TestFarm
System Reference Manual

Version 1r2

Table of Content

1 INTRODUCTION....4
2 TARGET APPLICATIONS....5
3 THE PEOPLE WORKING WITH A TESTFARM SYSTEM....7
4 ARCHITECTURE OVERVIEW....8
5 THE TEST INTERFACES....9
5.1 Overview....9
5.2 How the Test Interface communicates with the Test Engine....10
5.2.1 The Basics....10
5.2.2 Commands sent to the Test Interface....10
5.2.3 Replies received from the Test Interface....10
5.2.4 Overriding the Standard TestFarm Reply Format....11
5.3 Physical Connection of a Test Interface to the Test Engine....11
5.3.1 Connecting a TestFarm-Compliant Interface....11
5.3.2 Writing your own TestFarm-Compliant Interface....12
5.3.3 Connecting a Serial Link Interface....13
5.3.4 Connecting a TCP/IP Remote Interface....14
5.3.5 Connecting a GUI-Based Stand-Alone Instrument....15
5.4 Example of Interface Command Interpreter....16
6 THE SYSTEM SERVICES....17
6.1 What is it For ?....17
6.2 Defining the System Services....17
6.3 Running the System Services....17
7 THE TEST ENGINE....18
7.1 At a Glance....18
7.2 The Synchronization Mechanisms....19
7.3 The Test Log....20
7.3.1 What is the Test Log....20
7.3.2 The Test Log Format....20
7.3.3 Log Entries from the Test Engine....21
7.3.4 Global and Local Test Log....21
7.4 Managing Asynchronous Events and Real-Time constraints....22
7.4.1 Simultaneous and Asynchronous Events....22
7.4.2 Real-Time Constraints....22
8 THE TEST EXECUTION ENVIRONMENT....23
8.1 Overview....23
8.1.1 The Test Engine Library....23
8.1.2 The Test Interfaces Library....24
8.1.3 The Test Features Library....24
8.1.4 The Test Scripts....24
8.2 Symbols for Function Synopsis....24
8.3 Using the Test Engine Library....25
8.3.1 Test Engine Library Modules....25
8.3.2 Log File Management....25
8.3.3 Test Report Directory....26
8.3.4 Test Case Identification....26
8.3.5 Test Case interruption for Manual Testing....27
8.3.6 Managing Synchronization with Triggers....27
Introduction to Triggers....27
Trigger Expressions....28
Creating and Deleting Triggers....28
Managing the Trigger Occurrence Counter....29

© TestFarm 2006-2012

Waiting for Triggers....29
8.3.7 Magic Trigger Variables....31
Creating and Destroying a Magic Trigger Variable....31
Using a Magic Trigger Variable....32
Manipulating a Magic Trigger Variable....32
8.4 Writing a Test Interface Module....33
8.4.1 Test Interface Module Architecture and Environment....33
8.4.2 The Test Interface Base Class....34
8.4.3 Bringing a Test Interface Module to Life....38
8.5 Writing an XML System Definition....39
8.5.1 System Layout Philosophy....39
8.5.2 Structure of the System Definition File....40
8.5.3 Content of the System Definition File....42
File header....42
Configuration Elements....43
8.6 Generating the Test Feature Library....48
8.7 Using the Test Feature Library....48
8.7.1 System Information Resources....49
8.7.2 Interface Instance Variables....50
8.7.3 Service Start and Stop Functions....50
8.7.4 System Start and Stop Functions....50
8.7.5 Action Functions....51
The Concept....51
Prototyped Action Functions....52
9 THE TEST REPORT GENERATION SYSTEM....53
9.1 The Test Suite result files....53
9.2 Getting a Test Report document....53
9.2.1 Report Generation....53
9.2.2 Direct Browsing....53
9.3 Test Report Layout....54
9.3.1 Overview....54
9.3.2 Scenario and Test Cases....54
The Verdict of a Scenario:....54
The Criticity of a Scenario:....54
The Validation State of a Scenario:....54
9.3.3 The Report Header....54
9.3.4 The Verdict Summary....55
9.3.5 The Verdict Lists....55
Verdict List by Scenario....55
Verdict List by Test Case....55
9.3.6 The Output Dump....56
9.4 Test Report Stylesheet Parameters....56
10 THE USER INTERFACES....58
10.1 The Test Suite User Interface....58
10.1.1 Features....58
10.1.2 System Definition Attributes....58
10.2 The Manual User Interface....58
10.2.1 Purpose....58
10.2.2 Creating a Graphical Manual User Interface....58
10.2.3 Linking the Manual User Interface to the System....59
Widget Callbacks....59
System Definition Attributes....59
10.2.4 Using the Manual User Interface....59
11 GLOSSARY OF TERMS....60
12 LEGEND OF DIAGRAM LINKS....61
13 REFERENCES....62

© TestFarm 2006-2012

1 INTRODUCTION

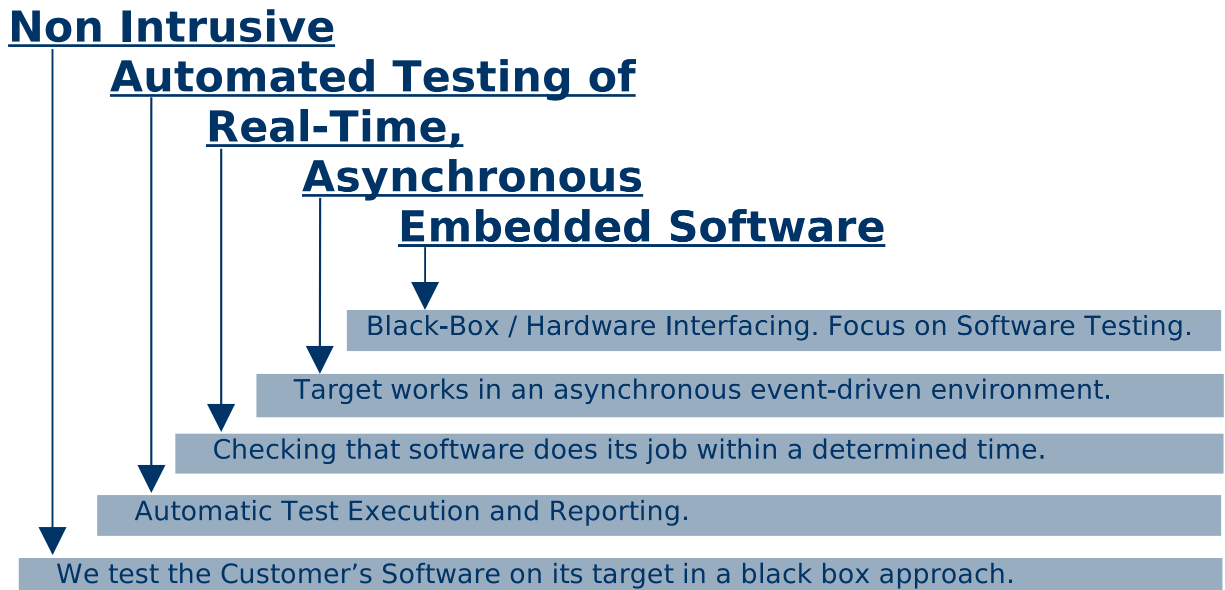
This document is the Reference Manual for the TestFarm Environment.

It explains how to build and configure a TestFarm platform. It describes the basic principles of the Test Engine. It also provides a reference documentation for the various TestFarm libraries used by the Test Scripts.

© *TestFarm 2006-2012*

2 Target Applications

The primary goal of the TestFarm platform is to perform non-intrusive automated testings of real-time embedded software. The following diagram summarize the main characteristics of a TestFarm system:



TestFarm is especially designed for testing **Embedded Systems** that use various standard or home made peripherals, in a heavily **Asynchronous** environment. Such systems are for instance payment terminals, which software has to interact with human actions (keypad, LCD, buttons, ...) while talking to supervision servers, smart-cards, etc. Of course, the TestFarm architecture is sufficiently open and **flexible** to be targeted at other environments such as Graphical User Interfaces or any application running on a workstation, but its main strength resides in the management of real-time and asynchronous events.

The TestFarm testing model conforms to a **Non-structural**, **Non-Intrusive** black box approach. Non-structural means that the Testing System does not have to be aware of the internal structure of the target software being tested. Whatever the OS, the programming language, the software architecture of the product under test, the Testing System is capable of stimulating and analysing the tested software through external interfacing. On an embedded system, this means to interface the target hardware. On a workstation, this means to have access to user interfaces and network interfaces. This gives the possibility to perform non-intrusive testing, without instrumenting the target software, thus allowing to really validate the software release that will be delivered to the final user (or the customer).

The TestFarm model allows to **Hide the Complexity** of the software by focusing on functional aspects. To cope with software complexity is the main weakness of structural / intrusive testing approaches, especially on object-oriented platforms with lots of abstract generic modules. This approach also allows to performs software testings while **Preserving Confidentiality** about the internal design of the system under test.

Test Suites are constructed with test scripts written in a **Well Known and Reputed language: PERL**. The TestFarm environment provides a rich collection of libraries

dedicated to that purpose. No proprietary language is used for writing the test scripts. This brings all the power of PERL to the Test System, while avoiding to learn yet another proprietary (and limited) language. For people who do not know this language, learning PERL is very easy and profitable, given the huge amount of resources available in the worldwide software and IT community.

The TestFarm **Wizard Tools** provide a clear Test Suite development framework. These tools play a key role in the “**Convergence To Simplicity**“ concept, the motto of the TestFarm spirit.
The test scripts can be written with a simple dedicated macro language. This macro-language provides strong script coding rules, thus allowing people with limited software development knowledge to write test scripts.
At the system configuration level, TestFarm administrators define the system interface and features in a XML file, from which system documentation and libraries are automatically generated.

During test suite execution, test results are stored in XML format. Test results include both data exchanged with the test interfaces and messages printed by the test scripts. Test report are automatically generated from the XML test results using customizable style sheets in standard XSL format. Default format for generated Test Reports is HTML. It contains several levels of details on the test results: from global statistics to detailed statistics and detailed information on what happened during the execution of the test suite.

The TestFarm platform provides flexibility and extensibility for connecting a **heterogeneous collection of Test Interfaces**. A Test Interface could be a measurement instrument, a software or hardware component, or even a complex LabView subsystem. Connected to the System Under Test, a Test Interface applies stimuli and gets reactions from it. This information are dispatched and synchronized by the Test Engine.

3 THE PEOPLE WORKING WITH A TESTFARM SYSTEM

Working in an TestFarm environment requires five basic roles that include different levels of skill. Of course, one people may combine several of these roles, but differentiating these roles allows to optimise human resource utilization within big corporate structures.

Role Class	***Role Identification***	***Description***
Manage the System	Instrument and Test Interface Supplier	Provide Test Interfaces Equipments.
	Test System Integrator and Administrator	Define system configuration, Integrate Test Equipments into the system. Configure and Setup the TestFarm platform.
Manage Test Suites	Test Project Manager	Manage the Development of Test Suites within the development process of the product under test. Collect the test reports and Help the product development team analyse the test results.
	Test Script Developer	Write the test scripts.
Execute Test Suites	Test Operator	Launch the Test Suites execution and the Test Report generation. Check the TestFarm systems are working properly to assist the system integrator(s) and administrator(s).

© TestFarm 2006-2012

4 Architecture Overview

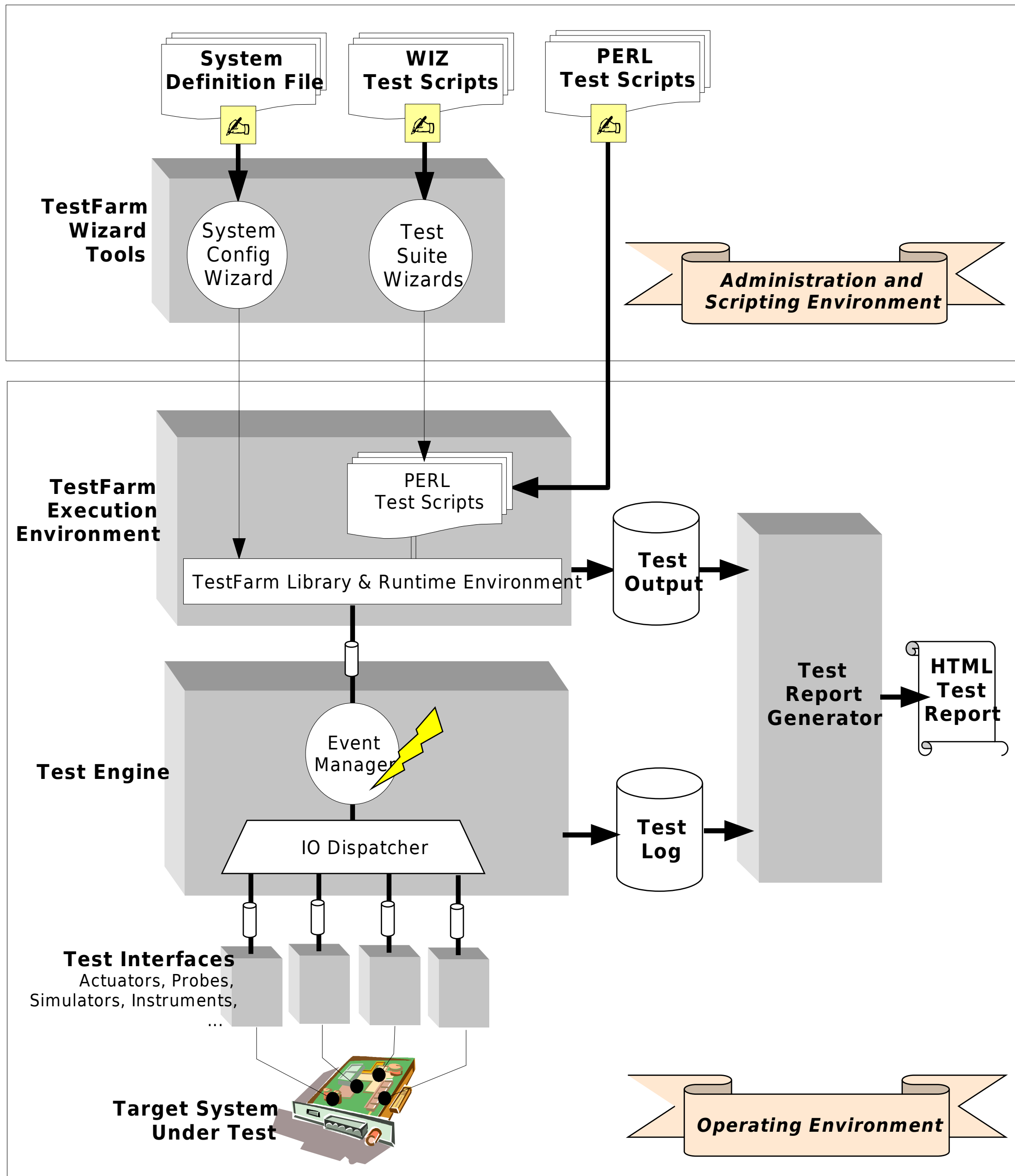


Figure 4.1: Architecture Overview

The TestFarm System Architecture is designed to connect a heterogeneous park of test interfaces, and to drive them from a powerful scripting environment. A test report generator automatically produces formatted HTML reports.

5 The Test Interfaces

5.1 Overview

The TestFarm platform provides flexibility and extensibility for connecting a heterogeneous park of **Test Interfaces**. A Test Interface may be a piece of software, a piece of hardware, or a combination of both. It is connected to the system under test to apply stimuli and get reactions from it. This information are dispatched and synchronized by the **Test Engine**.

For instance, if the tested product is a payment terminal, one would need a keypad actuator, a LCD spy, some switches, a smart-card simulator, etc. The Test Engine talks with these interfaces through a well-specified communication format based on ASCII command / replies.

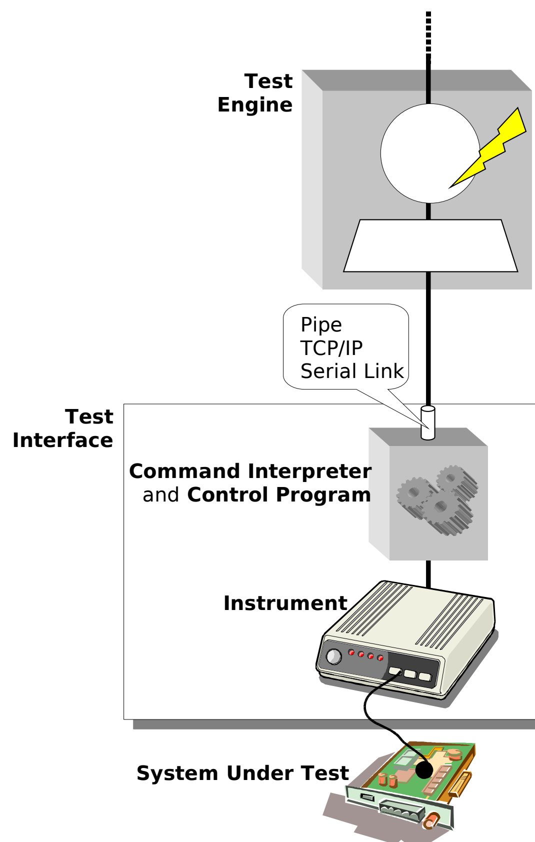


Figure 5.1 : Connecting a Test Interface to the Test engine

In the TestFarm terminology, the hardware part of a Test Interface is called an **Instrument**. The software part is called a **Command Interpreter** or a **Control Program**. In practice, the term “Control Program” is used when the software is controlling a piece of hardware through a dedicated device driver: for example, an I/O board plugged on the PCI bus of the computer is driven by a control program. The term “Command Interpreter” is preferred when the software acts only as a bridge between two different data formats: for example, an RS232 spy would convert the binary data received from the serial port into a readable hexadecimal dump.

A Test Interface can be connected to the Test Engine through 3 types of physical channels: Piped sub-process, TCP/IP or Serial Link:

- Most of the time, the **piped sub-process** solution is employed to implement a command interpreter between a physical instrument and the test engine, whatever the physical link is. No special API or development environment is required for integrating a new Test Interface into a TestFarm system. It is even possible (but not limited) to write an interface command interpreter in PERL, thus allowing to use the same programming language for both SUT Interfacing and Test Script writing.
- When accessing a remote instrument is necessary, the solution is to use a **TCP/IP link**. This solution is ideal if one has to encapsulate a testing sub-system into the TestFarm environment, such as network protocol simulators.
- A Modem, a Multimeter, a Function Generator, etc. can be used as Test Interfaces directly connected to the **serial port** of the computer running the Test Engine.

5.2 How the Test Interface communicates with the Test Engine

5.2.1 The Basics

The messages exchanged between the Test Interfaces and the Test Engine are ASCII strings. ASCII format is used for three main reasons:

- **Portability**: this avoids data formatting problems due to the use of different equipments (word width, byte ordering, etc.).
- **Readability** of the Test Log by human beings.
- ASCII text analysis is **the strength of PERL**. This is the way the TestFarm environment can easily interact with a heterogeneous park of Test Interfaces.

The Test Engine is in charge of dispatching commands to the Interfaces, and processing the replies. Processing replies consists in dating them, checking for awaited events, and storing them into a Test Log file. The Test Engine can be considered as a router between the Test Interfaces and the PERL Script Interpreter.

5.2.2 Commands sent to the Test Interface

The Test Engine simply sends ASCII commands to the Test Interfaces. A command can contain all the arguments that may be useful. Arguments are separated by one or more spaces (ASCII 20h) or tabs (ASCII 08h). A command line is terminated by a *NewLine* sequence, which could be either CR (Cariage Return, ASCII 0Dh), LF (Line Feed, ASCII 0Ah) or CR-LF.

Examples of commands:

```
version
sw 2,3:on
```

5.2.3 Replies received from the Test Interface

A reply from a Test Interface must conform to the **Standard TestFarm Reply Format**. This format consists in splitting the reply into 3 fields, separated by one or more spaces (ASCII 20h) or tabulations (ASCII 08h):

Reply Header		***Reply Body***
Local Time Stamp	Tag	Information
µseconds	Text	Text

The Local Time Stamp must not contain spaces or tabs. This field is a local date expressed in microseconds as a decimal number. The time stamp should not necessarily

© TestFarm 2006-2012

have a one-microsecond accuracy, but it must be expressed with this unit. The time stamp date is generated locally by the Test Interface, in order for the Test Script to be able to perform high precision time measurements between the messages generated by this Test Interface. Generating a time stamp is not mandatory for a Test Interface: in such a case, the local time stamp should be replaced with a star '*' (ASCII 2Ah).

The Tag must not contain spaces or tabs. It is a keyword reflecting which subset of the Test Interface the message comes from. It is an arbitrary ASCII word bound to the insight design of the interface. It may be exploited from the test scripts for filtering purpose. If the notion of tag is not relevant for the Test Interface, it should be replaced with a star '*' (ASCII 2Ah).

The Info field may contain spaces or tabs. This field carries the actual useful data of the message, which depend on the purpose of the message. No special format is imposed, excepted that it must be an ASCII text.

Examples of replies with a local time stamp and a tag:

```
3590804   VERSION UR111 21 (Aug 22 2006)
17453645  SWITCH  2:on
```

Examples of replies with dummy time stamp:

```
* MODE    Manual mode enabled
* VERSION 1.0 (Jul 15 2006)
```

5.2.4 Overriding the Standard TestFarm Reply Format

It may happen that a Test Interface cannot produce a message header: for instance a serial modem, or any off-the-shelf device that was not primarily designed for the TestFarm architecture. It may also appear that a message header is simply useless for the Test Interface.

The Test Engine can handle this situation: in such a case, the reply header can be omitted if you specify the "`-noheader`" option flag in the configuration template of the Test Interface module (section 8.4). The Test Engine will then take this option into account by automatically prepend the missing header with dummy values (stars).

5.3 Physical Connection of a Test Interface to the Test Engine

5.3.1 Connecting a TestFarm-Compliant Interface

A TestFarm-Compliant Interface is an instrument that comes with a control program which can be directly connected to the Test Engine as a piped sub-process. The control program is no more than a command interpreter that follows the TestFarm command/reply format recommendations. The controlled instrument is accessed through a hardware port (Parallel Port, Serial Link, Network Link, PCI, USB, GPIB, etc.), depending on its specification.

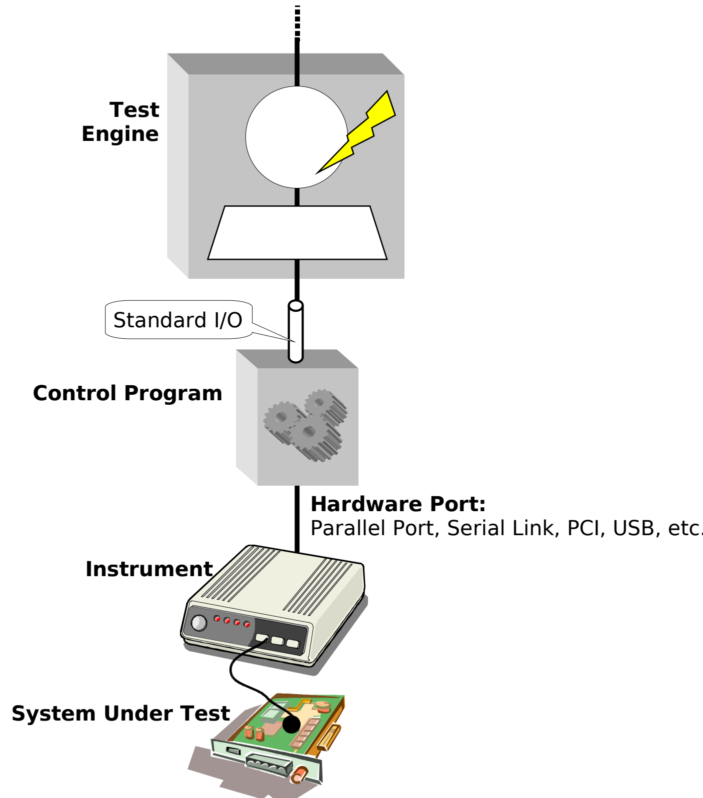


Figure 5.2 : Connecting a TestFarm-compliant Test Interface

The TestFarm platform has a catalog of standard interfaces. This catalog is subject to modification and enrichment:

- The **UR111** Interface provides 14 relays and 8 opto-coupled logical inputs.
- The **UF120** is an input recording device with a 18-bit FIFO and 4 opto-coupled logical inputs. The associated control program allows to defined logical input data flows, and to perform some post-processing for LCD display emulation, input monitoring, filtering, etc.
- The **Audio Interface** controls an off-the-shelf PC sound card. It allows to generate audio signals and to perform some spectrum analysis for frequency detection. It is also capable to perform FSK modulation and demodulation.
- **TestFarm Virtual User** (TVU) is a test interface that connects to a VNC server or a Video Capture Device, thus allowing to control and analyse a Graphical User Interface. VNC is a multi-platform remote control tool for Graphical User Interfaces.
  TVU can be used in two manners:
  ☞ To control the GUI of the target system under test (in either Windows, Unix/Linux or Macintosh environments)
  ☞ To control the GUI of an instrument that was not primarily designed for automated testing. This is typically the situation of those numerous lab equipments designed for human interaction only, with a Graphical User Interface as a unique mean of access.

5.3.2 Writing your own TestFarm-Compliant Interface

Using TestFarm standard Interfaces is not the only way of interfacing the target system under test.

If you have a home made Test Interface with dedicated hardware, it is easy to develop your own TestFarm-compliant control programs. No API or special libraries are required. You can use the language and the development environment you are accustomed to. You can even use a script interpreter (PERL, Python, Java, etc.).

© TestFarm 2006-2012

The only requirements for developing a TestFarm-compliant Test Interface control program are:

- The control program should receive ASCII commands from the standard input;
- The control program should dump ASCII the reply messages to the standard output, conforming to the format described in 5.2.3.

5.3.3 Connecting a Serial Link Interface

Connecting a Serial Link device as a Test Interface can be done directly if the device supports ASCII command/reply. This is the case of simple devices such as multimeters.

If the device does not natively support ASCII command/reply, you can interface it with a command interpreter. A command interpreter is a process that acts as an ASCII bridge between the device and the Test Engine. From a conceptual point of view, this is no more than a control program described in the previous section.

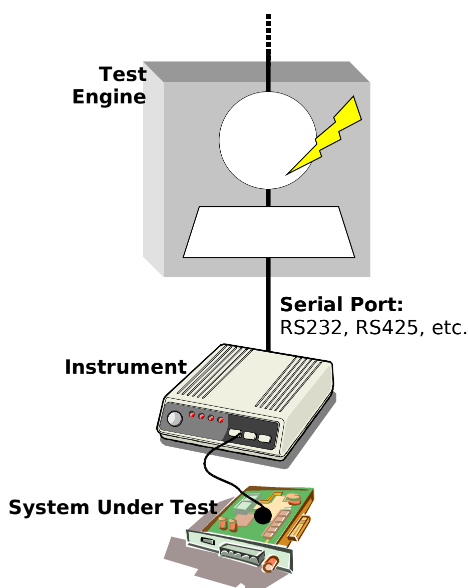


Figure 5.3 : Direct Connection to an RS232 Interface

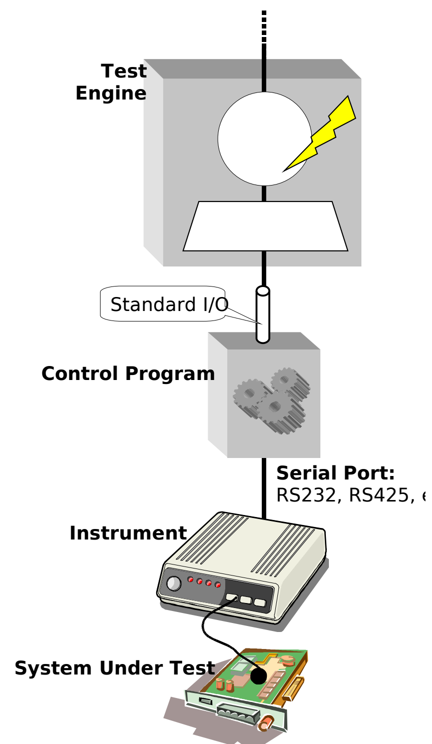


Figure 5.4 : Connecting an RS232 Interface through a Command Interpreter

© TestFarm 2006-2012

5.3.4 Connecting a TCP/IP Remote Interface

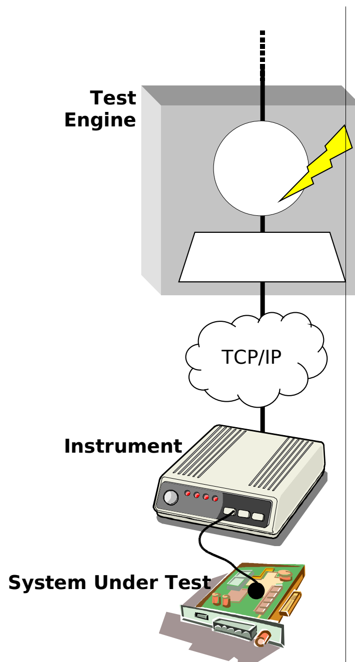


Figure 5.5 : Direct TCP/IP connection

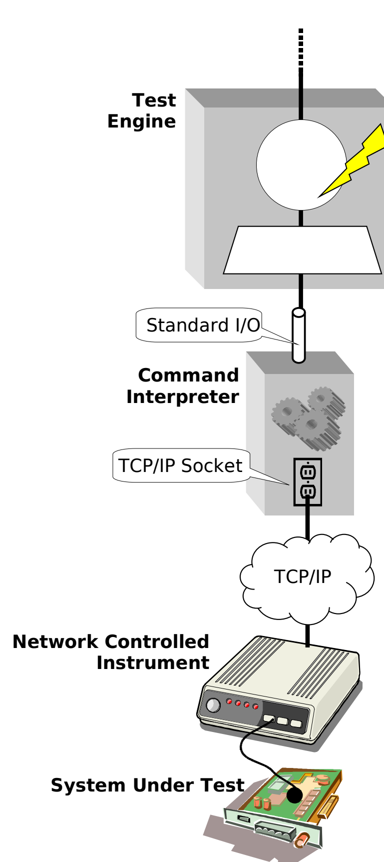


Figure 5.6 : TCP/IP connection through a Command Interpreter

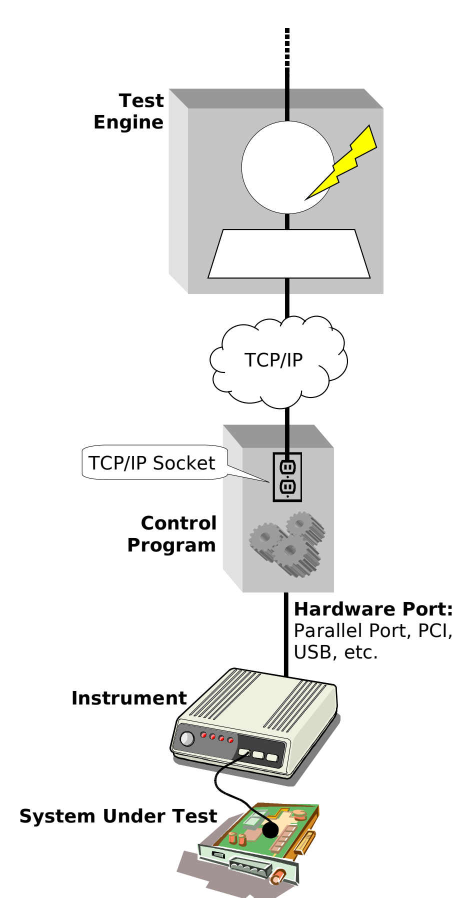


Figure 5.7 : TCP/IP connection to a remote control program

The Test Engine is capable of communicating with a TCP/IP connection. This allows to talk directly with instruments that can be accessed this way. This also permits to control a complex instrument driven by its own computer, by establishing a TCP/IP connection between the Test Engine and this computer.

The three figures above illustrate some methods to connect a Test Interface using a TCP/IP connection:

- The configuration shown in Figure 5.5 is trivial: the Test Engine directly connects to the instrument. This is possible when the instrument uses ASCII format as its user interface.
- The configuration of Figure 5.6 uses an additional command interpreter. This approach is necessary when the instrument is not driven by ASCII command/reply, but rather with a protocol that uses binary packets. The role of the command interpreter is to convert the binary packet protocol into an ASCII command/reply environment.
- Figure 5.7 shows a configuration where the instrument is typically a complex device running on a separate computer. On this computer, we embed a control program that talks with the API's and/or the device driver of the instrument. From a conceptual point of view, this configuration is similar to the "TestFarm-compliant Interface" situation described in section 5.3.1. The only difference resides in the fact that the control program is not accessed as a piped sub-process, but as a remote TCP/IP service.

5.3.5 Connecting a GUI-Based Stand-Alone Instrument

The TestFarm platform provides a connectivity scheme to drive instruments that are not primarily designed to be integrated into an automated testing environment. In the market or in your lab, you may find that kind of tool that can be used uniquely by a human being, in a stand-alone manner. When there is no API available, no device driver, no clear documentation about how to write such a device driver, no remote control feature, but only a Graphical User Interface, the ultimate solution is to drive this kind of tool from its native GUI through a VNC server.

VNC is an open multi-OS tool that allows to take control of a graphical environment through a network link. This tool was initially designed by the AT&T Laboratories at Cambridge (UK): "*VNC stands for Virtual Network Computing. It is, in essence, a remote display system which allows you to view a computing 'desktop' environment not only on the machine where it is running, but from anywhere on the Internet and from a wide variety of machine architectures*."

VNC is constructed as a client-server architecture. The server runs in the machine where the Graphical Environment has to be remotely controlled. The client is then capable to remotely mimic the keyboard and mouse actions through the network, as well as fetching portions of the screen. The protocol used by VNC to perform its remote control is named RFB, for *Remote Frame Buffer*. The TestFarm platform includes a VNC client that plays the role of a virtual VNC viewer. This tool provides a command set to emulate keyboard and mouse actions, and to analyze areas of the display. This provides the possibility to drive a VNC server from a Test Script.

Figure 5.8 illustrates how to integrate a stand-alone instrument into the TestFarm environment. For instance, the instrument may be a microprocessor emulator, a logic analyzer, an digital storage oscilloscope, etc.

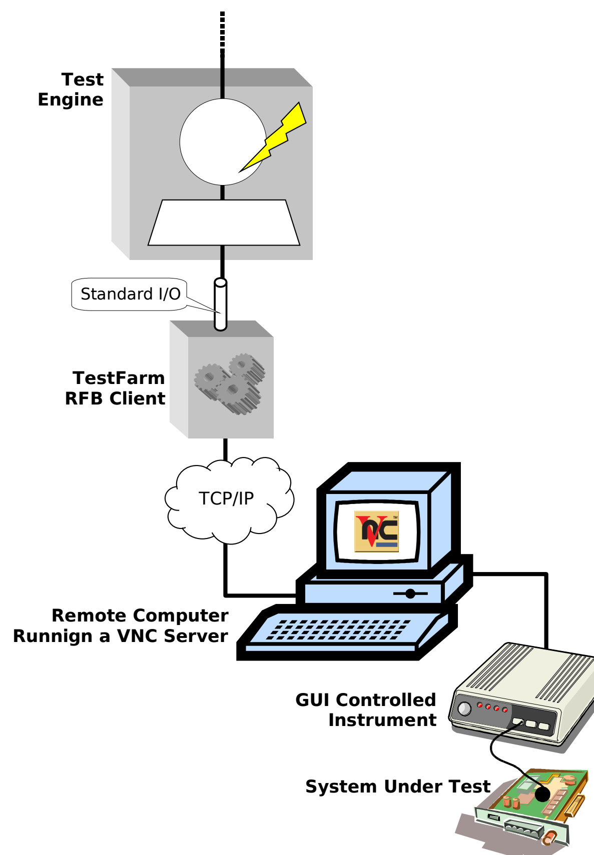


Figure 5.8 : Using a GUI-based instrument as a Test Interface

© TestFarm 2006-2012

5.4 Example of Interface Command Interpreter

This section presents an example of a command interpreter connected to the Test Engine as a piped sub-process. This example is written in PERL, but any other interpreted or compiled language would be fine.

The command interpreter is a dummy program that is not really useful. It simply gives a basic framework of a typical TestFarm-compliant command interpreter that manages Local Time Stamps and Tags.

Only three commands are recognized. Other commands produce an error message:

Command	***Description***
`version`	Displays the command interpreter version, according to the CVS repository it is stored in.
`echo`	Echos the arguments it is given.
`exit`	Exits the command interpreter.

```
#! /usr/bin/perl -w

# ===========================================================================
# The Sample ECHO Command Interpreter
# A dummy Test Interface command interpreter that
# does nothing really useful.
# Three commands are recognized:
#   version displays the command interpreter version,
#           according to the CVS repository it is stored in
#   echo    echos the arguments it is given
#   exit    exits the command interpreter
# Other commands produce an error message.
# ===========================================================================

use FileHandle;
use Time::HiRes qw(gettimeofday);

# Construct the version id
my $version = '$Revision: 657 $ ';  # SVN puts the right value here
$version =~ s/[\s\$:]//g;
$version =~ s/Revision//;
$version = 0 if ( $version eq "" );

# Get the big-bang date
$t0 = gettimeofday();

# Disable output buffering
STDOUT->autoflush();

while ( <STDIN> ) {
  my ($cmd, @args) = split /\s+/;

  # Compute and dump the time stamp in microseconds
  my $t1 = gettimeofday();
  my $tstamp = ($t1 - $t0) * 1000000;

  # Process the command
  if ( $cmd eq "version" ) {
    printf "%d VERSION ECHO $version\n", $tstamp;
  }
  elsif ( $cmd eq "echo" ) {
    printf "%d ECHO @args\n", $tstamp;
  }
  elsif ( $cmd eq "exit" ) {
    last;
  }
  else {
    printf "%d ERROR Illegal command '$cmd'\n", $tstamp;
  }
}
```

Figure 5.9 : A sample Interface Command Interpreter

6 The System Services

6.1 What is it For ?

A System Service is a program that needs to run in order for the TestFarm environment to work properly. A System Service could be any kind of program that assists the Test Interfaces to achieve their job: a visualiation tool, a network service, etc. The Concept of System Services is similar to that encountered in well known computer operating systems like Unix™/Linux™ or Windows™.

6.2 Defining the System Services

The System Services are declared in the SERVICE elements of the System Definition File presented in section Error: Reference source not found. Each service is given a identifier (attribute "id") and an optional activation mode (attribute "mode") indicating whether the service should be started with the Test Suite User Interface, the Manual User Interface, or any of them.

Two types of System Services are available:

- A **Command-Oriented Service** is specified as a command line in the "`cmd`" attribute, which indicates the command to execute to run the program that implements the service. In practice, this is the most frequently used type of System Service, because of its ease and simplicity.
- A **Function-Oriented Service** is specified by declaring three PERL functions START, STOP and STATUS, which respectively starts the service, stops the service, and returns the service status. These functions are declared in the System Definition File within the START, STOP and STATUS sub-elements of the SERVICE element. This kind of implementation is reserved for complex services requiring some PERL code to be managed.

After being generated using the TestFarm Configuration Wizard, the Test Feature Library contains two function for starting and stopping the services declared in the System Definition File. These functions are automatically called when the user interfaces are launched and terminated: they should not be called from a Test Script.

6.3 Running the System Services

The System Services are automatically started when the Test Suite User Interface or the Manual User Interface is launched . The same services are stopped when the user interface is terminated. Please refer to chapter Error: Reference source not foundError: Reference source not found, page Error: Reference source not found for a description of the TestFarm user interfaces.

Alternatively, it is possible to manually start and stop one or more System Services using the "`testfarm-service`" command. Please refer to the TestFarm User's Manual [2] for details.

If a service appears to be already started, it is not started again. The method employed to determine whether a service is started or note depends on the type of service:

- For a Command-Oriented Service, it is assumed to be started if a process that executes this command is running.

- For a Function-Oriented Service, its status is returned by the code written in the STATUS element of the service declaration in the System Definition File.

© *TestFarm 2006-2012*

7 The Test Engine

7.1 At a Glance

The role of the Test Engine is to dispatch and synchronize the ASCII commands and replies between the Test Interfaces and the Test Scripts. The features provided by the Test Engine can be summarized as follows:

- Commands from the Test Scripts are routed to the proper Test Interface;
- The Test Scripts can define **Triggers**, which consist in asking the Test Engine to detect replies coming from the Test Interfaces that match a regular expression.
- Replies from the Test Interfaces are stored into a **Test Log** file. They are also checked in order to update the defined Triggers. When a reply matches a Trigger, this trigger is **raised**. This is the way incoming events are detected from the Test Interfaces.
- The Test Scripts can use some synchronization primitives in order to wait for a Trigger or a group of Triggers to be raised.

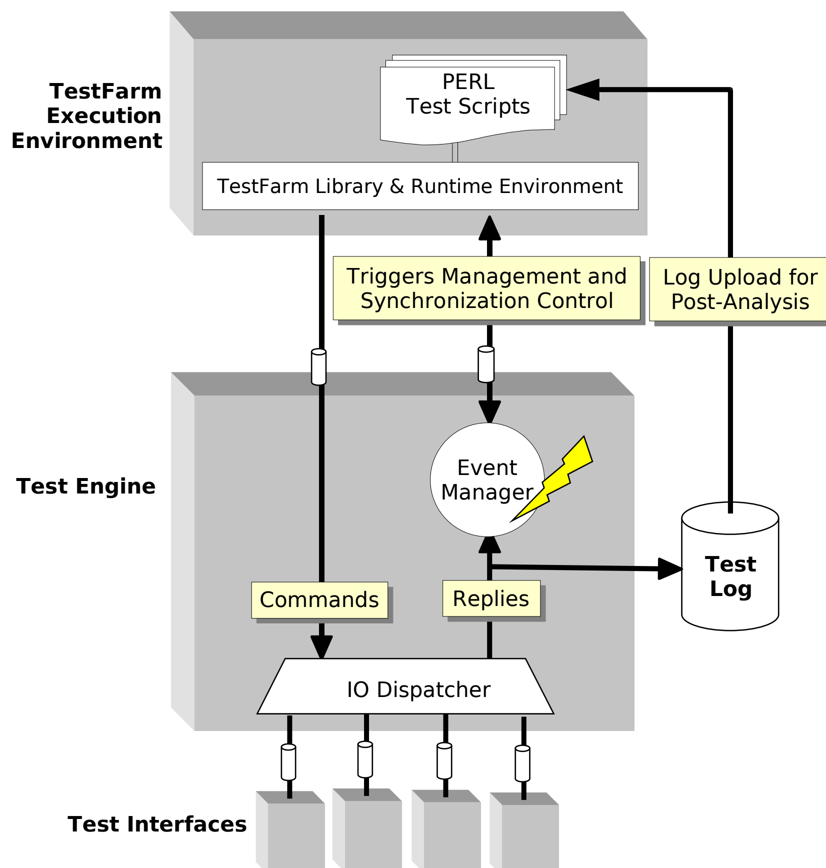


Figure 7.1: Overview of the Test Engine

© TestFarm 2006-2012

7.2 The Synchronization Mechanisms

The strength of the TestFarm platform is to provide advanced mechanisms that manage a large amount of simultaneous events coming asynchronously from several Test Interfaces. This is a typical environment in the world of embedded systems. Some embedded systems like payment terminals or automotive equipments need to control lots of peripherals. A non-exhaustive list of such peripherals includes keypads, LCD screens, smart-card readers, switches, network links, sensors, actuators, etc.

The key concept of the Test Engine resides in the command / reply dialog with the Test Interfaces:

- **Commands are asynchronous.**
  All the commands sent from the Test Engine to the Test Interface are asynchronous. Commands emitted by the Test Script are just forwarded to the Test Interface, without waiting for its completion.
  Sending commands to the Test Interfaces is done by calling the functions available in the Test Feature Library, which in turn invoke the Test Interface Library.
- **Synchronization is explicit.**
  The Test Script must explicitly define its synchronization points. In order to do that, it defines some triggers that listen to expected (or unexpected) events, and wait for these triggers.
  The Test Engine Library provides some PERL functions to define and wait for triggers. The Test Interface or Test Feature Libraries may also provide some functions to define preformatted triggers related to a particular Test Interface.

The typical execution flow for a test execution step is:

1	Prepare Triggers
2	Perform actions that are supposed to produce events the triggers are listening to
3	Wait for Triggers within a specified timeout
4	Check for Timeout and Raised Trigger(s)
5	Analyze the events that caused the Trigger(s) to be raised.

Figure 7.2 shows the subsection of a Test Script that uses our dummy Test Interface "`echo.pl`" described in 5.4. This is just a preliminary taste: refer to the following chapters of this manual to learn in detail how to write a test script. This example is intentionally simple. In the real word, we use a lot of triggers that controls events from several Test Interfaces. Wait points are often defined with a combinatorial expression of several triggers, constructed with logical operators. It is also a very common practice to define triggers to catch abnormal events.

```
# Define a trigger "MESSAGE" that will listen to message "Hello"
# from our Test Interface instanciated as object $INTERFACE
TrigDef($INTERFACE, "MESSAGE", "Hello");

# Action: send command "echo Hello World"
$INTERFACE->echo("Hello World");

# Wait for the command to reply within 1 s
my $trigged = TrigWait("MESSAGE", "1s");

# Analyse the event if detected
if ( $trigged ) {
  # Get the reply that caused the trigger to be raised
  my $reply = TrigInfo("MESSAGE");

  # Extract some interesting info from this reply,
  # using the PERL test extraction features
  $reply =~ /^(\d+) +ECHO +(.+)$/;

  print "EVENT CAUGHT: timestamp=", $1, " message='", $2, "'\n";
}

# Timeout
else {
  print "TIMEOUT !\n";
  $verdict = 1;
}
```

Figure 7.2 : A sample Test Script showing a simple synchronization

7.3 The Test Log

7.3.1 What is the Test Log

The Test Engine records the replies from all Test Interfaces and stores them into a Test Log file. The purpose of this Log File is to allow off-line post-analysis of the recorded replies: this can be done automatically from the Test Script, or by a human being who browses the Test Report. Since the log entries are time stamped, it is possible to take into account the real-time aspects of what happened during the test execution.

7.3.2 The Test Log Format

Before being recorded, each line of the Test Log is completed with a **Log Header** by the Test Engine:

Log Entry Header				***Test Interface Reply (5.2.3)***		
HR Date	HR Time	Global Time Stamp	Interface Id	Local Time Stamp	Tag	Information
DD-MMM-YYYY	hh:mm:ss	µseconds	Text	µseconds	Text	Text

The first two fields of The Log Entry Header provide human-readable date and time. This additional dating information will be useful for reading the Test Log as a part of a Test Report.

The third field is a Global Time Stamp, generated by the Test Engine itself. It allows to perform time measurements between messages coming from different Test Interfaces. The Global Time stamp may be less accurate than the Local Time Stamp, but it provides a common time base for all the Test Interfaces.

☞ The Local Time Stamp is local to the Test Interface that generated it. It should be used only for time measurements within this Test Interface. Since the clocks of the Test Interfaces are not supposed to be synchronized (unless your hardware configuration allows it), using the Local Time Stamp for measuring time between different Test Interfaces makes no sense: for that purpose, Global Time Stamps should be used.

7.3.3 Log Entries from the Test Engine

The Test Engine also dumps entries to the Test Log. This provides additional information about what is performed by the Test Script concerning Test Interface setup, trigger definitions and synchronization. When analyzing a problem in the Test Log, this information are precious to correlate the replies of the Test Interface to the progress of the Test Script.

The format of the log entries produced by the Test Engine is summarized in the tables below. A detailed description is given in section 8.3.

Log Entry Header				***Test Engine Message***		
HR Date	HR Time	Global Time Stamp			Tag	Information
DD-MMM-YYYY	hh:mm:ss	µseconds	ENGINE	*	●	●

Category	***Tag***	***Description***
Global Information	VERSION	The release version of the Test Engine
	RESULT	The name of the Local Log file (see 7.3.4)
	ECHO	An informational message echoed by the Test Script
Test Case Delimitation	CASE	Name of the Test Case that begins
	DONE	Name of the Test Case that finishes
	VERDICT	Verdict of the Test Case that just finishes: PASSED, FAILED, INCONCLUSIVE, SKIP
Test Interface Connectivity	PERIPH	Test Interface Declaration
	OPEN	A Test Interface is started
	CLOSE	A Test Interface is stopped
Event Management	TRIG	Trigger definition or information retrieval
	TIMEOUT	Wait Timeout setup
	WAIT	Entering or Leaving a wait operation

7.3.4 Global and Local Test Log

In order to make access to the Test Log easier, the Test Engine feeds two Test Log files:

- The **Global Log file** contains the log entries of the whole Test Suite. This file is a document intended to be attached to the Test Report, as a detailed film of what physically occurred during the Test Suite. The Test Report generator puts some hyperlinks that points to a colorized HTML version of this log, thus allowing to easily browse the Test Report.
- The **Local Log file** contains the log entries of the Test Script that is being currently executed. This file prevents the Test Script from opening the whole log, which could be

very resource consuming. When a new Test Script is executed, the previous Local Log file is deleted and a new one is created with the same file name.

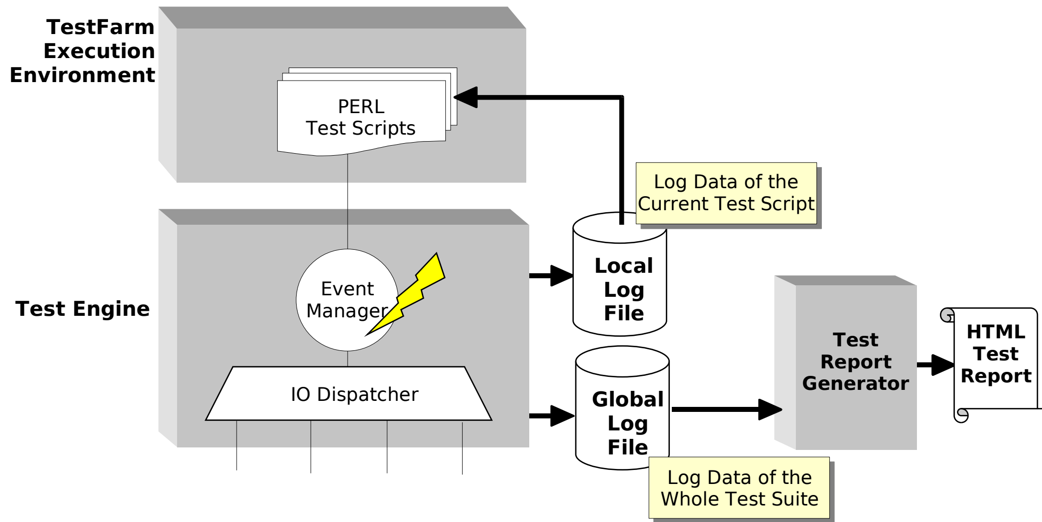


Figure 7.3 : Data flows of the Local and Global Test Logs

☞ The name of the Local Log file is returned by the function `LocalLog()` of the Test Engine Library. The Test Script should call this function to determine which file to open to read the Local Log.

7.4 Managing Asynchronous Events and Real-Time constraints

7.4.1 Simultaneous and Asynchronous Events

Managing simultaneous and asynchronous events that may come from the Test Interface can be achieved by using Triggers. The goal of triggers is to catch events, to count them, and to report their occurrence status. The Test Scripts is then capable of waiting for one or more trigger(s) to continue its execution. The Test Script can define as many Triggers as necessary to catch events from any Test Interfaces. A trigger is constructed with a regular expression, thus making it possible to catch an event either as an exact text match, or as a pattern containing wildcards. This brings the possibility to detect a family of events from a single trigger definition.

If synchronization through the Test Engine mechanisms is not fast enough, some hardware synchronization gears must be implemented at the Test Interface level. Many Test Instruments supports this kind of synchronization by providing external trigger input and output. With this approach, the test script won't manage the synchronization itself, but will rather setup the concerned Test Interfaces to synchronize each other using a hardware trigger interconnection.

7.4.2 Real-Time Constraints

Real Time constraints can be managed by two means:

- By dealing with timeouts when waiting for triggers;
- By analyzing time stamps on-the-fly, as events are triggered during the test execution;

- By analyzing the Test Log and its time stamps, at the end of the test script after the test actions are finished. This is useful when a Test Interface records an internal log that can be only fetched after the test script has finished its job.

© *TestFarm 2006-2012*

8 The Test Execution Environment

8.1 Overview

Once a park of Test Interfaces is connected to the Test Engine, it becomes possible to play with them from the Test Execution Environment. The Test Execution environment is based on the PERL interpreter. It is split into 4 layered blocks: the **Test Interfaces Library**, the **Test Engine Library**, the **Test Features Library** and the **Test Scripts**.

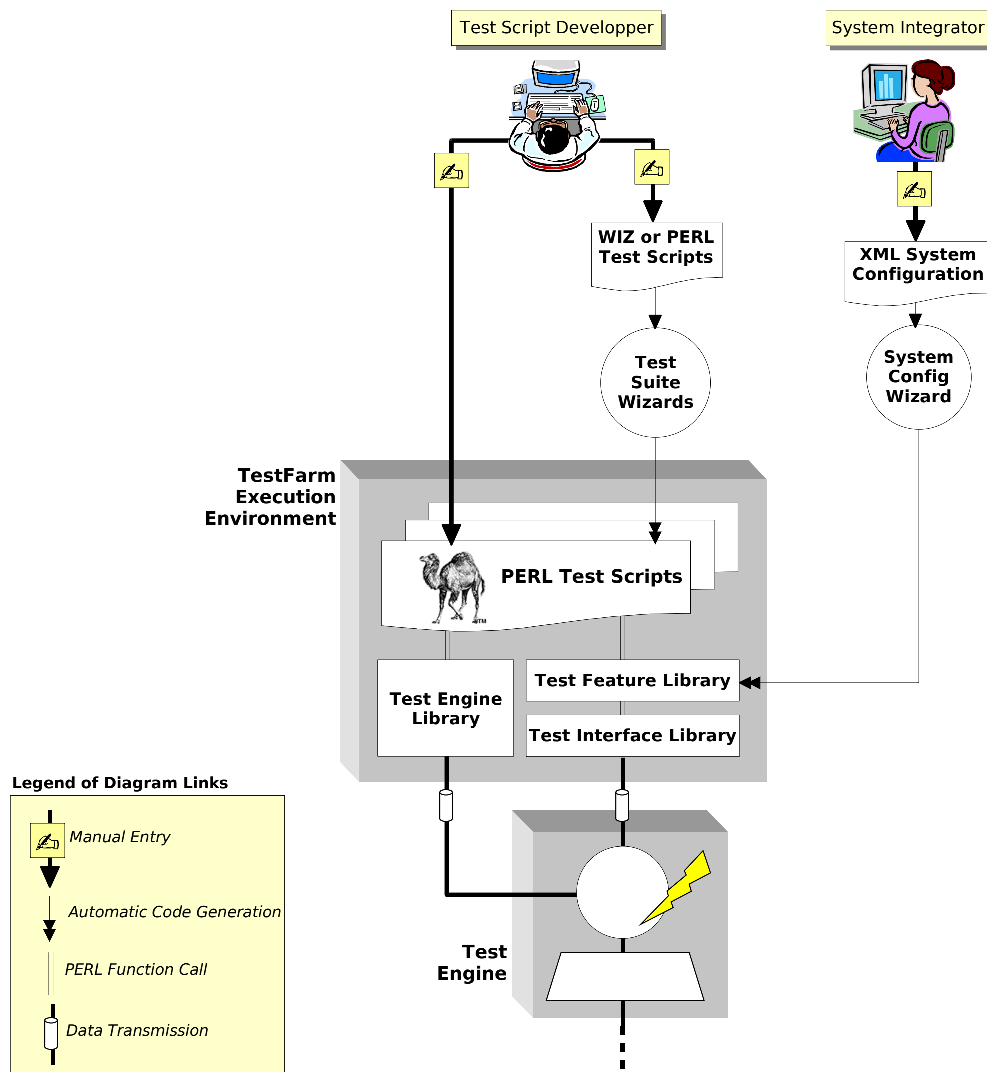


Figure 8.1 : Overview of the Test Execution Environment

8.1.1 The Test Engine Library

The **Test Engine Library** provides the basics TestFarm functions by which a Test Script can control the Event Manager of the Test Engine. This is a key component for managing real-time and asynchronous events.

This library is included as a standard component of the TestFarm platform.

© TestFarm 2006-2012

8.1.2 The Test Interfaces Library

The lowest-level layer is the **Test Interface Library**, which provides a collection of PERL modules that communicate with the Test Interfaces through the Test Engine. There is one module per type of Test Interface. The role of such a module is to convert the Test Interface commands into a set of PERL functions, in order for the Test Scripts to view the Test Interfaces as a collection of PERL functions, rather than a command/reply stream. At this level, we do not matter about the final purpose of an interface, but just how to use this interface: for example, a relay that replaces a reset button on the product under test would be seen as “Relay Number *X* of Interface *Y*”, not as “Reset Button”.

Each type of Test Interface comes with a module included in the Test Interface Library. To make an analogy between a TestFarm system and an operating system, the Test Interface Library can be seen as a collection of “Device Drivers”.

From a programming point of view, a Test Interface module implements an object class providing methods dedicated to the Test Interface features. Several instances of the object class may be created if the testing system contains several Test Interfaces of the same type. Using an object-oriented approach is helpful for structuring the system architecture, but do not be afraid: this is transparent to the people who write test scripts. They do not have to be skilled in object-oriented programming!

All the TestFarm-compliant Test Interfaces are delivered with a PERL module. If you have to integrate a home made Test Interface into your system, you can write your own by following the instructions in section 8.4. Writing a Test Interface module is the job of the Test Interface supplier or the TestFarm system integrator when the system is being constructed.

8.1.3 The Test Features Library

The second layer of the Test Execution environment is the **Test Feature Library**. In this layer, we declare a list of Test Features and map them among the hardware resources provided by Test Interfaces. To take up the example above, we shall define at this level an abstract feature identified as “Reset Button”, and populate it with some **Test Actions** such as “Push the Reset Button”.

The Test Feature library is generated automatically from an **XML System Definition File** written by the TestFarm system integrator. The TestFarm Configuration Wizard automatically generates the PERL code from this file. See section 8.5 for further details.

8.1.4 The Test Scripts

The highest level gathers the **Test Scripts**, which interact with the Test Interfaces and the Test Engine from the Test Feature library.

Test Scripts are gathered into a Test Suite. They are written by the Test Script developers, directly in PERL or using the TestFarm Wizard Tools. A detailed description of how to write a Test Suite is given in the *TestFarm Test Suite Reference Manual* [1]Error: Reference source not found.

8.2 Symbols for Function Synopsis

Sections 8.3 and 8.4 give a synopsis of the various functions and object method exported by the library.

A synopsis begins with a `use` PERL statement indicating which module the function or method is imported from. Then follow some statements showing how the function/method should be invoked, what parameters are expected, and which value is returned.

Some parameters are designated with symbols that have a particular meaning. Such special symbols are listed in the table below. The other parameters are strings or numbers that are given a self-explanatory name.

Symbol	Description
"*InterfaceId*"	A string constant or a string variable containing a Test Interface identifier
$InterfaceId	A string variable in which a Test Interface identifier is stored
$InterfaceObj	A Test Interface object variable
"*TrigId*"	A string constant or a string variable containing a Trigger identifier
"*TrigExpr*"	A string constant or a string variable containing a Boolean expression that combines Trigger identifiers.
$TrigId	A string variable in which a Trigger identifier is stored
$TrigList	A string variable in which a space-separated list of Trigger identifier is stored
$TrigVar	A Magic Trigger variable
"*Regex*"	A string constant or a string variable containing a POSIX regular expression
"*TimeValue*"	A string constant or a string variable containing a timeout value, which is a positive integer. By default, the timeout is in milliseconds, but the value may be suffixed with a time unit qualifier "ms", "s", "min" or "h" that indicate a time respectively in milliseconds, seconds, minutes or hours (e.g. "10s", "5min", etc.)
$Info	A string variable in which a Test Interface reply is stored

8.3 Using the Test Engine Library

8.3.1 Test Engine Library Modules

The Test Engine Library provides two PERL modules:

- **TestFarm::Engine** provides basic primitives for configuring the Test Engine. The most useful functions are devoted to Log File and Test Case management. Some other functions are available in this module, but they are reserved for low-level operations and should normally not be invoked directly from the Test Scripts.
- **TestFarm::Exec** provides primitives for sequencing Test Cases. It is mainly used internally by the TestFarm execution environment. It also exports the ManualInput() function used for interrupting a test script in order to give hand to the user for doing Manual Testing operations.
- **TestFarm::Trig** provides a set of functions to manage event triggers. This is a key component of the TestFarm platform, since it allows the Test Scripts and all other library modules to deal with Test Interface synchronization.

8.3.2 Log File Management

LocalLog	PERL Function

Synopsis

```
use TestFarm::Engine;

$FileName = LocalLog();
$FileName = LocalLog("FileName");
```

Description

If a file name is specified as an argument, this changes the the local log file name. If no argument is given, no change is performed. See section 7.3.4 for a description of the local log file.

Returned value

The name of the local log file. See section 7.3 for some explanations about log files.

GlobalLog	PERL Function

Synopsis

```
use TestFarm::Engine;

$FileName = GlobalLog();
$FileName = GlobalLog("FileName");
```

Description

If a file name is specified as an argument, this changes the the global log file name. If no argument is given, no change is performed. See section 7.3.4 for a description of the local log file.

Returned value

The name of the global log file. See section 7.3 for some explanations about log files.

8.3.3 Test Report Directory

ReportDir	PERL Function

Synopsis

```
use TestFarm::Engine;

$Directory = ReportDir();
```

Description

This function returns the name of the directory in which the log files and test report documents are stored. This can be useful to store files that are attached to the test report, like screenshots, graphics and any other data or illustrations.

Returned value

The name of the test report directory related to the test suite which execution is in progress.

8.3.4 Test Case Identification

TestCase	PERL Function

Synopsis

```
use TestFarm::Engine;
$CaseName = TestCase();
```

Returned value

This function returns the name of the current Test Case. This is the name of the Test Case node in which the calling Test Script is being executed. Refer to the *TestFarm Test Suite Reference Manual* [1] for more details about the concept of Test Case.

8.3.5 Test Case interruption for Manual Testing

ManualInput	PERL Function

Synopsis

```
use TestFarm::Exec;
($Verdict, $Reply) = TestFarm::Exec::ManualInput();
($Verdict, $Reply) = TestFarm::Exec::ManualInput("Question");
```

Description

Calling this function within a test script will cause it to halt, waiting for a manual action from the operator. If argument `"Question"` is given, the text is displayed in the output dump area of the TestFarm Runner. Then, the input area is shown, and the operator should give a response by entering a text and clicking a verdict button. This to causes the `ManualInput()` function to return the result entered by the operator, and the test script execution resumes.

Returned value

This function returns a tuple containing the response given by the operator:
`$Verdict` is the verdict clicked by the operator.
`$Reply` is the text entered by the operator.

8.3.6 Managing Synchronization with Triggers

Introduction to Triggers

Triggers are the core elements of the synchronization mechanism. Basically, a trigger is defined with a **regular expression**, the identifier of a source **Test Interface**, and an **occurrence counter**. The principle is simple: each time the Test Interface replies a message matching the regular expression, the occurrence counter is incremented. Sections explains how to create and delete triggers. Section presents some functions that manipulate the occurrence counter.

When created, a trigger is given a unique identifier, which is used by the various trigger manipulation functions. This identifier is also useful to write the trigger Boolean expressions needed by the synchronization functions.

Once triggers are defined, it becomes possible for the Test Script to **wait until a trigger** or **a combination of triggers** is satisfied. This combination is defined in a **Boolean expression** (section) using OR, AND and NOT operators. It is even possible to specify a condition on the occurrence counter for each trigger in the Boolean expression: this may be useful to wait for a complex event such as « *Wait until the LCD has displayed "Please*

© TestFarm 2006-2012

enter PIN code" twice ». Refer to section to understand how to wait for triggered events and to retrieve some information on them.

As an additional feature, the `TestFarm::Trig` module provides and convenient way to manipulate triggers: the **Magic Trigger Variables**. This provides the possibility to create a PERL variable tied by a trigger. Setting this variable changes the trigger regular expression. When the PERL interpreter unalocates such a variable, the trigger is automatically deleted, thus exempting the Test Script from doing it "manually". As a consequence, when one creates a local Magic Trigger Variable using the "`my`" PERL qualifier, it is automatically deleted when the execution goes outside the program block in which the variable is declared. Section 8.3.7 gives some details on how to implement Magic Trigger Variables.

Trigger Expressions

The trigger wait functions (sections and 8.3.7) require two parameters:

- A **Boolean expression** containing one or more trigger name(s). This Boolean expression may combine several triggers using the logical operators listed in the table below.
- A **timeout value**, specifying the maximum amount of time the trigger(s) should be waited for.

The wait function blocks until the result of the Boolean expression is TRUE, or until the timeout expires.

The supported logical operators of a trigger expression are:

Symbol	*Operator*	*Priority*	*Example*
`|`	Logical OR	1 (lowest)	`TRIG1	TRIG2`
`&`	Logical AND	2	`TRIG1 & TRIG2`
`!`	Logical NOT (unary operator)	3 (highest)	`!TRIG1`
`()`	Sub-expression prioritizing		`!(TRIG1 & TRIG2)`

Each operator has a priority that defines the rules of associativity in complex expressions, but it is also possible to use parenthesis to force priorities and/or to construct sub-expression.

Examples:

- "`TRIG1 & TRIG2 | TRIG3`" and "`(TRIG1 & TRIG2) | TRIG3`" are equivalent.
- "`(TRIG1 | TRIG2) & (TRIG3 | !TRIG4)`"

By default, an operand is TRUE if the occurrence counter of the trigger is greater than zero. However, it is possible to define an occurrence qualifier for each operand by suffixing the trigger identifier with "`*N`" (like in "`MYTRIG*2`"). In such a form, the operand is TRUE if the occurrence counter of the trigger is equal to or greater than *N*.

Examples:

- "`MYTRIG`" and "`MYTRIG*1`" are equivalent.
- "`TRIG1 & TRIG2*3`" is TRUE if trigger TRIG1 has been matched one or more times AND trigger TRIG2 has been matched three or more times.

Creating and Deleting Triggers

TrigDef	**PERL Function**

Synopsis

```
use TestFarm::Trig;

$TrigId = TrigDef($InterfaceObj, "TrigId", "Regex");
$TrigId = TrigDef("InterfaceId", "TrigId", "Regex");
$TrigId = TrigDef(undef, "TrigId", "Regex");
```

Description

Define (i.e. create) a trigger that listen for replies coming from a Test Interface, and matching the regular expression *Regex*. The Test Interface could be specified either by its identifier *InterfaceId* or by its object instance variable `$InterfaceObj`: the two forms are equivalent.

If the Test Interface argument is `undef`, the trigger listens for all Test Interfaces. This should be avoided as much as possible, since it consumes a lot of system resource.

Returned value

This function returns the name of the newly created trigger.

TrigUndef	PERL Function

Synopsis

```
use TestFarm::Trig;

TrigUndef("TrigId", …);
```

Description

Undefine (i.e. delete) a trigger that was previously created with the `TrigDef` function or the `TrigDef` Test Interface object method (described in section 8.4.2). This is a good practice to undefine triggers that are not useful any more, in order not to overload the Test Engine event manager. If you want the triggers to be automatically undefined when leaving a function or a program block, you can use the Magic Trigger variables described in section 8.3.7.

Managing the Trigger Occurrence Counter

TrigClear	PERL Function

Synopsis

```
use TestFarm::Trig;

TrigClear("TrigId", …);
```

Description

Clear the occurrence counter of one or more triggers which names are listed as the function arguments.

TrigCount	PERL Function

Synopsis

```
use TestFarm::Trig;

$Count = TrigCount("TrigId");
```

Returned value

This function returns the value of the occurrence counter of trigger *TrigId*.

Waiting for Triggers

TrigTimeout PERL Function

Synopsis

```
use TestFarm::Trig;

$OldValue = TrigTimeout("TimeValue");
```

Description

Sets the default wait timeout to *TimeValue*. The default wait timeout is used by the trigger wait functions when their *TimeValue* arguments are omitted (functions `TrigWait`, `TrigWaitInfo`, or their counterparts methods `Wait`, `WaitInfo` in Magic Trigger Variables).

- The time value must be a positive integer.
- By default, the time value is in milliseconds, but the argument may be suffixed with a time unit qualifier “`ms`”, “`s`”, “`min`” or “`h`”, which means that the time value is expressed respectively in milliseconds, seconds, minutes or hours (e.g. “`10s`”, “`5min`”, etc.).

Returned value

This function returns the previous timeout value.

TrigWait PERL Function

Synopsis

```
use TestFarm::Trig;

$TrigList = TrigWait("TrigExpr");
$TrigList = TrigWait("TrigExpr", "TimeValue");
```

Description

Wait until the trigger expression `Trig`*`Expr`* is TRUE, or the wait timeout expires.

If argument *TimeValue* is given, the wait timeout is set to this value. If not, the default wait timeout value set by the `TrigTimeout` function is used.

Returned value

An empty string if the wait timeout expired.

A space-separated list of the triggers having a TRUE value in the expression, as explained in section .

TrigInfo PERL Function

Synopsis

```
use TestFarm::Trig;

$Info = TrigInfo("TrigId");
```

DESCRIPTION

When a trigger is matched, this function returns the Test Interface reply that caused the trigger to be matched.

RETURNED VALUE

An empty string if the trigger was not matched.

The latest Test Interface reply (as presented in section 5.2.3) if the trigger has been matched at least once.

TrigWaitInfo **PERL Function**

SYNOPSIS

```
use TestFarm::Trig;

$Info = TrigWaitInfo("TrigId");
$Info = TrigWaitInfo("TrigId", "TimeValue");
```

DESCRIPTION

This function is a simplified combination of `TrigWait` and `TrigInfo`: it waits for the trigger *TrigId* within a timeout of *TimeValue* (or the default wait timeout if omitted), and returns the latest Test Interface reply that caused the trigger to be matched.

RETURNED VALUE

An empty string if the trigger was not matched.

The latest Test Interface reply (as presented in section 5.2.3) if the trigger has been matched at least once.

8.3.7 Magic Trigger Variables

Creating and Destroying a Magic Trigger Variable

TrigVarDef **PERL Function**

SYNOPSIS

```
use TestFarm::Trig;

$TrigId = TrigVarDef($InterfaceObj or "InterfaceId", $TrigVar);
$TrigId = TrigVarDef($InterfaceObj or "InterfaceId", $TrigVar, "Regex");
```

DESCRIPTION

Create a Magic Trigger Variable `$TrigVar` tied to a trigger. The trigger identifier is automatically generated and is returned by the function and by reading the so created tied variable.

The trigger listens to replies coming from the Test Interface indicated in the third argument. The Test Interface could be specified either by its identifier *InterfaceId* or by its object instance variable `$InterfaceObj`: the two forms are equivalent.

If the argument *Regex* is present, this sets the trigger regular expression. If omitted, the regular expression can be set later by setting the variable `$TrigVar`. Setting `$TrigVar` again will update the trigger regular expression.

When PERL frees `$TrigVar` (which occurs if `$TrigVar` is declared as a local “`my`” variable in a module or a program block), the trigger is automatically deleted.

Returned value

The function return the trigger identifier, which can be also retrieved by reading `$TrigVar`.

TrigVarUndef — PERL Function

Synopsis

```
use TestFarm::Trig;
TrigVarUndef($TrigVar);
```

Description

Delete the trigger tied to the magic trigger variable `$TrigVar`. Afterwards, the variable is undefined, and is not a Magic Trigger Variable any more.

Note that this is reversible behaviour: setting `"$TrigVar = undef"` has the same effect.

Using a Magic Trigger Variable

The basic usages of a Magic Trigger Variable are illustrated in Figure 8.2.

Writing to a Magic Trigger Variable defines or redefines the regular expression of the associated trigger: this provides a convenient way to set and update a trigger.

Reading a Magic Trigger Variable returns the trigger identifier, which can be used as an argument for the classical trigger management functions described in section 8.3.6. As a consequence, referencing a Magic trigger Variable within a trigger expression will then automatically expend the trigger identifier.

```
# Define a Magic Trigger Variable that listens to USCS
TrigVarDef(my $trig, $USCS);

# Set trigger regex to catch command completion
$trig = "CWT +Character Waiting Time";

# Send USCS command
$USCS->cwt($args);

# Wait for command completion within 2 seconds
my $info = TrigWaitInfo("$trig", "2s");
unless ( $info ) {
  print "IN_VERDICT Unable to get CWT value\n";
  return 0;
}
```

Figure 8.2: Using a Magic Trigger Variable

Manipulating a Magic Trigger Variable

TrigVarId — PERL Function

Synopsis

```
use TestFarm::Trig;
$TrigId = TrigVarId($TrigVar, "TrigId");
```

Description

Change the trigger identifier associated to the magic trigger variable `$TrigVar`, which is useful if you want to override the trigger identifier that was automatically given when the

trigger variable was created. Since there is a risk of conflicts between trigger identifiers, this function must be used with care.

Returned value

The function return the trigger identifier, which can be also retrieved by reading `$TrigVar`.

TrigVarPrefix **PERL Function**

Synopsis

```
use TestFarm::Trig;
TrigVarPrefix($TrigVar, "String");
```

Description

Define a prefix that will be prepended to the regular expression of the magic trigger variable `$TrigVar`.

For example, when a prefix “`MYPREFIX `” is defined, writing “`MY PATTERN`” to `$TrigVar` sets the trigger regular expression to “`MYPREFIX MY PATTERN`”. Rewriting later `$TrigVar` with “`HELLO WORLD`” redefines the regular expression as “`MYPREFIX HELLO WORLD`”.

8.4 Writing a Test Interface Module

8.4.1 Test Interface Module Architecture and Environment

The Test Interface library is a collection of PERL modules called **Test Interface Modules**. The role of such a module is to provide a set of PERL functions that communicate with the Test Interface control program through the Test Engine (as presented in section 5.1). This allows to use the various equipments through function calls, rather than a command/reply stream.

In the Test Execution Environment, a Test Interface is materialized as an **object instance**. As a consequence, a Test Interface module must implement such an object class, which provides a set of functions (or methods) that communicate with the Test Interface control program or command interpreter.

The TestFarm platform includes a PERL module `TestFarm::Interface`. This module implements a base object class all Test Interface Modules should inherit. This section explains how to efficiently write a Test Interface Module based on this `TestFarm::Interface` base class.

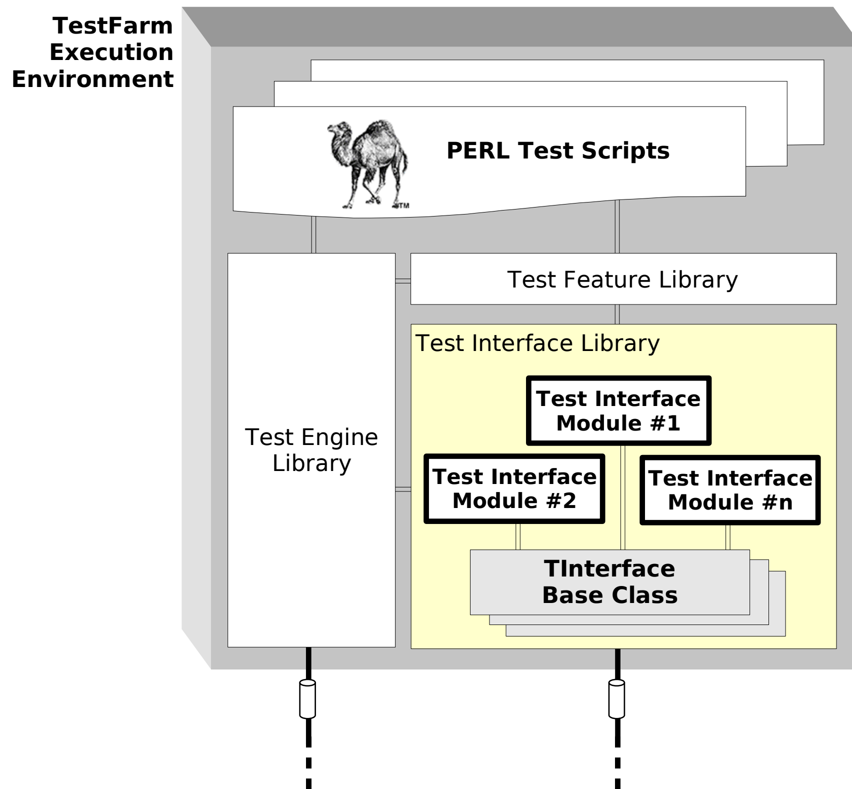


Figure 8.3: The Test Interface Library in the Test Execution Environment

8.4.2 The Test Interface Base Class

This section gives a detailed description of the Test Interface base class provided by the "**`TestFarm::Interface`**" module. A Test Interface module should inherit this base class using the standard PERL inheritance mechanisms described in section 8.4.2.

new	**PERL Object Constructor**

SYNOPSIS

```
use TestFarm::Interface;

$InterfaceObj = TestFarm::Interface->new(\%TEMPLATE, "Addr");
$InterfaceObj = TestFarm::Interface->new(\%TEMPLATE, "Method://Addr");
$InterfaceObj = TestFarm::Interface->new(\%TEMPLATE, "InterfaceId", "Addr");
$InterfaceObj = TestFarm::Interface->new(\%TEMPLATE, "InterfaceId", "Method://Addr");
```

DESCRIPTION

This method is the object creator invoked when a Test Interface object is instanciated. The constructor of a Test Interface module should invoke this method in respect of the PERL object inheritance mechanism (see section 8.4.3).

The caller has to provide a reference to a short configuration template stored in a hash table identified as "`\%TEMPLATE`" in the synopsis. The Test Interface configuration template contains the following entries:

Key	Content	Example
DESCRIPTION	Short description of the Test Interface	"UF120 USB Interface"
TYPE	The type of Test Interface (one word). ☞ It is preferable not to set this entry, in which case the Test Interface type is determined from the name of the Test Interface module (e.g. "AUDIO" for module "Audio.pm").	"UF120"
COMMAND	Command prefix, used when the Test Interface is connected as a piped sub-process. ☞ When not set, the default value is determined from the lower-cased name of the Test Interface module (e.g. "audio" for module "Audio.pm").	"uf120 -sn"
FLAGS	Test Engine compliance flag (optional): ➩ Empty : Interface reply format is TestFarm-compliant (5.2.3). ➩ "**-noheader**": Interface reply format is not TestFarm-compliant (5.2.4).	

The arguments *InterfaceId*, *Method* and *Addr* are passed when the Test Interface object is instanciated by the START function of the Test Feature Library. They are respectively given the values of the eponymous INTERFACE XML attributes described in section .

RETURNED VALUE

The newly created PERL object instance. The newborn Test Interface will inherit from this object instance.

id — PERL Object Method

SYNOPSIS

```
use TestFarm::Interface;
$InterfaceId = $InterfaceObj->id();
```

RETURNED VALUE

Returns the Test Interface identifier as defined in the XML System Definition file.

type — PERL Object Method

SYNOPSIS

```
use TestFarm::Interface;
$InterfaceType = $InterfaceObj->type();
```

RETURNED VALUE

Returns the Test Interface type, which in most cases is the uppercased name of the Test Interface module.

open — PERL Object Method

Synopsis

```
use TestFarm::Interface;
$InterfaceObj->open();
```

Description

Establish connection between the Test Interface and the Test Engine. This method should normally be invoked from the Test Interface object constructor.

This method exploits some parameters previously passed to the object constructor (see method "new" above):

Parameter	Description
Command	The COMMAND entry of the Test Interface configuration template.
Method	The argument *Method*. Contains the method attribute value of the Test Interface declaration in the XML System Definition file (section 8.5)
Addr	The argument *Addr*. Contains the addr attribute value of the Test Interface declaration in the XML System Definition file (section 8.5)

The type of connection depends on the access method specified in parameter Method:

Value of *Method*	Type of Connection	Description
PROC or none	Piped sub-process	The Test Engine executes the command "*Command* *Addr*" as a sub-process and redirects its standard input/output to use them as the command/reply stream.
TCP	TCP/IP connection	The Test Engine establishes a TCP/IP connection to *Addr*, which should have the format "*IPAddress*:*PortNumber*" (e.g. "192.168.1.2:8888").
SERIAL	Serial link	The Test Engine opens the serial link "*Addr*", which contains the device name of the serial port (e.g. "/dev/ttyS0").

close — PERL Object Method

Synopsis

```
use TestFarm::Interface;
$InterfaceObj->close();
```

Description

Close connection between the Test Interface and the Test Engine. This method is automatically called when a Test Interface object instance is destroyed.

trig_id — PERL Object Method

Synopsis

```
use TestFarm::Interface;
$TrigId = $InterfaceObj->trig_id();
$TrigId = $InterfaceObj->trig_id("Pref");
```

Description

Creates a trigger identifier that is guaranteed to be unique and not conflicting with another previously created trigger.

The trigger identifier is constructed with a concatenation of *Pref*, followed by the Test Interface identifier, followed by a number (incremented each time the function is called). If argument *Pref* is omitted, the default prefix “`TRIG`” is used.

For example, let us consider an interface identified “`DLOG`” and instanciated in variable `$DLOG`: calling “`$DLOG->trig_id()`” several times will return successively “`TRIGDLOG1`”, “`TRIGDLOG2`”, etc.

Returned value

The newly created trigger identifier.

command — PERL Object Method

Synopsis

```
use TestFarm::Interface;

$Info = $InterfaceObj->command("Command");
$Info = $InterfaceObj->command("Command", "Regex");
$Info = $InterfaceObj->command("Command", "Regex", "TimeValue");
```

Description

This method sends the command *Command* to the Test Interface.

If argument *Regex* is omitted, the method returns immediately. If *Regex* is specified, the method returns after the Test Interface has sent a reply matching *Regex*. *Regex* is a trigger regular expression (see functions `TrigDef` and `TrigVarDef` in section 8.3).

If *TimeValue* is omitted, the method waits within default timeout value (see function `TrigTimeout` in section). If *TimeValue* is specified, the method waits within this amount of time.

⇨ This method is typically called from the action functions described in section 8.7.5. It may also be called directly from a test script, but it is not recommended as it breaks the abstraction levels of a well architectured testing system.

Returned value

The Test Interface reply that caused the method to return.
`undef` if no reply is awaited or if a timeout occurred.

sync — PERL Object Method

Synopsis

```
use TestFarm::Interface;

$InterfaceObj->sync();
$InterfaceObj->sync("TimeValue");
```

Description

This method implements a Test Interface synchronization barrier. It works only if the Test Interface supports the command “echo”, which behaviour is described in the example of section 5.4.

The operations performed by this method are:

© TestFarm 2006-2012

- Ask the Test Interface to echo a message.
- Wait for this message to be echoed within a delay of *TimeValue* (20 seconds by default if omitted).

RETURNED VALUE

The synchronization message, or `undef` if the synchronization barrier timed out.

8.4.3 Bringing a Test Interface Module to Life

In order to work properly, a Test Interface module has to define a configuration **Template**, an object **Constructor**, and a few **Methods** that define the basic operations one can perform on the Test Interface. An example of these three kinds of resource is proposed Figure 8.4.

TEMPLATE	PERL Hash Table

The Test Interface configuration template is a hash table containing some characterization entries, as defined in the description of the `TestFarm::Interface` base class constructor (section 8.4.2).

This template is passed as an "Identity Card" when creating the `TestFarm::Interface` instance the Test Interface inherits from.

IMPLEMENTATION

```
my %TEMPLATE = (
  'DESCRIPTION' => "…",     (required)
  'TYPE'        => "…",     (optional, default is upper-cased module name)
  'COMMAND'     => "…",     (optional, for piped sub-process only , default is lower-cased module name)
  'FLAGS'       => "…",     (optional, default is for TestFarm-compliant devices)
);
```

EXAMPLES

❶ Example of configuration template for the UR111 USB interface, which is natively a TestFarm-compliant device, operated as a piped subprocess.

```
my %TEMPLATE = (
  'DESCRIPTION' => 'UR111 USB Interface',
);
```

❷ Example of configuration template for a Agilent 34401A multimeter connected to a serial port. This is a typical often used non TestFarm-compliant device.

```
my %TEMPLATE = (
  'DESCRIPTION' => 'HP34401A Multimeter',
  'FLAGS'       => '-noheader'
);
```

new	PERL Object Constructor

SYNOPSIS

```
use MyInterface;

$Object = MyInterface->new("Addr");
$Object = MyInterface->new("Method://Addr");
$Object = MyInterface->new("InterfaceId", "Addr");
$Object = MyInterface->new("InterfaceId", "Method://Addr");
```

DESCRIPTION

This method is Test Interface object creator (symbolized as *MyInterface* in the synopsis above). It should create an instance of the `TestFarm::Interface` base class and inherit from it. Afterwards, it may also ask the Test Engine to physically establish the connection with the Test Interface. As in any object constructor, some other interface setup or object initialisation actions may also be performed.

IMPLEMENTATION

```
sub new {
  my $proto = shift;
  my $class = ref($proto) || $proto;
  my $self = TestFarm::Interface->new(\%TEMPLATE, @_);     (Inherit TestFarm::Interface)
  $self->open();                                            (Establish connection)
  …                                                         (Put initialisation code here)
  return bless($self, $class);                              (Return new object instance)
}
```

```
##
## TestFarm Audio Interface
## Interface Library
##
## (C) Basil Dev 2006
##
## $Revision: 173 $
## $Date: 2006-07-26 17:41:27 +0200 (Wed, 26 Jul 2006) $
##

package TestFarm::Audio;
@ISA = qw ( TestFarm::Interface );

use TestFarm::Trig;
use TestFarm::Interface;

my %TEMPLATE = (
  'DESCRIPTION' => "TestFarm Audio Interface",
);


#
# new( [<Name>,] [(PROC|TCP|SERIAL)://]<Address> );
#
sub new {
  my $proto = shift;
  my $class = ref($proto) || $proto;
  my $self = TestFarm::Interface->new(\%TEMPLATE, @_);
  bless($self, $class);
  $self->open('.+Sound\ device\ opened.+', '3s');
  return $self;
}


sub sleep {
  my $self = shift;
  $self->command("sleep @_");
}
```

Figure 8.4: The beginning of a sample Test Interface module

8.5 WRITING AN XML SYSTEM DEFINITION

8.5.1 System Layout Philosophy

Let us keep in mind that a TestFarm system is build with a collection of **Test Interfaces** that implement **Test Features**. The Test Features are animated by the test engine, in order to execute a Test Suite. The purpose of a **System Definition File** is to perform a logical declaration of the system layout by declaring the Test Interfaces and the Test

Features they implement. This file will then be used as an input for the **System Configuration Wizard** to generate the **Test Feature Library**.

The System Definition also contains some information on the hardware equipments used in the system, in order to provide a detailed description of the system profile. This includes equipment serial numbers, hardware versions, or any useful data that characterize the test equipments and the product under test. This kind of information may have several usages:

- To produce rigorous test reports showing a description of the testing system environment;
- To write flexible Test Scripts that are able to automatically adapt to the system layout by fetching system information from the Test Feature Library.
- To facilitate the maintenance operations if you have to manage several systems.

The basic item of a System Definition is the Test Feature, which is implemented by a Test Interface. It is important to understand those two concepts: A Test Interface is an equipment, whereas a Test Feature is a logical function it implements. A Test Interface may implement one or more Test Features. A Test Feature is always attached to a Test Interface: this clearly appears in the structure of a System Definition File presented in 8.5.2.

As an example, let us consider a board containing several relays, which are used to actuate a reset button, a validation button, and a smart-card insertion switch. In this configuration, the Test Interface is the relay board, and the Test Features are the three functions it implements: “Reset”, “Validate” and “Insert”.

As another example, let us image a system performing physical measurements. We use some sensors connected to a multi-channel analogue acquisition board to measure temperature and pressure. In this context, the Test Interface is the acquisition board, whereas the Test Features are “Temperature” and “Pressure”.

8.5.2 Structure of the System Definition File

The System Definition is declared within an **XML** file. XML is an acronym for “*eXtensible Markup Language*”. It is a very common format inherited from the World Wide Web. This format is used to describe data in a flexible and structured manner. The structure of the XML System Definition is illustrated in Figure 8.5. This diagram is divided into three columns.

The leftmost column shows the configuration tree structure, in which the nodes are represented in square boxes. A node may be accompanied with a punctuation mark “**?**”, “*****” or “**+**”. If you are accustomed with the XML syntax, you may know what this means:

- No mark means that the node is unique and mandatory;
- A “**?**” mark indicates that there may be zero or one node;
- A “*****” mark indicates that there may be zero or several nodes;
- A “**+**” mark indicates that there should be one or several nodes.

The middle column of the diagrams shows the role of each node, within a small banner.

If a node needs some attributes, they are indicated in a table at the rightmost column of the diagram. Please be aware that some attributes are mandatory.

© *TestFarm 2006-2012*

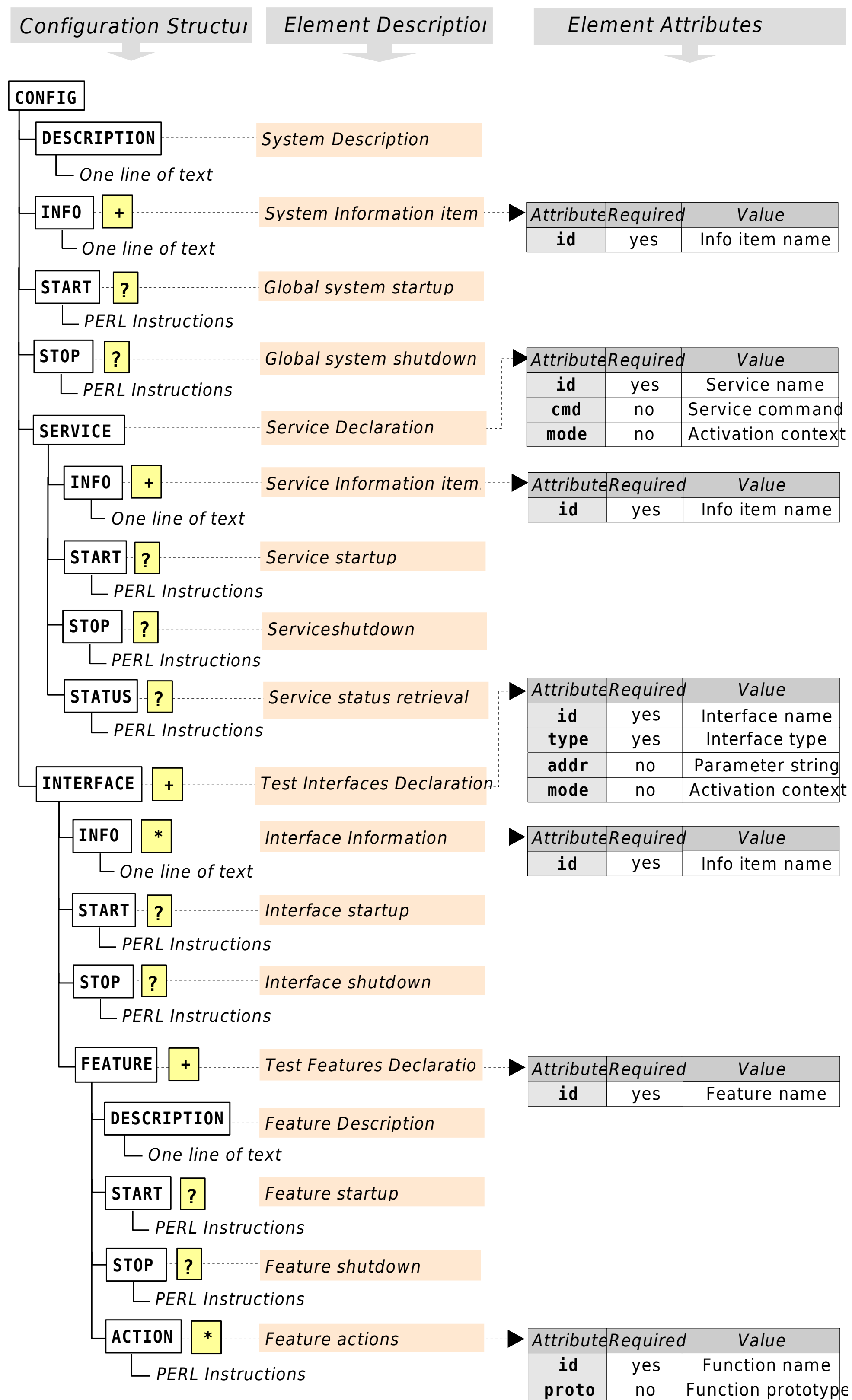


Figure 8.5: Structure of a XML System Definition File

This structure is declared in a DTD file (Document Type Definition) included in the TestFarm platform, thus allowing to edit the System Configuration file using an XML editor.

© TestFarm 2006-2012

An XSL file (eXtensible Stylesheet Language) also brings the possibility to view the XML file in a formatted manner using an XML-compliant web navigator (such as Mozilla, Figure 8.6).

TestFarm System Configuration

System Information

Info	Content
System Description	Basil Dev TestFarm Training Kit
Serial Number	001

Services

Service	Command	Mode	Info
LCD	lcd-screen -alpha -g 20x2		

Interfaces, Features and Actions

Interface	Type	Address	Mode	Info	Features
UF120	TestFarm::UF120			version:	Display
UR111	TestFarm::UR111			version:	Keypad Power Reset Relay
AUDIO	TestFarm::Audio	-f 16000		version:	Buzzer ~~PLL~~
TKIT	TKitTest	/dev/ttyUSB0			UART

Feature	Mode	Description	Interface	Actions
Buzzer		Buzzer Frequency detection	AUDIO	BuzzerDetect
Display		LCD Text display	UF120	DisplayClock DisplayTrig
Keypad		Keypad actions	UR111	KeyMap Dial KeyPress KeyTrig
~~PLL~~	DISABLE	PLL Tone generation	AUDIO	~~PLLToneOn~~ ~~PLLToneOff~~
Power		Power Supply switch	UR111	PowerSwitch
Relay		Relay Activity	UR111	RelayTrig RelayShow
Reset		Hardware reset	UR111	ResetSwitch Reset
UART		UART Monitor	TKIT	TKitCall

Figure 8.6: Viewing the System Definition File using a web navigator

8.5.3 Content of the System Definition File

File header

The System Configuration file must begin with some references to the TestFarm standard DTS and XSL files. This is a classical XML header telling how to control the syntax and the viewing format of the system configuration file.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="/opt/testfarm/lib/config.xsl"?>
<!DOCTYPE CONFIG SYSTEM "/opt/testfarm/lib/config.dtd">
```

Configuration Elements

The System Definition file is made of elements gathered into the structure described in 8.5.2. An element is identified with an XML tag, optionally completed with some attributes. This section gives some details on the role and content of these elements.

`CONFIG` — XML Root Element

DESCRIPTION

The root element of the document

ATTRIBUTES

none

CONTENT

Children XML elements: `DESCRIPTION`, `INFO`**+**, `START`**?**, `STOP`**?**, `INTERFACE`**+**.

`SERVICE` — XML Element

DESCRIPTION

Declaration of a System Service. System Services are described in chapter Error: Reference source not found.

ATTRIBUTES

- **`id`** is the System Service identifier.
- **`cmd`** (optional) indicates the command to be launched if the service is command-oriented.
- **`mode`** (optional) specifies in which environment the service is activated:

mode	*Description*
ANY (default)	The service is activated in any environment
AUTO	The service is activated in the Test Suite Execution environment only
MANUAL	The service is activated in the Manual User Interface only
DISABLE	The service is never activated

CONTENT

Children XML elements: `INFO`*****, `START`**?**, `STOP`**?**, `STATUS`**?**.

`INTERFACE` — XML Element

DESCRIPTION

Declaration of a Test Interface.

ATTRIBUTES

- **`id`** is the Test Interface identifier.
- **`type`** indicates the type of Test Interface, which is the name of the Test Interface module from which the interface will be isntanciated.
- **`addr`** (optional) defines a connection method and address, which may be a device node name, a network address, etc. Format is: ***`METHOD://ADDRESS`***.

© TestFarm 2006-2012

Type of Test Interface	METHOD	ADDRESS	Example
Piped subprocess (default)	**PROC**	Subprocess command parameters	"-sn A1:0002"
Serial link	**SERIAL**	Serial link device name	"SERIAL:///dev/ttyS0"
TCP/IP connection	**TCP**	IP address and port number of the remote endpoint	"TCP://192.168.1.2:8888"

- **mode** (optional) specifies in which environment the interface is activated:

mode	Description
ANY (default)	The interface is activated in any environment
AUTO	The interface is activated in the Test Suite Execution environment only
MANUAL	The interface is activated in the Manual User Interface only
DISABLE	The interface is never activated

CONTENT

Children XML elements: INFO*****, START**?**, STOP**?**, FEATURE**+**.

FEATURE XML Element

DESCRIPTION

Declaration of a Test Feature.

ATTRIBUTES

- **id** is the Test Feature identifier.

CONTENT

Children XML elements: DESCRIPTION, START**?** , STOP**?**, ACTION*****.

INFO XML Element

DESCRIPTION

Informational text for documentation and administration purpose. Section 8.7.1 details how a Test Script can use the Test Feature Library to retrieve these information resources.

ATTRIBUTES

- **id** is the Element identifier.

CONTENT

Each element should contain one text line. Additional lines are ignored.

- If the parent element is CONFIG,
  the INFO elements contain **System Information Items**. Empty elements are ignored.
- If the parent element is SERVICE,
  the INFO elements contain **Service Information Items**. Empty elements are ignored.
- If the parent element is INTERFACE,
  the INFO elements provide **Interface Information Items**. If such an element is empty, the information item is retrieved dynamically by invoking the method id of the Test Interface module. If the function is absent or returns nothing, no information item is produced.

DESCRIPTION XML Element

Description

Description text of a CONFIG or FEATURE element. This element is for documentation purpose only.

Attributes

none

Content

This element should contain one text line. Additional lines are ignored.

START XML Element

Description

This element defines the system start-up procedures, which are encoded in the START and START_SERVICE functions of the Test Feature library. The encoding of this function is detailed in section 8.7.4.

Attributes

none

Content

PERL Instructions.

- If the parent element is CONFIG,
  the content is dumped into the BEGIN function of the Test Feature Library module.
- If the parent element is SERVICE,
  the content is dumped into the service startup function of the Test Feature Library module.
- If the parent element is INTERFACE or FEATURE,
  the local variable **$I** is available for convenience: it points to the object instance of the current/parent Test Interface.

STOP XML Element

Description

This element defines the system shutdown procedures, encoded in the STOP and STOP_SERVICE functions of the Test Feature library. The encoding of this function is detailed in section 8.7.4.

Attributes

none

Content

PERL Instructions.

- If the parent element is CONFIG,
  the content is dumped into the END function of the Test Feature Library module.

© TestFarm 2006-2012

- If the parent element is `SERVICE`,
  the content is dumped into the service shutdown function of the Test Feature Library module.
- If the parent element is `INTERFACE` or `FEATURE`,
  the local variable **`$I`** is available for convenience: it points to the object instance of the current/parent Test Interface.

STATUS XML Element

Description

This element defines the status retrieval function for a SERVICE element.

Attributes

none

Content

PERL Instructions.

ACTION XML Element

Description

Declaration of an Action function. See details in 8.7.5.

Attributes

- **`id`** is the Action identifier, which is also the name of the action function in the Test Feature Library.
- **`proto`** (optional) is the PERL function prototype.

Content

PERL Instructions.
The local variable **`$I`** is available for convenience: it points to the object instance of the parent Test Interface.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="/opt/testfarm/lib/config.xsl"?>
<!DOCTYPE CONFIG SYSTEM "/opt/testfarm/lib/config.dtd">

<!-- $Revision: 230 $ -->
<!-- $Date: 2006-08-26 12:53:59 +0200 (Sat, 26 Aug 2006) $ -->

<CONFIG>
  <DESCRIPTION>
    Basil Dev TestFarm Training Kit
  </DESCRIPTION>
  <INFO id="Serial Number">
    001
  </INFO>
  <SERVICE id="LCD" cmd="alpha2x20_display"/>
  <INTERFACE id="LCD" type="TestFarm::UF120" addr="-sn A1:0002">
    <INFO id="version" />
    <START>
      return 0;
    </START>
    <STOP>
      $I->sync('5s');
    </STOP>
    <FEATURE id="Display">
      <DESCRIPTION>LCD Text display</DESCRIPTION>
      <START>
        return 1 if $I->link('DISPLAY', 'FIFO');
        $I->connect('DISPLAY', 'proc:///opt/testfarm/bin/alpha2x20 -draw -alias 0x20=0x5F -f 40 -dp-clear +cs-conf');
        DisplayClock(1);
        return 0;
      </START>
      <STOP>
        DisplayClock(0);
        $I->unlink('DISPLAY');
      </STOP>
      <ACTION id="DisplayClock" proto="$">
        if ( $_[0] ) {
          $I->reset();
          $I->clock('+0', 'rw=0');
        }
```

Figure 8.7: A Sample XML System Definition file (beginning)

© TestFarm 2006-2012

8.6 Generating the Test Feature Library

The TestFarm System Configuration Wizard is a tool that parses an XML System Definition file to generate the Test Feature Library. This operation should be performed by the system integrator. The generated Test Feature Library is a PERL module that will be loaded from the Test Scripts to call the various functions defined in the Configuration Elements of the XML file. Please refer to the TestFarm User's Manual [2] for details about how to use the System Configuration Wizard.

The Test Feature Library should not be linked to the Test Suite, but rather to the Test System it is executed on: this makes the Test Feature Library a kind of resource abstraction layer for the Automated Testing System. In respect of this approach, it is a good practice to store the Test Feature Library beside the XML file it comes from, in a dedicated directory called the TestFarm Configuration Directory. By default, the Configuration Directory is "**/var/testfarm/lib**", but it may be changed by setting the environment variable **TESTFARM_CONFIG**.

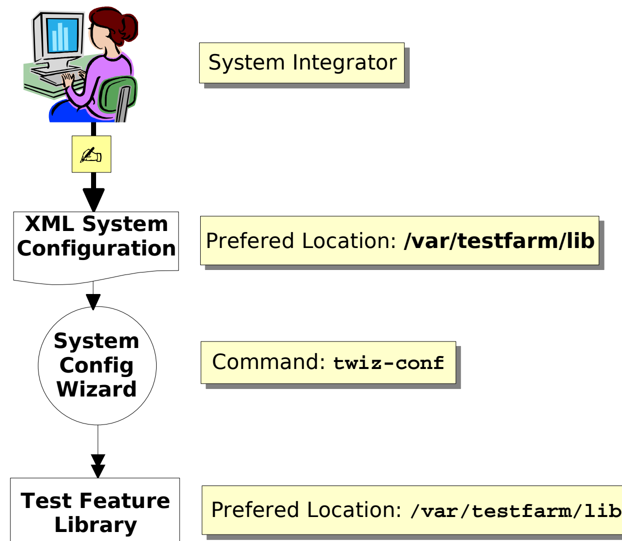


Figure 8.8: Using the System Configuration Wizard

8.7 Using the Test Feature Library

This details the content of the Test Feature library, generated from the XML System Definition file using the System Configuration Wizard (section 8.6).

The Test Feature library is actually a PERL module that provides the following resources:

- A list of PERL "use" statements for importing all the required modules from the Test Engine library and the Test Interface library.
- Information about system components and the features they provide (section 8.7.1);
- Test Interface object instance variables (section 8.7.2);
- Start-up and shutdown functions (section 8.7.4);
- Action functions (section 8.7.5).

8.7.1 System Information Resources

All the information and description elements of the System Definition file are accessible in the Test Feature Library: they are gathered in three hash tables **%INFO**, **%FEATURE** and **%ACTION**. As an additional and convenient resource, calling the function **INFO** will output all these data in order to display them in the header of the HTML test report.

%INFO	**PERL Hash Table**

DESCRIPTION

The table %INFO contains the INFO data of the XML System Definition File. All System and Interface Information Items are gathered into a unique %INFO table.

SYSTEM INFORMATION ITEMS

- The key of a table entry is the information identifier itself (XML attribute id of the INFO element).
- Each table entry contains the first text line of the INFO element it comes from. The other lines are ignored.
- An empty INFO element is ignored (no entry table entry is created).

INTERFACE INFORMATION ITEMS:

- The key of an Interface Information Item is constructed with three words: the prefix “Interface”, the interface identifier (XML attribute id of the INTERFACE element) and the information identifier (XML attribute id of the INFO element). Examples: “Interface UR111 version”, “Interface UF120 serial”, etc.
- Each table entry contains the first text line of the INFO element it comes from. The other lines are ignored.
- If an INFO element is empty, the table entry data is retrieved dynamically by invoking the method id (XML attribute id of the INFO element) of the Test Interface object instance. If the method is absent or returns nothing, no table entry is created.

%FEATURE	**PERL Hash Table**

DESCRIPTION

The table %FEATURE is the list of all features declared in the system.

- The key of a table entry is the feature identifier (XML attribute id of the FEATURE element).
- Each table entry contains the feature description (provided in the child element DESCRIPTION of the feature).

%ACTION	**PERL Hash Table**

DESCRIPTION

The table %ACTION is the list of all actions provided by the system. Each action also produces an Action Function described in 8.7.4.

- The key of a table entry is the action identifier (XML attribute id of the ACTION element).
- Each table entry contains the parent Test Interface identifier (XML attribute id of the parent INTERFACE element).

INFO	**PERL Function**

DESCRIPTION

The function INFO prints the content of the %INFO hash table, so that it appears in the header of the HTML Test Report. This function does not require arguments.

- System Information Items are displayed first;
- Interface Information items are displayed by key alphabetical order.

8.7.2 Interface Instance Variables

As stated in section 8.1.1, a Test Interface is seen from the Test Script as an object instance, created from an object class provided by the Test Interface Library.

Each Test Interface declared in the System Definition File is materialized by an object instance, created by the Start-up function and stored in a PERL which identifier is the Test Interface identifier (XML attribute id of the parent INTERFACE element).

Let us take an example from the sample INTERFACE element (Figure 8.9). The Test Interface type is “TestFarm::UF120”, and the Test Interface identifier is “LCD”. This declaration will create an object instance in the PERL variable “$LCD” from the Test Interface library “TestFarm::UF120”. The interface attribute addr is passed as an argument to the object constructor, in order for the Test Interface module to know which hardware device the object instance will address.

```
<INTERFACE id="LCD" type="TestFarm::UF120" addr="-sn A1:0002">
  <INFO id="version" />
  <START>
    return 0;
  </START>
  <STOP>
    $I->sync('5s');
  </STOP>
  <FEATURE id="Display">
    <DESCRIPTION>LCD Text display</DESCRIPTION>
    <START>
      return 1 if $I->link('DISPLAY', 'FIFO');
      $I->connect('DISPLAY', 'proc:///opt/testfarm/bin/alpha2x20 -draw
      DisplayClock(1);
```

Figure 8.9: A sample INTERFACE element in a System Definition File

8.7.3 Service Start and Stop Functions

The START and STOP elements of the Service declaration (i.e. within the SERVICE elements) are encoded into the START_SERVICE and STOP_SERVICE functions in the Test Feature Library. These functions should not be invoked from the Test Scripts, because they are reserved for System startup and shutdown when the Test Suite user interface or the Manual user interface is launched (chapter Error: Reference source not found).

8.7.4 System Start and Stop Functions

The START and STOP elements of the System Definition File are respectively converted into START and STOP functions in the Test Feature Library (see section). These functions should normally not be called from a test script, because they are called automatically when a test suite starts and finishes.

START	PERL Function

DESCRIPTION

The **System Start Function**. This function does not require arguments.

It returns 0 on success, or 1 if a start operation failed (either for starting an Interface or a Feature).

The operations performed by the function START are:

<table>
<tr><td>Display message:</td><td colspan="2"><code>Start System <u>system name</u>:</code>
<code>Features: <u>list of features</u></code>
<code>Actions: <u>list of actions</u></code></td></tr>
<tr><td colspan="3">Execute the PERL instructions contained in the <code>CONFIG/START</code> element</td></tr>
<tr><td rowspan="5">For each Interface</td><td colspan="2">Display message: <code>Start Interface <u>id</u></code></td></tr>
<tr><td colspan="2">Create object instance of module <u>type</u> in variable $<u>id</u></td></tr>
<tr><td colspan="2">Update the <code>%INFO</code> hash table with static and dynamic information items.</td></tr>
<tr><td colspan="2">Execute the PERL instructions contained in the <code>INTERFACE[]/START</code> element</td></tr>
<tr><td>For each Feature</td><td>Execute the PERL instructions contained in the <code>FEATURE[]/START</code> element</td></tr>
</table>

STOP	**PERL Function**

Description

The **System Stop Function**. This function does not require arguments.

The operations performed by the function STOP are:

<table>
<tr><td rowspan="3">For each Interface (reverse order)</td><td>For each Feature (reverse order)</td><td>Execute the PERL instructions contained in the <code>FEATURE[]/STOP</code> element</td></tr>
<tr><td colspan="2">Execute the PERL instructions contained in the <code>INTERFACE[]/STOP</code> element</td></tr>
<tr><td colspan="2">Destroy the object instance in variable $<u>id</u>. Undefine variable $<u>id</u>.</td></tr>
<tr><td colspan="3">Execute the PERL instructions contained in the <code>CONFIG/STOP</code> element</td></tr>
</table>

8.7.5 Action Functions

The Concept

The features of an Automated Testing System are operated from the Test Scripts through Action Functions. Action Functions implement a sort a abstraction layer between the object methods of the Test Interface Library (which could are somehow input/output primitives), and the basic operations performed by the Test Scripts. Ideally, an Action Function should remain simple, even trivial (i.e. with only one instruction); it should address only one Test Interface, and should not perform complex operation such as synchronizing several Test Interfaces together.

While it is possible to access a Test Interface by invoking its object instance methods, it is preferable to do it by using Action Functions. This make the Test Scripts more clear, and avoid the Test Script writers to matter about object-oriented programming.

As a comprehensive example, let us consider a action named “`PowerSwitch`”. This action is physically implemented by driving the relay contact number 1 of the UR111 USB interface. The diagram below shows how it is encoded from the System Definition file to the Test Feature Library:

A fragment of XML System Definition File

The corresponding Action Function encoded in the Test Feature Library

```
<FEATURE id="Power">
  <DESCRIPTION>Power Supply switch</DESCRIPTION>
  <ACTION id="PowerSwitch">
    my $st = $_[0] || 0;
    $I->switch("1:$st");
  </ACTION>
</FEATURE>
```

```
sub PowerSwitch {
  my $I = $UR111;
  my $st = $_[0] || 0;
  $I->switch("1:$st");
}
```

Using an Action Function, a Test Script does not need to know how a Reset action is physically implemented: it just has to call the Reset Action Function. Furthermore, if the system wiring has to be changed for some reason, only the system integrator is concerned: (s)he needs to update the XML System Definition file, to recompile it using the System Configuration Wizard, and all the Test Suites will continue to work properly by transparently using the updated Action Function(s).

Prototyped Action Functions

If the ACTION XML element is declared with the “proto” attribute, the Action Function is encoded with a prototype declaration, as illustrated below:

```
<ACTION id="DisplayClock" proto="$">
  if ( $_[0] ) {
    $I->reset();
    $I->clock('+0', 'rw=0');
  }
  else {
    $I->clock('off');
  }
</ACTION>
```

```
sub DisplayClock($) {
  my $I = $UF120;
  if ( $_[0] ) {
  $I->reset();
  $I->clock('+0', 'rw=0');
  }
  else {
  $I->clock('off');
  }
}
```

9 The Test Report Generation System

9.1 The Test Suite result files

While a Test Suite is executed, the results are dumped in XML format to a Test Output file and a Test Log file. Let us recall the roles of these files:

- The Test Output file contains all the information about the Test Suite execution, including the structure of the Test Tree, the verdicts, the messages printed by the test scripts, etc.
- The Test Log file contains a record of the command/reply exchanges between the Test Interfaces and the Test Engine.

After the Test Suite is finished (or aborted), the Test Log file is appended to the Test Output file, thus gathering all the test execution results in a single file.

9.2 Getting a Test Report document

9.2.1 Report Generation

An HTML Test Report file is generated by applying an XML stylesheet to the Test Output file. By default, a stylesheet “`output.xsl`” (available as a standard TestFarm component) is applied. It is also possible to use another stylesheet by entering its name in the Report Option menu of the Test Suite User Interface: this provides the possibility to use a customized report layout.

When the Test Report is generated from the Test Suite User Interface, it is stored as a “`*.html`” file in the “`report`” subdirectory of the Test Suite workspace

- It is possible to generate a HTML Test Report from a Test Output file that is being written by a Test Suite in progress: the Test Report generation tool automatically recognizes and recovers uncomplete XML files.
- Optionally, the Test Log is converted in HTML format, thus bringing the possibility to browse the Test Log using a classical web browser, by clicking on an hyperlink in Test Report.

9.2.2 Direct Browsing

If the Test Output file is directly loaded from an XML-compliant web browser (such as Mozilla Firefox, or MS Internet Explorer), it is automatically formatted using the default TestFarm stylesheet. This is a conveniant way to view the test results without performing a Test Report generation operation.

This direct browsing method may not be possible when the Test Output file is being written by a Test Suite in progress. In such a situation the XML structure is not complete, so the browser may refuse to display it.

9.3 Test Report Layout

9.3.1 Overview

The standard Test Report layout contains 4 sections, which may configurated from the Report Options menu of the Test Suite User Interface:

1. The **Report Header** gives some information about the Test Suite references, the Test System configuration, and any information that might have been retrieved concerning the product under test (hardware configuration, firmware version, etc.).
2. The **Verdict Summary** shows some global statistics about the test results: the start and finish dates, the number of Scenarios and Test Cases.
3. The **Verdict List** is a table showing the results for each Scenario and Test Case, including the verdict, criticity, duration, description, and a short information message about the verdict.
4. The **Output Dump** displays all the messages that have been printed by the Test Script. If HTML Test Log generation is enabled, an hyperlink to these files are also present.

9.3.2 Scenario and Test Cases

In the Test Report, the Test Cases are grouped by Scenario. As defined in the *Test Suite Reference Manual* [1], a Scenario is a sequence containing Test Cases only, i.e. which does not contain any sub-sequences.

In a Test Report, the test results can be listed by Scenario. If this option is enabled, the Criticity, Verdict and Validation State of a Scenario is computed from the Test Cases it contains in respect of the following rules:

The Verdict of a Scenario:

- PASSED if it contains at least one PASSED and no FAILED/INCONCLUSIVE Test Case;
- FAILED if it contains at least one FAILED Test Case;
- INCONCLUSIVE if it contains at least one INCONCLUSIVE and no FAILED Test Case;
- SKIP if it contains only SKIPped Test Cases.

The Criticity of a Scenario:

If the verdict of the Scenario is *X*, the criticity is the highest criticity of the Test Cases that produced this verdict.

The Validation State of a Scenario:

The lowest validation state of the Test Cases it contains.

9.3.3 The Report Header

The Report Header is populated with the following information:

- The title “`TestFarm Report`”, followed by the text lines printed by the Test Scripts that are prefixed with “`IN_TITLE`” (like in « `print “IN_TITLE Hello World!\n”` »)
- The Test Suite name, which is the base name of the Test Suite file.
- The Test Suite description, which is defined by the `#$DESCRIPTION` directive of the Tree Root file.

- The Test Suite reference, which is defined by the #$REFERENCE directive of the Tree Root file.
- The Operator Name (optional), which is entered by the person who launches the Test Suite execution.
- The text lines printed by the Test Scripts that are prefixed with “IN_HEADER”.

9.3.4 The Verdict Summary

The Verdict Summary contains the following items:

- The start date, the finish date, and the Test Suite duration.
- The number of Scenarios and Test Cases:
  ⇨ Total number in the Test Suite;
  ⇨ Number of Executed Scenarios/Test Cases, and the percentage of total;
  ⇨ Number of Significant Scenarios/Test Cases (i.e. those that have been executed and that have a criticity), and the percentage of total.
- For each verdict PASSED, FAILED, INCONCLUSIVE, SKIP: the number of significant Scenarios and Test Cases, and the corresponding percentage.

9.3.5 The Verdict Lists

Verdict List by Scenario

The Verdict List by Scenario is a table giving the following information for each Scenario:

- The name of the Scenario (parenthesized if the validation state of the Scenario is not VALIDATED);
- The number of Test Cases in the Scenario;
- The criticity of the Scenario;
- The description of the Scenario, which is defined by the #$DESCRIPTION directive of its Tree Branch file.
- The duration of the Scenario;
- The Verdict of the Scenario.

Verdict List by Test Case

The Verdict List by Test Case is a table giving the following information for each Test Case:

- The name of the Test Case (parenthesized if the validation state of the Test Case is not VALIDATED).
- If the “Generate HTML Test Log” option is enabled, the name is followed by a star, which is an hyperlink to the Test Log section corresponding to the Test Case;
- The criticity of the Test Case;
- The description of the Test Case, which is defined by the #$DESCRIPTION directive of the Test Script.
- The duration of the Test Case;
- The verdict of the Test Case;
- The text lines printed by the Test Script that are prefixed with “IN_VERDICT”.
  A good practice is to print such messages when the Test Case is FAILED, in order to concisely give the reason of the failure.

© TestFarm 2006-2012

9.3.6 The Output Dump

The Output Dump shows all the text lines printed by the Test Scripts.

For each Test Case, the following information are shown:

- The name of the Test Case (parenthesized if the validation state of the Test Case is not VALIDATED);
- If the "Generate HTML Test Log" option is enabled, the name is followed by a star, which is an hyperlink to the Test Log section corresponding to the Test Case;
- The duration of the Test Case;
- The verdict of the Test Case;
- The complete validation state of the Test Case;
- The text lines printed by the Test Script, whatever the line prefix is.

9.4 Test Report Stylesheet Parameters

The Test Suite User Interface provides a Report Options menu. This menu updates an XML layout parameters file stored in the user configuration directory "`~/.testfarm/report`".

The Test Report generator converts these options into XSL stylesheet parameters, which are recognized by the default report stylesheet in order to generated the requested Test Report layout.

The table below lists the Report Options:

Option	***Default***	***Description***
Use standard layout	Yes	Indicates whether to use the standard TestFarm report stylesheet or not.
Stylesheet name	(none)	If the "Use standard layout" is disabled, this specifies which XSL stylesheet to use.
Ask for Operator Name and Report Suffix	No	If Yes, a Dialog window is raised when launching the Test Suite, asking for: - Operator Name, displayed in the report header; - Report Suffix, which is a string appended to the name of the generated HTML report file.
Show IN_TITLE messages	Yes	If disabled, do not show IN_TITLE print'ed lines in the report title.
Show IN_HEADER messages	Yes	If disabled, do not show IN_HEADER print'ed lines in the report header.
Show IN_VERDICT messages	Yes	If disabled, do not show IN_VERDICT print'ed lines in the verdict lists.
Show durations	Yes	If disabled, do not show the durations in report header, verdict lists and output dump sections.
Parenthesize names if not validated	Yes	If enabled, Scenario and Test Case names are parenthesized if the validation state is not "`VALIDATED`"
Generate HTML Test Log	No	If enabled, generate the HTML Test Log files and put the corresponding hyperlinks in the report.
Show verdicts by Scenario	Yes	If enabled, show the Scenario statistics in the verdict summary and the verdict list by Scenario.
Show verdicts by Test Case	Yes	If enabled, show the Test Case statistics in the verdict summary and the verdict list by Test Case.

© *TestFarm 2006-2012*

Option	***Default***	***Description***
Show criticity level	Yes	If enabled, show the criticity level in the verdict lists.
Show verdicts when non-significant	No	If disabled, do not show a Scenario/Test Case in the verdict lists if not significant (i.e. if it has no criticity)
Show verdicts when PASSED / FAILED / INCONCLUSIVE / SKIP	Yes	If the verdict value is enabled, show in the verdict lists the Scenarios/Test Cases that have such a verdict.
Enable output dump	Yes	If enabled, show the output dump section
Show validation state	Yes	If enabled, show the complete validation state for each Test Case.
Show output dump when PASSED / FAILED / INCONCLUSIVE / SKIP	Yes	If the verdict value is enabled, show in the output dump Test Cases that have such a verdict.

© TestFarm 2006-2012

10 The User Interfaces

10.1 The Test Suite User Interface

10.1.1 Features

The Test Suite User Interface is the front end of the TestFarm execution environment.

Its job is to load a Test Suite, to start the System Services and the Test Execution environment, in order to execute the Test Suite. The Test Suite User Interface also provides the ability to control the execution thanks to the following features:

- Test Case **Execution Control** (Start, Step, Abort, Reset, PERL debugging);
- Execution **Progress Visualisation** (Verdicts, Script Output, Test Log);
- Test **Report Generation**.

The TestFarm User's Manual [2] provides a detailed description of how to use the Test Suite User Interface.

10.1.2 System Definition Attributes

In the System Definition file (section Error: Reference source not found), it is possible to define Interfaces or Services that are activated only from the Test Suite User Interface, by setting the attribute "**mode**" of the SERVICE or INTERFACE elements to "**auto**". If this attribute is omitted or set to "ANY", the Interface or Service is activated from either the Manual User Interface or Test Suite User Interface.

10.2 The Manual User Interface

10.2.1 Purpose

Once the Test Features are defined in the System Definition file, they are ready to be used from the Test Scripts. As a complementary operating mode, it may also be useful to manually drive the Test Features, in order to perform some low-level setup or maintenance operations on the Test Interfaces. The TestFarm platform allows to directly invoke the Test Actions from a graphical user interface created by the System Integrator using the Glade interface builder.

10.2.2 Creating a Graphical Manual User Interface

Basically, each System Definition file should be accompanied with a Manual User Interface. Two files are to be considered when building a Manual User Interface:

- An **Interface Definition File** created using the Glade or Glade-2 Interface Building tool. The base name of this file is the same as for the System Definition file, and the suffix is "**.glade**".
- An optional **Interface Support Library**, which is a PERL module providing some action functions that are specific to the Manual Interface. It is allowed to call a Test Feature Library function from the Interface Support Library (the reverse is not true). The base

name of this file is the same as for the Interface Definition File, and the suffix is "**.support.pm**". Alternatively, it can be simply named "**support.pm**".

These files are placed beside the System Definition File: the prefered directory is "**/var/testfarm/lib**"

Example:

System Definition File	/var/testfarm/lib/MySystem.xml
Interface Definition File	/var/testfarm/lib/MySystem.glade
Interface Support Library	/var/testfarm/lib/MySystem.support.pm *or* /var/testfarm/lib/support.pm

10.2.3 Linking the Manual User Interface to the System

Widget Callbacks

The User Interface widgets are linked to the TestFarm Environment by invoking the Test Feature Library and/or the Interface Support Library functions as Widget Event Callbacks. A widget is a graphical object (e.g. a button) that may generate an event (e.g. "clicked"), which calls a function (a callback). In the context of the Manual User Interface, the callback functions must be present in the Interface Support Library or in the Test Feature Library.

Defining callbacks can be easily done using the Glade interface builder, by editing the "Signals" form of each Widget you want to hook a function call from.

System Definition Attributes

In the System Definition file (section Error: Reference source not found), it is possible to define Interfaces or Services that are activated only from the Manual User Interface, by setting the attribute "**mode**" of the SERVICE or INTERFACE elements to "**manual**". If this attribute is omitted or set to "ANY", the Interface or Service is activated from either the Manual User Interface or the Test Suite User Interface.

10.2.4 Using the Manual User Interface

The TestFarm Manual User Interface utility (command "**testfarm-manual**") is in charge of linking and launching the Test Feature library and the Manual Interface files presented above. It brings the Graphical Interface up, launches the System Services, and then starts the whole system as defined in the System Definition file. Once the Manual Interface is up and running, any action on the widgets it contains (buttons, etc.) will invoke the function it is hooked to.

Alternatively, it is possible to launch the Manual Interface from the Test Suite User Interface. This allows to perform manual actions while the Test Suite is in progress, which is helpful for debugging or troubleshooting.

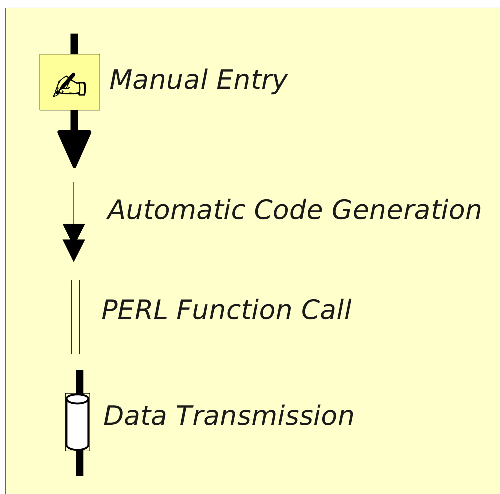
11 Glossary of Terms

Test Engine
Test Interface
Test Feature
Instrument
Command Interpreter
Interface Library
Test Suite
Test Tree
Test Case
Test Script
Test Sequence
Scenario
Tree Root File
Tree Branch File
Wizard Tools
SUT, System Under Test, Product Under Test
XML, eXtensible Markup Language
DTD, Document Type Definition
XSL, eXtensible Stylesheet Language
Glade

© *TestFarm 2006-2012*

12 Legend of Diagram Links

The diagrams illustrating the interconnection of different system components use a common rule for characterizing the links that assemble the blocks together:



13 REFERENCES

[1] IP070036-en : TestFarm Test Suite Reference Manual.

© TestFarm 2006-2012

List of Figures

FIGURE 4.1: ARCHITECTURE OVERVIEW....8
FIGURE 5.1 : CONNECTING A TEST INTERFACE TO THE TEST ENGINE....9
FIGURE 5.2 : CONNECTING A TESTFARM-COMPLIANT TEST INTERFACE....12
FIGURE 5.3 : DIRECT CONNECTION TO AN RS232 INTERFACE....13
FIGURE 5.4 : CONNECTING AN RS232 INTERFACE THROUGH A COMMAND INTERPRETER....13
FIGURE 5.5 : DIRECT TCP/IP CONNECTION....14
FIGURE 5.6 : TCP/IP CONNECTION THROUGH A COMMAND INTERPRETER....14
FIGURE 5.7 : TCP/IP CONNECTION TO A REMOTE CONTROL PROGRAM....14
FIGURE 5.8 : USING A GUI-BASED INSTRUMENT AS A TEST INTERFACE....16
FIGURE 5.9 : A SAMPLE INTERFACE COMMAND INTERPRETER....17
FIGURE 7.1: OVERVIEW OF THE TEST ENGINE....19
FIGURE 7.2 : A SAMPLE TEST SCRIPT SHOWING A SIMPLE SYNCHRONIZATION....21
FIGURE 7.3 : DATA FLOWS OF THE LOCAL AND GLOBAL TEST LOGS....23
FIGURE 8.1 : OVERVIEW OF THE TEST EXECUTION ENVIRONMENT....24
FIGURE 8.2: USING A MAGIC TRIGGER VARIABLE....32
FIGURE 8.3: THE TEST INTERFACE LIBRARY IN THE TEST EXECUTION ENVIRONMENT....34
FIGURE 8.4: THE BEGINNING OF A SAMPLE TEST INTERFACE MODULE....39
FIGURE 8.5: STRUCTURE OF A XML SYSTEM DEFINITION FILE....41
FIGURE 8.6: VIEWING THE SYSTEM DEFINITION FILE USING A WEB NAVIGATOR....42
FIGURE 8.7: A SAMPLE XML SYSTEM DEFINITION FILE (BEGINNING)....47
FIGURE 8.8: USING THE SYSTEM CONFIGURATION WIZARD....48
FIGURE 8.9: A SAMPLE INTERFACE ELEMENT IN A SYSTEM DEFINITION FILE....50

© TestFarm 2006-2012