by API defined in MIN and exported to the script. All functions are placed in min namespace.

11.6.1 print method

The *print* functionality can be used to print messages on the Console UI.

```
function case_function() -- print test
            min.print("Hello World");
            return TP_PASSED;
end
```

It is also possible to use message formatting, to achieve this one must use built-in Lua *string* functionality called *format*

```
function case_function() -- print test with formatting
            min.print(string.format("Message: %s, id %d",Hello,112));
            return TP_PASSED;
end
```

11.6.2 load method

The *load* functionality is used to load Lua Test Class provided by the end user. As a parameter it takes module name. As a result it returns a handle to the loaded Lua Test Class that can be used to call user defined test cases from the loaded Test Module.

If loading fails then returned value is *nil*

```
function case_function() -- load test
            foo = min.load("Bar");
            if foo == nil then
                        return TP_FAILED;
            end
            foo.Example();
            min.unload(foo);
            return TP_PASSED;
end
```

Example above will load module *Bar.so* (extension is added automatically) from paths specified in *min.conf* file. Module handler is checked then if loading has been correct.

11.6.3 unload method

The *unload* functionality is used to unload loaded Lua Test Class. As a parameter it takes the handle to the loaded Test Module. See example in previous paragraph.

11.6.4 sleep method

The *sleep* functionality is used to pause test case execution for a specified amount of time. As a parameter it takes the time in milliseconds.

Example shows 1s sleep:

```
function case_function() -- sleep example
            min.sleep(10000);
end
```

11.6.5 request method

The *request* functionality is used to request an event. As a parameter it takes the name of the event to be requested and as an optional second parameter the event type which can be StateEvent or IndicationEvent. By default it is StateEvent.

Examples:

```
function case_function() -- request example state event
            min.request("Event1");
end
```

```
function case_function() -- request example state event 2
            min.request("Event1",StateEvent);
end
```

```
function case_function() -- request example indication event
            min.request("Event1",IndicationEvent);
end
```

Note that it is needed to release event after it has been requested.

11.6.6 release method

The *release* functionality is used to release requested event. As a parameter it takes the name of the already requested event.

```
function case_function() -- release example
            min.request("Event1");
            min.release("Event1");
end
```

11.6.7 set method

The *set* functionality is used to set an event. As a parameter it takes the name of the event to be set and as an optional second parameter the event type which can be StateEvent or IndicationEvent. By default the type is IndicationEvent.

Examples:

```
function case_function() -- set example indication event
            min.set("Event1");
end
```

```
function case_function() -- set example state event 2
            min.set("Event1",StateEvent);
end
```

```
function case_function() -- set example indication event
            min.set("Event1",IndicationEvent);
end
```

11.6.8 unset method

The *unset* functionality is used to unset a **state** event that has been set before. As a parameter it takes the event name.

Example:

```
function case_function() -- unset example
            min.set("Event1",StateEvent);
            min.unset("Event1");
end
```

11.6.9 wait method

The *wait* functionality is used for waiting for requested event. Event might be requested before wait is used, but this is not a must. If event has not been requested it will be requested and released in wait function. As a parameters wait takes event name and optional event type which can be StateEvent or IndicationEvent. By default it is IndicationEvent.

Example:

```
function case_function() -- wait example state event
            min.wait("Event1");
end
```

```
function case_function() -- wait example 2 state event
            min.request("Event1");
            min.wait("Event1");
            min.release("Event1");
end
```

```
function case_function() -- wait example state event
            min.wait("Event1",IndicationEvent);
end
```

```
function case_function() -- wait example 2 state event
            min.request("Event1",IndicationEvent);
            min.wait("Event1",IndicationEvent);
```

```
            min.release("Event1",IndicationEvent);
end
```

Note that it is a must to wait for an event of the type the event has been requested.

11.6.10 run method

The *run* functionality is used to run a test case from Test Module in a combiner mode. Test Module can be every MIN test module. Run functionality takes several parameters, it is a must to specify at least module name and test case (as an id or test case name).

As a result it returns test case execution.

It is not necessary to provide configuration file if not needed (as it is in MIN Scripter).

Examples:

```
function case_function() -- run example
            ret = min.run("TestModule",1);
end
```

Example above shows running test case of id 1 from given Test Module

```
function case_function() -- run example 2
            ret = min.run("TestModule","config.cfg",1);
end
```

Example above shows running test case of id 1 from given Test Module, with specified configuration file

```
function case_function() -- run example 3
            ret = min.run("TestModule","config.cfg","Example case");
end
```

Example above shows running test case of given title, from given Test Module, with specified configuration file

```
function case_function() -- run example 4
            ret = min.run("TestModule","Example case");
end
```

Example above shows running test case of given title, from given Test Module

11.6.11 slave_allocate method

The *slave_allocate* functionality allocates slave device which is needed for remote test case execution. As parameters it takes slave type and slave name. It returns a handle to the slave which is then used for calling methods on slave.

Available slave types are: SlaveTypePhone and SlaveTypeTablet

Example:

```
function case_function() - slave_allocate example
            slavehnd = min.slave_allocate(SlaveTypePhone,"Slave1");
end
```

Note that the allocated slave must be freed.

11.6.12 slave_free method

The *slave_free* functionality is used to free allocated slave device. As a parameter it takes handler to the slave.

Example:

```
function case_function() -- run example
            slavehnd = min.slave_allocate(SlaveTypePhone,"Slave1");
            min.slave_free(slavehnd);
end
```

11.6.13 Remote test case execution

It is possible to execute functionalities on the slave device. In order to do this one must use slave handler:

```
function case_function() -- remote example
            slavehnd = min.slave_allocate(SlaveTypePhone,"Slave1");
            slavehnd:request("Event1");
            min.slave_free(slavehnd);
end
```

```
function case_function() -- remote example 2
            slavehnd = min.slave_allocate(SlaveTypePhone,"Slave1");
            slavehnd.request(slavehnd,"Event1");
            min.slave_free(slavehnd);
end
```

Note that the first parameter is always slave handler. You must specify it when using '.' Or use ':' because then it is added automatically.

11.6.13.1 request method

Method *request* is used to request an event from slave.

```
function case_function() -- remote example
            slavehnd = min.slave_allocate(SlaveTypePhone,"Slave1");
            slavehnd:request( "Event1");
            min.slave_free(slavehnd);
end
```

11.6.13.2 release method

Method *release* is used to release an event.

```
function case_function() -- remote example
            slavehnd = min.slave_allocate(SlaveTypePhone,"Slave1");
            slavehnd:release( "Event1");
            min.slave_free(slavehnd);
end
```

11.6.13.3 run method

Method *run* is used to run a test case on a slave device. Syntax is the same as for min.run functionality.

```
function case_function() - remote run example
            slavehnd = min.slave_allocate(SlaveTypePhone,"Slave1");
            slavehnd:run("TestModule",1);
            min.slave_free(slavehnd);
end
```

11.6.13.4 expect method

Method *expect* is used to wait for variable that will be passed from slave. As a parameter it takes name of the variable under which it has been send.

```
function case_function() - remote expect example
            slavehnd = min.slave_allocate(SlaveTypePhone,"Slave1");
            a = slavehnd:expect("foo");
            min.slave_free(slavehnd);
end
```

11.6.13.5 send method

Method *send* is used to send variable through external interface. As a parameter it takes the name under which it will be sent and the variable (number or string).

```
function case_function() - remote expect example
            slavehnd = min.slave_allocate(SlaveTypePhone,"Slave1");
            a = 5;
            slavehnd:send("foo",a);
            min.slave_free(slavehnd);
end
```

12. Python interpreter module

When using MIN, it is also possible to use python scripting language to create test case logic. MIN can treat specific Python functions as test cases, python extension module is provided that makes it possible to use MIN-specific features (like events or ability to combine test cases from other test modules).

12.1 Introduction

The script language provided by "scripter" test module has several limitations. If it is needed to implement more sophisticated logic, one can use functions written in python as test cases. In that case, architecture of system is as presented on Figure 12.1.

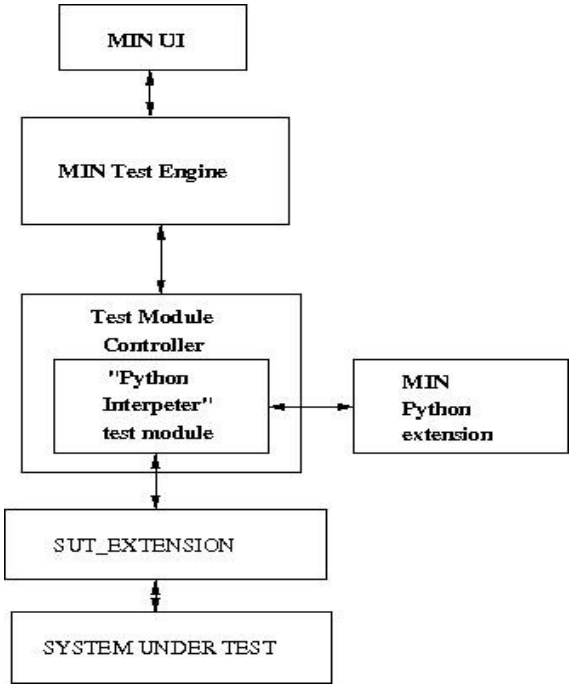

Figure 12.1 Key elements in Python test case execution.

"Python Interpreter" test module and "MIN" Python extension are provided with MIN release. Python Interpreter test module is seen by system like any other test module. MIN Python extension is extension module that is imported by python script that serves as test case file (this will be explained in more detail later). SUT_EXTENSION is another extension module that python script needs to import. This one, however, is implemented by tester (based on template, also provided with MIN release). This extension serves as an interface to actual system under test.

12.2 Python test module usage

12.2.1 Configuration

To use python scripted test cases, user needs to configure python interpreter module like any other module, by appropriate entries in MIN configuration files, like in the following example (name of test case file is an example of course):

```
[New_Module]
ModuleName = pythonscripter
TestCaseFile = pythonscript.py
[End_Module]
```

Python scripter module reads script given as test case file, and uses functions with names starting with "case_" as test cases. More information on syntax of test case file will be given later. Then, python module is used as any other test module of MIN.

It should be noted, that during the execution, min modifies PYTHONPATH by appending definition of directories used to store files necessary for min python module. Appended directories are defined in min configuration file, in "ModSearchPath" entries. Variable is restored to original value after closing MIN.

12.3 Python test case definition

12.3.1 Test case file details

Generally, just the Python script might be not enough to conduct testing. The simplest way to access the tested functionality might be from the C code, not Python code. One way to facilitate that is to use Python extension written in C. You are provided with template for that extension. This template contains a skeleton of all the functions that "Python extension" library has to contain, and stubs for example test functions. Then you can import a compiled library to you Python script, in which you can implement the logic of the test.

Apart from the simplest functionality – to load and execute a test case, MIN provides some additional features in the "test module API". Equivalent of this "API" for Python test cases is "MIN Python extension library". When this library is imported to a Python script, it gives you access to MIN functionalities such as printing to the MIN console UI, executing test cases from other modules and synchronizing test execution by events. Detailed reference of those functions will be provided later.

As was mentioned before, "Python scripter" module will read the specified Python script ("*.py"), and consider each function with name starting with "case_". If function has "docstring", contents of the docstring will be used as a title of test case by MIN, otherwise the function name will be used. If test case function returns "0", test case will be considered as passed by MIN any other value will be considered "failure" (if function returns "NULL", it is considered "crashed"). If you do not define return value for a test case function, the result will be undefined (most likely – failed).

Table below shows example python scripter test case definition:

```
import min_ext
import sut_module1


def case_ms1():
        """ping"""
        min_ext.Allocate_slave("phone_b")
        min_ext.Request_remote_event("phone_b","ping",)
        min_ext.Run_remote_case("phone_b","scripter","slave.scr",1)
        retval = min_ext.Wait_event("ping")
        if (retval == 0):
                retval = sut_module1.Do_stuff()
        min_ext.Release_remote_event("phone_b","ping")
        min_ext.Free_slave("phone_b")
        return retval
```

12.3.2 MIN python extension library reference

MIN python extension library provides users with set of functions that enable them to use the same features as in other types of modules. Functions are described below (note that arguments are specified in order in which they should appear in function calls):

Function	**Description**
`Print_to_cui`	Prints specified message to MIN console UI
Parameters	
`(mandatory) String`	Contains communicate to be printed
Return value	
`Int`	Always 0
Example	
min_ext.Print_to_cui("Printout text")	

Function	**Description**
`Set_indication_event`	Sets MIN event of type "indication" and specified name
Parameters	
`(mandatory) String`	Name of event to be set
Return value	
`int`	Always 0.
Example	

min_ext.Set_indication_event("MyEventInd")

Function	**Description**
`Set_state_event`	Sets MIN event of type "state" and specified name
Parameters	
`(mandatory) String`	Name of event to be set
Return value	
`Int`	Always 0
Example	
min_ext.Set_state_event("MyEventState")	

Function	**Description**
`Unset_event`	Unsets MIN event of type "state" and specified name. Note, that only events of type "state" can be unset.
Parameters	
`(mandatory) String`	Name of event to be unset
Return value	
`Int`	Always 0
Example	
min_ext.Unset_event("MyStateEvent")	

Function	Description
`Request_event`	Requests event. If test case will be waiting for some event, it first needs to request
Parameters	
`(mandatory) String`	Name of event to be requested
`(optional) "state"`	Keyword that should be specified if event is of type "state". If it is omitted system will assume that requested event is of type "indication"
Return value	
`Int`	Always 0
Example	
min_ext.Request_event("MyEventInd") or: min_ext.Request_event("MyEventState,"state")	

Function	Description
`Release_event`	Releases the previously requested event when it is not needed anymore.
Parameters	
`(mandatory) String`	Name of event to be requested
Return value	
`Int`	Always 0
Example	
min_ext.Release_event("MyEvent")	

Function	Description
`Wait_event`	Waits on previously requested event
Parameters	
`(mandatory) String`	Name of event
Return value	
`Int`	0 if event was properly received, other value in case of error
Example	
min_ext.Wait_event("MyEvent")	

Function	Description
`Complete_case`	Executes test case from other MIN test module. Function doesn't return until started test case finishes execution.
Parameters	
`(mandatory) String`	Name of test module
`(optional)String`	Name of test case file
`(mandatory) String`	Name (title) of test case
Return value	
`Int`	Result of test case (see Test Module API description)
Example	
min_ext.Complete_case("min_hardcoded_module","testcase_title") or min_ext.Complete_case("min_other_module","testcase_file","testcase_title")	

Function	Description
`Start_case`	Starts asynchronous execution of test case from another module. Note that Python test case won't end until this test case finishes, but result of test case is not provided. It is possible to synchronize with this test case via events.
Parameters	
`(mandatory) String`	Name of test module
`(optional)String`	Name of test case file
`(mandatory) String`	Name (title) of test case
Return value	
`int`	Always 0
Example	
min_ext.Start_case("min_hardcoded_module","testcase_title") or min_ext.Start_case("min_other_module","testcase_file","testcase_title")	

Function	Description
Create_logger	Creates MIN logger structure (Keep in mind, that the logger created in test case has to be destroyed in the same case, otherwise there will be resource leak)
Parameters	
(mandatory) String	Name of directory to store log
(Mandatory)String	Name of file
(mandatory) String	Format (can be "html" or "txt")
Return value	
Int	Handle to MIN logger structure (used later to log and destroy logger)
Example	
My_logger = min_ext.Create_logger("/tmp","logfile.log","txt")	

Function	Description
Log	Writes specified string to log.
Parameters	
(mandatory) int	Handle to logger (returned by Create_log))
(optional)char	Style (can be b/B for bold, u/U for underline and i/I for italic – other values are ignored.). Styles work of course only in case of html logger.
(mandatory) String	Text to be written
Return value	
Int	Always 0
Example	
min_ext.Log(My_logger,"b","log this text")	

Function	Description
Destroy_logger	Destroys MIN logger structures, required to free system resources allocated by Create_logger.
Parameters	
Int	Handle to logger returned by Create_logger
Return value	
Int	Always 0

Example
min_ext.Destroy_logger(My_logger)

Function	**Description**
`Allocate_slave`	Allocates slave device, in external controlled master/slave scenarios
Parameters	
`(mandatory) String`	Name of slave device
Return value	
`Int`	0 if operation was successful, error value otherwise
Example	
min_ext.Allocate_slave("other_device")	

Function	**Description**
`Free_slave`	Releases slave device in external controlled master/slave scenarios
Parameters	
`(mandatory) String`	Name of slave device
Return value	
`Int`	0 if operation was successful, error value otherwise
Example	
min_ext.Free_slave("the_other_device")	

Function	**Description**
`Request_remote_event`	Requests event from slave device in externally controlled master/slave scenarios
Parameters	
`(mandatory) String`	Name of slave device
`(mandatory) String`	Name of the event
`(optional) "State"`	Indicates if the requested event is of type state. If it is specified as a string different than "State" (case insensitive), or not specified at all, event will be considered indication event.
Return value	
`Int`	0 if operation was successful, error value otherwise

Comments
After the event is requested from remote device, master test case can wait on it in the same way it would wait on local event.
Example
min_ext.Request_remote_event("other_device",”some_indication_event”) or min_ext.Request_remote_event(“the_other_device”,”some_state_event”,”state”)

Function	**Description**
`Release_remote_event`	Releases previously requested remote event.
Parameters	
`(mandatory) String`	Name of slave device
`(mandatory) String`	Name of event
Return value	
`Int`	0 if operation was successful, error value otherwise
Comments	
Function fails in case of problems in master/slave communication or faulty argument specification	
Example	
min_ext.Release_remote_event("other_device",”event”)	

Function	Description
`Run_remote_case`	Executes Test case in previously allocated slave device.
Parameters	
`(mandatory) String`	Name of slave device
`(mandatory) String`	Name of module
`(optional) String`	Name of test case file (if applicable to given module)
`(mandatory) Int`	Number of test case.
Return value	
`Int`	0 if operation was successful, error value otherwise
Comments	
Function is considered successful if remote test case was started. After that, only available method of communication with slave test case is by events. Result of remote test case will not be provided. However, python test case will not finish, until slave test case finishes (or external controller timeout expires).	
Example	
min_ext.Run_remote_case("other_device",”hardcoded_min_module”,1) or min_ext.Run_remote_case(“another_device”,”other_min_module”,”testcase_file”,3)	

12.4 PyUnit cases wrapper

MIN release package provides also Python template/wrapper for PyUnit tests. If you have already developed tests following the PyUnit format, you can also execute those under control of MIN. To do this:

- Edit *minwrap.py* template (placed under /min_py_module/min_ext/), replacing all occurrences of PyUnit_module_xxx with the name of the module containing PyUnit test cases. For convenience, save the file with another name, in directory specified in “PYTHONPATH” environment variable.
- Configure *py_moduleTestModule* in your *min.conf*, giving file that you saved in previous point as a test case file.
- MIN will now see your PyUnit test cases as its own and will be able to execute those.

13. Using MIN Logger for logging purposes

This chapter contains the MIN Logger API description for guidance on how to use MIN Logger for logging purposes.

The purpose of MIN Logger is to get information from the modules in order to write different log files or to send information via Linux Syslog system to standard log output file. Logging to Null Output is also available from MIN Logger.

Figure 13-10 shows the basic architecture of MIN Logger and how it is involved in MIN. MIN Engine uses MIN Logger for Test Module Controller and Test Module logs. MIN Logger can be used when logging from test modules.

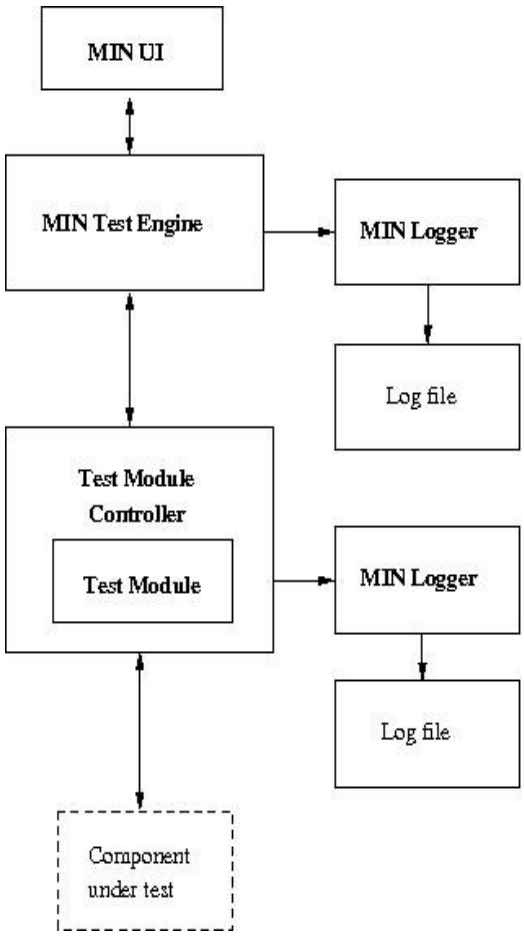

Figure 13-10 MIN Logger

13.1 MIN Logger API

MIN Logger is divided into the following logger structures:

- `MinLogger`

The purpose of `MinLogger` is to get log or data information and send that forward generating the required form.

- `MinTxtLogger`, `MinHtmlLogger`, and `MinDataLogger`

  The log or data information may be generated to different forms. The common way is to log information to a basic text file. There is also a possibility to generate a log file in HTML format that is in some cases more readable.

- `MinLoggerFileOutput`, `MinLoggerNullOutput` and `MinLoggerSyslogOutput`

  The purpose of the output methods is to write log or data information to a file or Linux Syslog output. Here is also output method for null output.

13.1.1 MinLogger API

The methods of MIN Logger are listed and explained below:

The main methods of `MinLogger` are:

- `mnl_create` for creating MIN Logger "object" (see Table 13-37).
- `mnl_destroy` for destroying MIN Logger instance (see Table 13-38).
- `mnl_log` for logging message (see Table 13-39).
- `mnl_write_delimiter` for writing delimiter to log (see Table 13-40).
- `mnl_write_own_delimiter` for writing user defined delimiter to log (see Table 13-41).
- `mnl_output_type` for get information about used output plug-in (see Table 13-42).
- `mnl_logger_type` for get information which logs are created (see Table 13-43).

Table 13-37 mnl_create for creating MIN Logger

Method	**Description**
`mnl_create`	Creates MIN Logger "object".
Parameters	
`const TSChar* path`	Output directory.
`const TSChar* file`	Output filename.
`unsigned int loggertype`	Type of the logger that is in use.
`unsigned int output`	Output plug-in type number.
`TSBool overwrite`	Overwrite file if exists flag.
`TSBool withtimestamp`	Add timestamp flag.
`TSBool withlinebreak`	Add line break flag.
`TSBool witheventranking`	Do event ranking flag.
`TSBool pididtologfile`	Process ID to log file flag.
`TSBool createlogdir`	Create output directory if not exists flag.
`unsigned int staticbuffersize`	Size of the static buffer.

`TSBool unicode`	Unicode flag.
Return value	
`MinLogger*`	Pointer to created MIN Logger data structure.

Table 13-38 mnl_destroy for destroying existing MIN Logger instance

Method	**Description**
`mnl_destroy`	Destroys MIN Logger instance.
Parameters	
`MinLogger** mnl`	The pointer reference to MinLogger data structure.
Return value	
`void`	None.

Table 13-39 mnl_log for write log messages

Method	**Description**
`mnl_log`	Writes delimiter to log.
Parameters	
`MinLogger* mnl`	Pointer to the MinLogger instance to be used.
`TSStyle style`	The style to be used.
`const TSChar* format`	Format of the message to be logged.
`...`	Extra parameters, according to the message format.
Return value	
`int`	Return error code 0 if logging operation completed successfully, else error code -1 if operation failed.

Table 13-40 mnl_write_delimiter for write delimiter to log

Method	**Description**
`mnl_write_delimiter`	Writes delimiter to log.
Parameters	
`MinLogger* mnl`	Pointer to MIN Logger instance.
Return value	
`void`	None.

Table 13-41 mnl_write_own_delimiter for write user defined delimiter

Method	Description
`mnl_write_own_delimiter`	Writes user defined delimiter to log
Parameters	
`MinLogger* mnl`	Pointer to MIN Logger instance.
`const TSChar`	Character used as a delimiter.
`unsigned int t`	Number of character in the delimiter string.
Return value	
`void`	None.

Table 13-42 mnl_output_type for get output plug-in information

Method	Description
`mnl_output_type`	Returns the information which output plug-in is used with logger.
Parameters	
`MinLogger* mnl`	Pointer to MIN Logger instance.
Return value	
`unsigned int`	"OR" modified mask of output plug-in types that were created for this logger.

Table 13-43 mnl_logger_type for get information about created logs

Method	Description
`mnl_logger_type`	Returns the information which logs are created.
Parameters	
`MinLogger* mnl`	Pointer to MIN Logger instance.
Return value	
`unsigned int`	"OR" modified mask of log typed plug-in types that were created.

13.1.2 Use of MIN Logger

There are three logging output variations: file output, null output and Syslog output. It is also available three logging format types: *normal text*, *html* and *data* formats. When be taken logging system for use then it is needed to define logger output and type by mnl_create() method. Other

parameters are consists target logging directory, filename, additional information options and used buffer size definitions. If logger creation be completed successfully then method returns pointer to created logger data structure. If creation failed then returned pointer is INITPTR value.

```
MinLogger* min_logger = mnl_create( "/temp/logs/"
                                    , "min_testing.log"
                                     , ESHtml
                                    , ESFile
                                    , ESTrue
                                    , ESTrue
                                    , ESTrue
                                    , ESFalse
                                    , ESFalse
                                     , ESFalse
                                     , 1000
                                     , ESFalse );
```

Example 1. Use of mnl_create( ) method

Writing to logging file, null output or Syslog file is executed by mnl_log() method and it is possible to use text styles (e.g. bold, underline or "remark" additions) and "sprintf" style string formatting with mnl_log() method. This method returns error code if log writing failed by some reason.

```
int retval = mnl_log( min_logger
                     , ESBold
                    , "Process ID = %s. Current number = %d", pid, data );
```

Example 2. Use of mnl_log( ) method

It is also possible to add pre-defined or own specific delimiter character into logging file or output.

```
mnl_write_delimiter( min_logger );

mnl_write_own_delimiter( min_logger, ':', 1 );
```

Example 3. Use of mnl_write_delimiter( ) and mnl_write_own_delimiter( ) methods

MIN Logger API includes also getting methods to check used logger output plug-in and types.

```
unsigned int output_types = mnl_output_type( min_logger );

unsigned int logger_types = mnl_logger_type( min_logger );
```

Example 4. Use of mnl_output_typer( ) and mnl_logger_type( ) methods

Created logger data structure will be deleted by mnl_destroy() method after logging output not needed any more. If deletion be completed successfully then used logger pointer is set with INITPTR value.

```
mnl_destroy( &min_logger );
```

Example 5. Use of mnl_destroy( ) method

For convenience MIN provides several macros that can be used for logging purposes. Those macros are MIN_XXX(char *format, ...) where XXX is one of: TRACE, DEBUG, NOTICE, INFO, WARN, ERROR and FATAL. Those names indicates importance level for logged message (FATAL is the highest, TRACE is the lowest (the most verbose)).

Before one start using those macros the min_log_open(char *component, int level) function must be called. The supplied component name is then used in log messages to indicate the source of logged information, the level parameter is present only for binary compatibility and can be whatever value, usually 0.

When one is done with logger usage min_log_close() function should be called.

Typical usage looks like:

```
min_log_open("Component Name",0);
MIN_INFO("Log message number %d",1);
MIN_DEBUG("Debug");
min_log_close();
```

14. Using MIN Event System for test cases synchronization

This chapter describes the use of MIN Event System for test cases synchronization.

14.1 Event interface for the test modules

The interface for event system is defined in file `min_test_event_if.h` and has three interface functions `min_event_create(), min_event_destroy ()` and `Event()`. For more information see section 14.2.

14.1.1 State events

The state events are used to indicate that some specific state is active or inactive. State events are cached, that is, their state is stored in Test Engine. This means that when a test case requests a state event, the state of the event is checked, and if it is active, the event is set immediately and it remains set as long as it is unset by the one who has set it. If the state is inactive, the event is set for the requesting client immediately after the event is set.

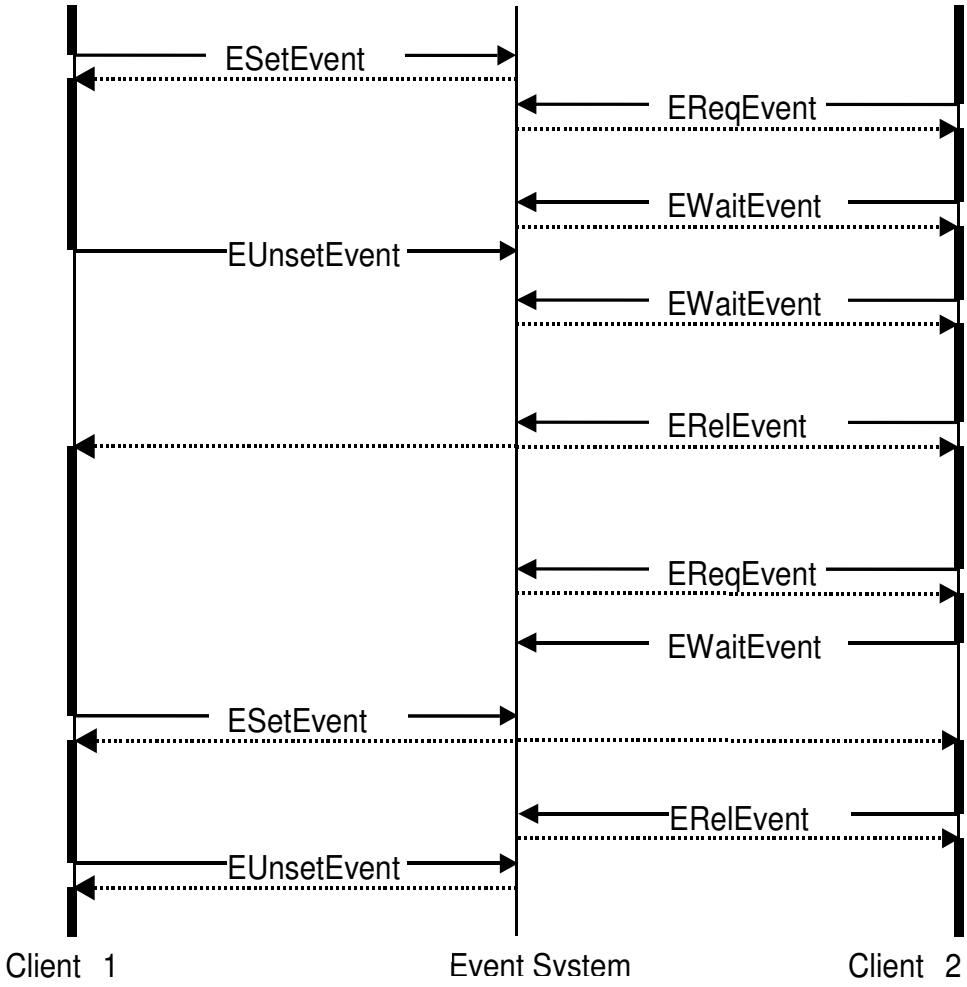

Figure 14-11 State event handling

In Figure 14-11, first client 1 sets a specific state event. Then client 2 requests the event and goes waiting for it. The waiting returns immediately because the event is already set. After some processing, client 2 checks that the event is still active and after waiting returns immediately and proceeds with processing of its tasks. In the meanwhile, client 1 unsets the event, but the unset blocks because client 2 has requested the event. After client 2 releases the event, the unset returns to client 1. Client 2 starts processing again and requests the event again and goes waiting for it. Because the event is unset, the waiting blocks until client 1 sets the event and then the waiting returns to client 2. Then client 2 does its tasks and finally releases the event and client 1 unsets the event.

14.1.2 Indication events

The indication events are used to send an event that a specific occasion has happened. Indications are not cached, so in order to receive an indication, it must have been requested before the indication event is set.

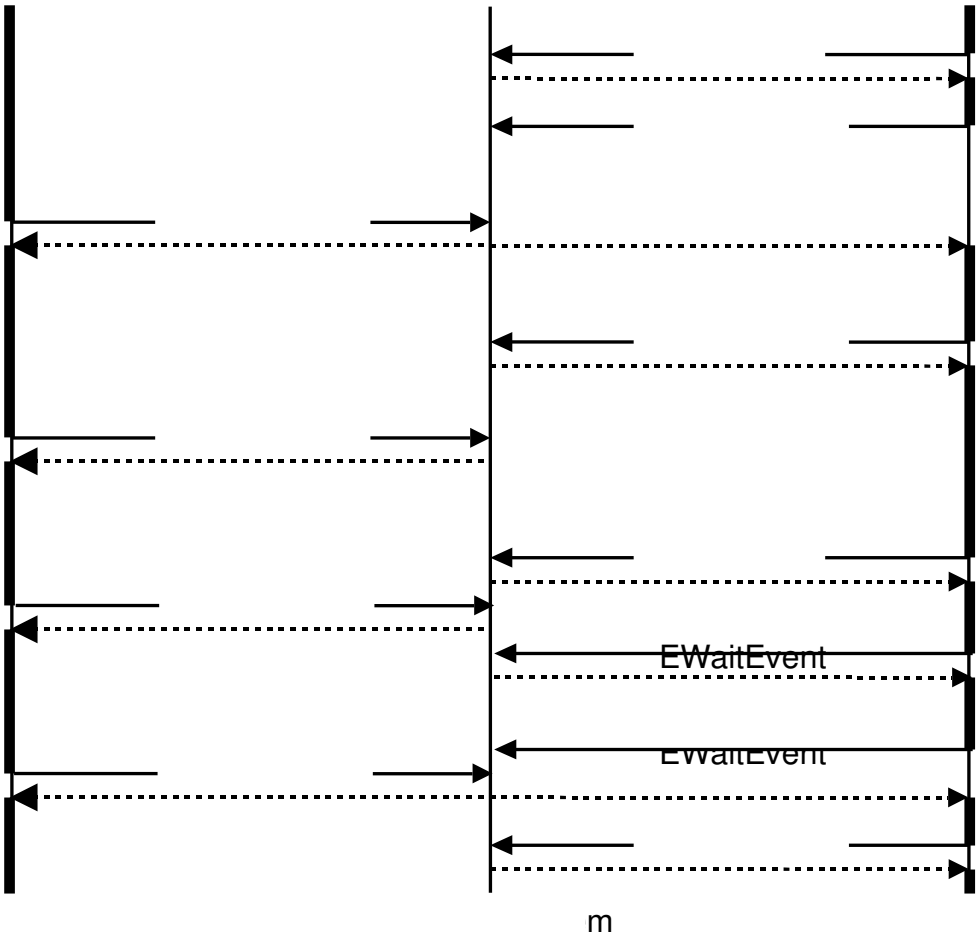

Figure 14-12 Indication event handling

Figure 14-12 depicts the indication event handling. Client 1 sets a specific indication event always after some specific occasion. After client 2 starts, it requests the indication event and goes waiting for it. After client 1 sets the event, the waiting returns to client 2 and it proceeds the processing. After client 2 has done its tasks, it releases the event. Then client 1 sets the event again, but because client 2 has no pending event request, it does not get the event. Next, client 2 requests the event again. Client 1 then sets the event and when client 2 goes waiting for the event, it returns immediately because the event is set. After that, client 2 goes waiting for the event again and the waiting returns to client 2 when client 1 sets the event again. Finally, client 2 releases the event.

14.2 MIN Event System usage

MIN Event System can be used for various purposes and in different ways. As an example, a state event can be used to indicate when a phone call is active: a state event is set right after the call is connected and unset just before the call is deactivated. Some other test case may request the state event and wait until it is set, then send an SMS and release the state event after that. It is guaranteed that the state remains until the requester releases the event, which means that the called `unset` blocks until the release is called.

An indication event can be used for example to indicate that an HTTP packet is received: An indication event is set every time when an HTTP packet has been received. Some other test case

may then e.g. wait for the indication event and then start heavy loading of the system to interfere with the Web page receiving.

The following is a sample code for setting a state event:

```
minEventIf *s_event = min_event_create ("TestModuleState1", EState);
event->SetType (s_event, ESetEvent);
Event (s_event);
```

The following is a sample code for unsetting a state event:

```
event->SetType (s_event, EUnsetEvent);
Event (s_event);
```

The following is a sample code for setting an indication event:

```
minEventIf *i_event = min_event_create ("TestModuleIndication1", EIndication);
event->SetType (i_event, ESetEvent);
Event (i_event);
```

The following is a sample code for requesting, waiting and releasing a state event:

```
minEventIf *s_event = min_event_create ("TestModuleState1", EState);
s_event->SetType (s_event, EReqEvent);
Event (s_event);
s_event->SetType (s_event, EWaitEvent);
Event (s_event);
/* Do something */
s_event->SetType(s_event, ERelEvent);
Event (s_event);
```

The following is a sample code for requesting, waiting and releasing an indication event:

```
minEventIf *i_event = min_event_create ("TestModuleIndication1", EIndication);
i_event->SetType (i_event, EReqEvent);
Event (i_event);
i_event->SetType (i_event, EWaitEvent);
Event (i_event);
/* Do something */
i_event->SetType (i_event, ERelEvent);
Event (i_event);
```

15. MIN Text interface

This chapter describes the MIN Text interface.

15.1 Critical information

MIN Text interface is designed to be used, to handle C strings allocated on the heap, without worrying about the amount of memory needed to be allocated. Text interface can grow or shrink automatically.

MIN Text interface is always allocated on the heap by using tx_create method. It is required by the end user to free all allocated resource by calling tx_destroy on it.

15.2 Usage example

```
Text *string = tx_create("MIN");     // allocate Text with "MIN" as initializer
tx_c_append(" is great");            // modification of Text data
tx_destroy(&string);     // deallocate Text, and buffer used to hold string data
```

15.3 MIN Text interface API description

Text *tx_create (const char * txt)

Creates Text structure and initializes it with C-string passed as a parameter.

void tx_destroy (Text ** txt);

Destroys Text structure allocated with tx_create.

void tx_append (Text * dest, const Text * src)

Appends to one Text interface another.

void tx_copy (Text * dest, const Text * src)

Copies content of one Text to another.

void tx_c_append (Text * dest, const char *src)

Appends C string to Text interface.

void tx_prepend (Text * src, Text * dest)

Prepends to one Text interface another.

void tx_c_prepend (Text * src, const char * dest);

Prepends to Text interface c string.

void tx_c_copy (Text * dest, const char *src);

Copies content of c string to Text interface.

char *tx_get_buf (Text * txt)

Getter for C string representation of data held in Text interface. Returns the copy of data held by the Text interface.

char *tx_share_buf (Text * txt)

Getter for adress of data held in Text interface. Returns pointer to the data held by Text interface.

void tx_back_trim (Text * txt, const char *chars)

Removes specified set of characters from the end of Text

char tx_at (Text * txt, unsigned int index)

Gives character from the Text that is at given position. If index exceeds the length of the data NULL byte is returned.

void tx_int_append(Text * dest,const char *options, int src)

Appends to Text interface an integer. Options parameter indicates the modifiers (the same as for printf) [flags][width][.precision][length]. They specify the minimal number of digits the integer will have, the precision, and so on.. for details see man printf.

16.Test interference

The following chapter describes functionality and usage of MIN's Test Interference.

16.1 Overview

Test interference allows programmers to execute test cases in system under stress – for some components it might be useful to see, how the code behaves if the CPU is busy, when other processes are eating up memory and so on. To simulate system load, MIN uses tools contained in sp-stress package, which is a part of Maemo SDK. Tools can simulate: cpu load, memory usage and IO operations. Test interference is usable in "coded" ("hardcoded", MINUnit and so on), as well as in scripted test cases. Tools from sp-stress package are executed in process running in parallel to test case.

16.2 Prerequisites

To use test interference, it is mandatory to have sp-stress package installed. Lack of the package will make scripted cases, that use test interference, invalid. In case of "coded" test cases – user needs to check explicitly if creation of test interference succeeded.

16.3 Test Interference API

MIN provides a set of C functions, that make it possible to use test interference functionality in the same way as other MIN facilities.

16.3.1 ti_start_interference

Function that creates test interference "instance" and starts the interference process. By "instance" we understand `testInterference` structure. Pointer to this structure is returned by `ti_start_interference` function, and is used for manipulating created interference process (pausing/resuming and stopping). When function returns, test interference process is started. As argument, function takes enumerator `TInterferenceType`, and value of load. `TinterferenceType` enumerator has the following values:

```
typedef enum {
        ECpuLoad,
        EMemLoad,
        EIOLoad
} TInterferenceType;
```

Enumerator (along with all test interference functions) is available if file includes the header `min_test_interference.h`.

Function has the following prototype:

```
testInterference   *ti_start_interference (TInterferenceType aType,
                                           int aLoadValue
                                          );
```

and should be used in following way:

```
testInterference* disturbance;
disturbance = ti_start_interference(ECpuLoad, 75);
```

In case of `EcpuLoad`, load value holds amount of CPU time that should be taken, in percent. In case of `EmemLoad`, `aLoadValue` holds amount of memory to be taken, in megabytes. In case of `IOLoad` (which performs disk IO operations), this value is ignored. It should be also noted, that if it is not possible to create test interference (for any reason, including missing sp-stress package), function will return `NULL`. More detailed problem information can then be found in syslog.

16.3.2 ti_start_interference_timed

If there is need to make more complicated interference scenario (for example, to mimic "normal" device usage to some extent), `ti_start_interference_timed` function might be used. This causes generated disturbance to start and stop in specified time intervals, until stopped permanently. Function has the following prototype:

```
testInterference   *ti_start_interference_timed (TInterferenceType aType,
                                                 int aIdleTime,
                                                 int aBusyTime,
                                                 int aLoadValue
                                                );
```

And should be used in the following way:

```
testInterference* disturbance;
disturbance = ti_start_interference_timed(EMemLoad,10000,10000,99);
```

This will cause interference process to take 99 megabytes of memory for 10000 milliseconds(10 seconds), then release it for 10 seconds, and repeat cycle until interference is stopped. `aIdleTime` argument specifies the time during which interference process is stopped. `aBusyTime` holds amount of time the interference is actually causing the disturbance. Both time values are specified in milliseconds. Return value of function follows the same rules as `ti_start_interference`.

16.3.3 ti_pause_interference

Function is used to pause test interference process at any moment. It has the following prototype:

```
void ti_pause_interference(testInterference* aInterference);
```

and can be used in the following way:

```
ti_pause_interference(disturbance);
```

`aInterference` argument is pointer to `testInterference` structure, returned by `ti_start_interference` or `ti_start_interference_timed` function.

16.3.4 ti_resume_interference

Function is used to resume previously paused interference process. If the process was not paused, call is ignored.

Function has the following prototype:

```
void ti_resume_interference(testInterference* aInterference);
```

and can be used in the following way:

```
ti_pause_interference(disturbance);
```

`aInterference` argument is pointer to `testInterference` structure.

16.3.5 ti_stop_interference

Function used to stop and destroy previously created `testInterference`. Function has the following prototype:

```
void ti_stop_interference(testInterference* aInterference);
```

and can be used in the following way:

```
ti_stop_interference(disturbance);
```

`aInterference` argument is a pointer to `testInterference` structure, and after function returns, it will be set to `NULL`. Interference process will be killed. Memory taken for test interference data will be freed.

16.4 Using test interference in scripted test cases.

Test Interference can be also used in scripted test cases, by scripter commands. New keyword, `testinterference` is used for that. Starting interference is done in the following way:

```
testinterference name command type value idle_time busy_time
```

testinterference – mandatory keyword

name – identifier of created testinterference

command – can have two values, "start" or "stop"

type – can have the following values: "cpuload", "memload" and "ioload". Determines type of started interference.

value – numeric parameter, for cpuload determines amount of taken CPU time in percent. For memload it holds amount of taken memory in megabytes. For ioload value is ignored.

idle_time – numeric parameter, specifies time in cycle when disturbance is stopped (in milliseconds). Giving it value 0 is equal to starting constant disturbance.

busy_time – numeric parameter, time the disturbance is active during cycle.

All numeric parameters are mandatory (due to way parameters are handled in scripter).

Failure to call "stop" command for all "started" interferences in test case will make the case invalid.

17.Compiling C++ test modules

It is possible to use C++ inside MIN test modules. Currently (MIN_2009w04) Test Module Template Wizard has limited support for C++, some manual changes are required from the user for C++ code to compile. The support in Test Module Template wizard can be improved if required.

17.1Enabling C++ in MINUnit or HardCoded test module

The following steps are needed for enabling C++ in MINUnit or HardCoded test module created by the Test Module Template Wizard.

1. Uncomment *AC_PROG_CXX* in `<module_name>/configure.in`
2. Uncomment *CC=g++* and *<module_name>_la_CXXFLAGS* in `<module_name>/Makefile.am` file.
3. Change the `<file_name>.c` files into `<file_name>.cpp` in `<module_name>/src` directory and `<module_name>/Makefile.am` file.

Appendix: GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the

publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following

text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols

a complete Transparent copy of the Document, free of added material.
If you use the latter option, you must take reasonably prudent steps,
when you begin distribution of Opaque copies in quantity, to ensure
that this Transparent copy will remain thus accessible at the stated
location until at least one year after the last time you distribute an
Opaque copy (directly or through your agents or retailers) of that
edition to the public.

It is requested, but not required, that you contact the authors of the
Document well before redistributing any large number of copies, to give
them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under
the conditions of sections 2 and 3 above, provided that you release
the Modified Version under precisely this License, with the Modified
Version filling the role of the Document, thus licensing distribution
and modification of the Modified Version to whoever possesses a copy
of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct
   from that of the Document, and from those of previous versions
   (which should, if there were any, be listed in the History section
   of the Document). You may use the same title as a previous version
   if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities
   responsible for authorship of the modifications in the Modified
   Version, together with at least five of the principal authors of the
   Document (all of its principal authors, if it has fewer than five),
   unless they release you from this requirement.
C. State on the Title page the name of the publisher of the
   Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications
   adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice
   giving the public permission to use the Modified Version under the
   terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections
   and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add
   to it an item stating at least the title, year, new authors, and
   publisher of the Modified Version as given on the Title Page. If
   there is no section Entitled "History" in the Document, create one
   stating the title, year, authors, and publisher of the Document as
   given on its Title Page, then add an item describing the Modified
   Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for
   public access to a Transparent copy of the Document, and likewise
   the network locations given in the Document for previous versions
   it was based on. These may be placed in the "History" section.
   You may omit a network location for a work that was published at

different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a

translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.2
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

```
If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts,
replace the "with...Texts." line with this:

    with the Invariant Sections being LIST THEIR TITLES, with the
    Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other
combination of the three, merge those two alternatives to suit the
situation.

If your document contains nontrivial examples of program code, we
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License,
to permit their use in free software.
```