User's Manual - Ordina Test Script Engine v1.00

12-10-2005 11:10

History

Date	Version	Issue
02-05-2005	0.90	Completely reworked the product. New version number system.
18-05-2005	0.99	Reviewed
05-10-2005	1.00	Lofar version created.

CONTENTS

TABLE OF TABLES

TABLE OF FIGURES

1 INTRODUCTION

1.1 About this Document

This document is the user's manual for Ordina Test Script Engine (OTSE).

This chapter explains the purpose and the benefits of using the Ordina Test Script Engine. The second chapter explains the installation procedure of the tool. The third chapter describes the function of the tool and also how to use it.

1.2 Purpose and Use

The main purpose for OTSE is to provide a development tool that makes it possible to verify protocols. OTSE can be a useful tool in the development phase as it can automatically perform multiple test cases by the use of scripts.

1.3 Delivery Objects

A delivery contains the following objects:

- User's Manual (this document)
- Release notes
- Ordina Test Script Engine software
- Example files

1.4 Abbreviations

Abbreviations	**Explanation**
BNF	Bachus Naur Form
EUT	Equipment Under Test
GUI	Graphical User Interface
OTSE	Ordina Test Script Engine
TSE	Test script Engine

Table 1-1. Abbreviations.

1.5 Glossary

1.6 References

2 INSTALLATION AND SETUP

2.1 Installation Procedure

OTSE is delivered with an installation program. This helps you to both install and uninstall. Start the installation by executing the file “rpm –i TSE-x.x-x.ix86.rpm”. The Redhat Package Manager (RPM) program will install TSE in the “/opt/lofar” directory.

2.2 Uninstallation

Execute “rpm –e TSE-x.x-x” on the command line to uninstall the Ordina Test Script Engine, where x.x-x indicates the version previously installed.

3 ORDINA TEST SCRIPT ENGINE IN A SYSTEM

Ordina Test Script Engine is a software tool that runs on Linux. OTSE will be one central component when developing and testing protocol implementations.

In the next subchapters the usages of OTSE in the different architectures are presented.

3.1 Ordina Test Script Engine components

Ordina Test Script Engine consists of the following parts:

- PROT-file (Protocol Specification)
- Scripts
- Batch files
- The test script engine

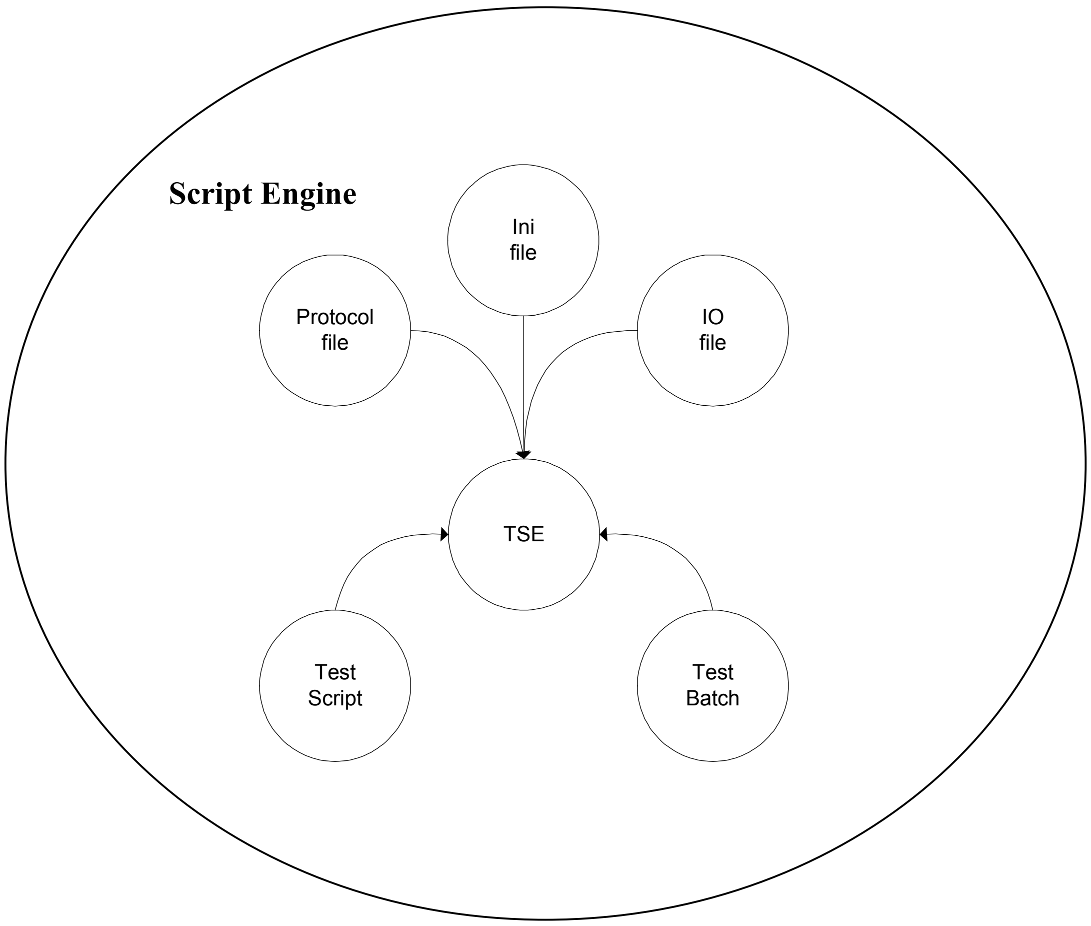


Figure 3-1. Script engine components

Protocol Specification, ini, io, scripts and the batch file are stored in separate external text files, and can be modified by the user.

3.1.1 Protocol Specification

The Protocol Specification[1] consist of three sections:

- The **type** definition section where the data types are defined to have a specified size and are the building block for commands and events.
- The **function** definition section where the commands are defined with their parameters. In this section all possible output to the I/O-ports are given.
- The **event** definition section where the events are defined with their parameters and identified in the test script by their constant values. Here all valid inputs from I/O-ports are presented.

The scripting and the layout of the configuration layout will be further described in chapter 4.

```
[type]
t_Command             = { 2, 0x0000, 0xFFFF, COMMAND }
t_Inquiry_Length      = { 1, 0x01  , 0x30  , TIME, 1.28 }
t_Period_Length       = { 2, 0x0000, 0xFFFF, TIME, 1.28 }
t_Max_Period_Length   = { 2, 0x0003, 0xFFFF, TIME, 1.28 }

[functions]
NOP = { 0x00, 0x0000 }

Inquiry = { 0x01, 0x0001,
  LAP                             : t_LAP,
  Inquiry_Length                  : t_Inquiry_Length,
  Num_Responses                   : t_Num_Responses
-
}

[events]
Inquiry_Complete_Event = { 0x01,
  Status                          : t_Status,
  Num_Responses                   : t_B1
}
```

Figure 3-2. Example of a PROT file

[1] In some documents called “configuration file” or “PROT file”

3.1.2 IO Files

In the IO files the communication channels are defined that are used by the state machines defined in the script files.

```
[io]
socket1={SOCKET,LOFAR,16834,”server 127.0.0.1 7775”}
socket1={SOCKET,LOFAR,16834,”client 127.0.0.1 7775”}
```

Figure 3-3. Example of an IO file

Each entry has the following syntax:

<name> = {<kind>,<protocol>,<buffer size>, <options>}

where:

- name : the name of the channel, which is used by the test script.
- kind : the kind of connection can only be socket for now.
- Protocol : the protocol used during the connection.
- Buffer size : the size of the buffer where the received messages are stored.
- Options <optional> : the options for the channel in case of a socket connection. The type of the socket client/server. The remote or local address where to connect or to listen on. The last is the port on which is communicated.

3.1.3 Ini File

The ini file contains the user settings for the TSE. The user can set the following items:

```
Loop                    = 0 // Do not loop
StopOnError             = 1 // Stop when an error occured
logview                 = 0
logline                 = 0
logtofile               = 0
Replay                  = 0
```

Figure 3-4. Example of the ini file

- LogToFile : indicates if extra logging is required.
- LogView : created a file in where all the state transistions are logged. .
- LogLine : Logs all data that is received or sent. .
- Replay : replay the last log file of a test script or test batch. .
- Loop : loop the script or batch file.
- StopOnError : stop on error when looping a test scripts or batch file.

3.1.4 Script files

The script file defines the sequence of the test, which commands to send and which events to expect. In the script file one or several state machines are defined.

```
Inquiry_State_Machine(L,N) = {
VAR Max,Min.
s0 : Inquiry(L, 0x10, N) ; s1.

s1 : Command_Status_Event(0x00,,Inquiry) ; s2.
s1 : Command_Status_Event(,,Inquiry) ; s0.

s2 : Inquiry_Result_Event(,,,,,,) ; s2.
s2 : Inquiry_Complete_Event(0x00,) ; s0.
s2 : Inquiry_Complete_Event(,) ; error.
s2 : TIMER(0x07) ; ready.

ready : Inquiry_Result_Event(,,,,,,) ; ready.
ready : Inquiry_Complete_Event(,) ; s0.
ready : TIMER(0x05) ; timeout.

timeout : Inquiry_Result_Event(,,,,,,) ; timeout.
timeout : Inquiry_Complete_Event(,) ; s0.
timeout : TIMER(0x05) ; s0.

error : TERMINATE; error.
}
```

Figure 3-5. Example on a state machine

Each state has a list of allowed commands or events. If a command or event occurs, the state machine will perform a state transition. In the new state another set of commands and events are allowed.

In the above example **s0** is a state and in this state, only the command Inquiry is possible. This command with its parameters will be sent; a transition to **s1** is then performed. In the new state an event **Command_Status_Event** is expected.

At the end of each file a Testscript section connects state machines to an io-channel, for example: **COM1: Inquiry_State_Machine(…)**

The script language is further discussed in chapter 4 and 5.

3.1.5 Batch files

A complete test normally consists of a collection of smaller test sequences. Multiple script files can be arranged in batches. A test batch file is an ordinary text file with a simple format. When a batch file is started, all defined scripts will process in sequential order. One general log file plus a log file per script is generated. The general log file contains a one-line result per script. The name of this log file is the same as the name of the batch file with the extension replaced by ".log". Script logs are named in the same way, script file name with the extension ".log". More on batch files in chapter 5.5.

```
prot_1_0_l2cap.prot
timer_null2.tse
timer_null.tse
end.
```

Figure 3-6. Example of a Batch File

In the example above is **prot_1_0_l2cap.prot** the configuration file, **timer_null2.tse** and **timer_null.tse** are two scripts, which will be executed, in sequential order.

3.1.6 Script Interpreter

The script interpreter "Ordina Test Script Engine" is an executable file. It can be started from the command line.

3.2 Output from OTSE

The output from Ordina Test Script Engine is logging information from state transitions, commands and received events. The progress can be viewed by examining the log-file. In the script overview logging is it possible to see the current state of the state machines. An error is generated when a script did not end in the ok (or OK) state.

3.3 Quick start

This chapter contains basic information of how to get you quickly started with OTSE.

3.3.1 Preparation

Before starting OTSE, make sure you have configured the IO file by assigning EUTs to channels.

3.3.2 Test

Start with either open batch/script file(s) manually by entering: *tse <protocol file> <io file> <test script> <test script log file>* and press enter. The prompt will return when OTSE has finished

4 SCRIPT LANGUAGE INTRODUCTION

The following chapters will describe the script language and protocol specification of OTSE. This will be of interest for an OTSE user who either wants to write new test scripts or wants to get a deeper understanding of the program.

OTSE has a script language specially designed for test purposes. Initially it was developed to automate the tests made by Ordina on the Bluetooth HCI-modules. The script language is general enough to be useful for other test purposes. A few naming conventions remain that reflects the main use in the Bluetooth domain.

Since OTSE is targeted at automatic tests, two important areas had to be addressed when choosing the script language:

Speed/Ease of use: Make the process of writing test scripts as fast as possible.

Input to and output from OTSE: The communication between the equipment under test (EUT) and OTSE had to be adapted to test conditions — as little runtime input from the test team, and good communication logs should be produced.

4.1 State machines

State machines are used to describe how a machine (read computer) reacts to external events. Events make the machine, in a specified state, jump to another state. This is called a state transition.

There is no memory or registers to hold information for later use. The only information that changes dynamically is the *current state* of the state machine. Below is an example of how a state machine could be described in a graph.

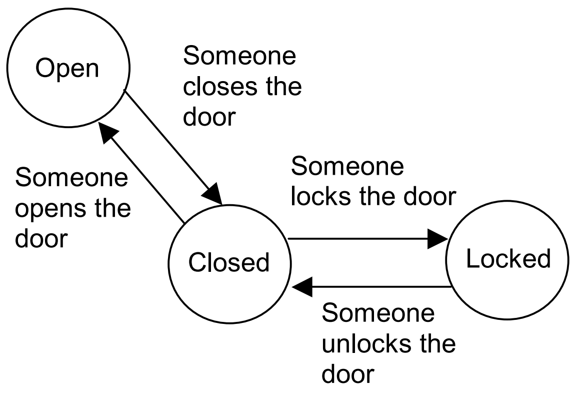


Figure 4-1. State chart representing a door. The current state of the door changes depending on external stimuli: every arrow represents a valid state-transition and has the reason for the transition marked beside.

A state transition is described by three parameters: Origin state, destination state and the event that triggers the transition.

The script language in OTSE is based on state machines. State transitions occur when events occur or when commands are performed. The latter means that a transition between two states can cause actions in the outside world. In the test environment this means that two-directional communication with the EUT is possible.

4.2 State machine example

An example from the Bluetooth world could be illuminating. We will neglect the details of the communication protocol here.

The EUT is in this case Bluetooth HCI–module. The test case described here is a very simple check of the basic function of the module.

A good HCI-module would respond to a `Reset`-command with a `Command Complete` event. The response should be returned before a certain timeout. Figure 4-2 depicts a state machine that tests the HCI-module.

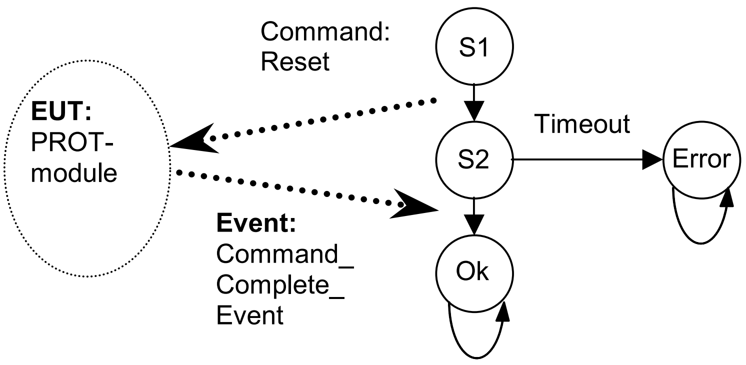


Figure 4-2. During the state machine a command is sent it then waits for an event from the HCI-module.

This state machine could be written in a test script as follows:

```
[statemachines]
Tx=
{
S1      : Reset ; S2.
S2      : Command_Complete_Event(,Reset,0x00) ; Ok.
S2      : TIMER(0x01) ; Error.

Ok      : TERMINATE.
Error   : TERMINATE.
}
```

Each of the state transitions (the arrows in Figure 4-2) is represented as one line in the script. For example, the first line describes the transition from state `S1` to state `S2`. During this transition a reset command is sent to the EUT.

When `S2` is the current state, there are two possible transitions: to state `Ok` or to state `Error`. If a command complete event is received that matches the requirements on line two a transition to `Ok` is made. If no matching event has arrived, a timeout transition is made after 1.28 seconds[2].

The action TERMINATE tells OTSE to stop the execution of the script, so the two last lines define the two terminal states Ok and Error. *The test result is the terminal state of the script*. Ok means that the EUT passed the test and Error means failure. A test script might contain a chain of different tests that forms a more complex test case.

[2] OCSE uses by default a smallest unit of time of 1.0 seconds.

4.3 Protocol specification

A command or an event triggers every state transition. There are a few internal commands and events, but a main feature of the script language is that it is possible to define new commands/events adapted to the protocol used in communication with the EUT. Those are all defined in a protocol specification file (*.prot). OTSE prompts the user when an error, e.g. inconsistency and/or duplicate data, in the protocol is found.

The incoming stream of bytes on the communications port will be examined according to the Protocol Specification. Whenever a received string of bytes matches the definitions in the protocol specification file it is recognized as an event.

The same applies when a state machine sends out a command it is converted by the communication interface.

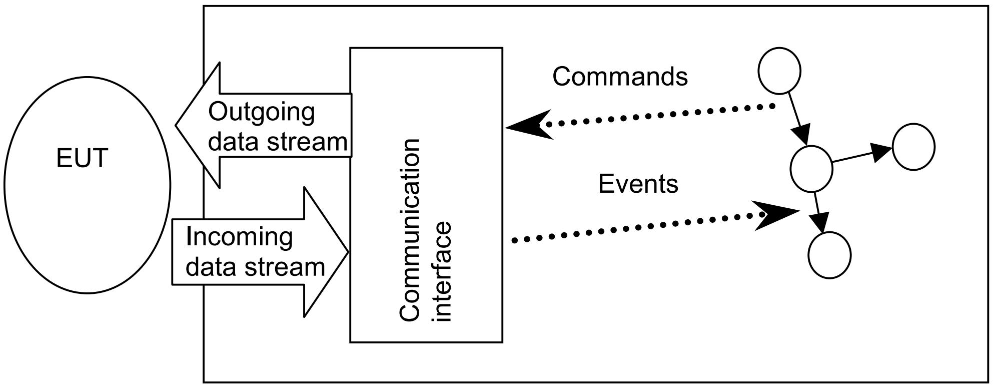


Figure 4-3. The Protocol specification file controls the relation between abstract commands/events and I/O.

There are many advantages of using abstract commands and events in the state machine scripts. Among others:

- More readable scripts.
- Small changes in communication protocol will only influence the protocol specification file and not all script files.

4.4 Storage and Synchronization

It is possible to activate several state machines in a single script file. They can be, for instance, be connected to two HCI-modules and execute complex communication procedures over the air. Such state machines can share data storage and also use the simple synchronization facilities. Signals can be sent between different state machines using the same argument passing technique[3] that is used for sharing data.

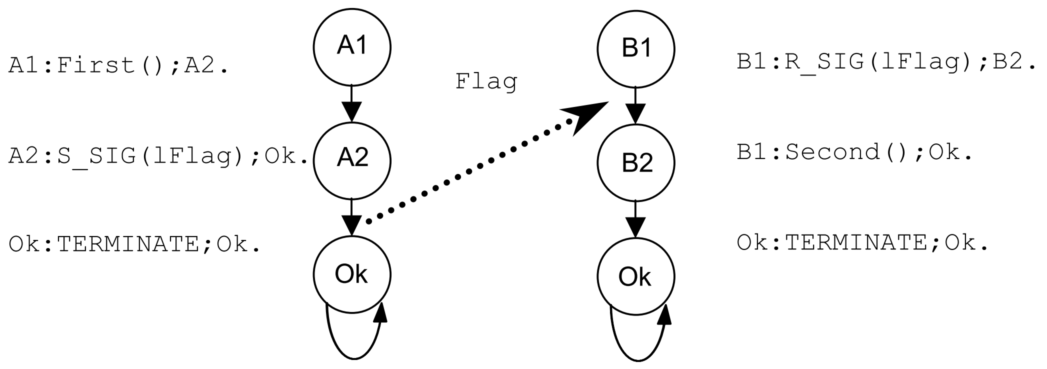


Figure 4-4. Two state machines that are synchronized via the variable flag. The transition B1→B2 is blocked until any other state machine executes an S_SIG-action. `lFlag` *is actually two different local variables but they are assigned, upon invocation of the two state machines, to the same* `Flag` *variable in the [testscript]-section.*

In OTSE script language there are two ways of storing data, using variables and data buffers. Together with the possibility of random behavior it allows you to create short scripts that explores a large number of different test cases.

The current value of variables and signals is presented in hexadecimal form in the overview window. The content of data buffers is displayed as strings in Show Headers mode.

4.5 Use of files

In order to run OTSE a few files have to be present:

- The protocol specification (file name extension ".prot").
- The ini file (file name tse.ini).
- The IO file (extension *.io).
- The scripts (extension ".btsw"). Here variables and state machines are defined.
- A batch file (extension ".testbatch"). This is an optional file used when several scripts are supposed to be executed in serial manner.

During execution of script or batch files OTSE produces to different types of log files (extension ".log"). They reflect the log information that also is shown on screen.

[3] See chapter 5.3.2.

5 SCRIPT LANGUAGE REFERENCE

5.1 Introduction

This manual describes the OTSE script language. There are three types of input files: .btsw, .prot and .testbatch that has different syntax. The first two types are divided in sections, see Table 5-1 below, and will be described in depth in this chapter. The syntax for batch processing is described in chapter 5.5.

Script file .tse	Protocol Specification file .prot
[statemachines] [testscript]	[type] [functions] [events]

Table 5-1. The mandatory sections for the Script and Protocol Specification files.

Common for all sections are the 5 classes of tokens: identifiers, keywords, constants, operators, and white spaces. They will be described in the following subchapters.

5.1.1 Conventions used in this chapter

All code examples are typeset in `Courier` in this chapter.

The grammars for the different sections are presented in BNF form. The symbols are typeset in *`italic type Courier`*. **`BOLD`** style words and characters are terminals. There are also a few terminal symbols: *`identifier`*, *`qstring`*, *`number`*, *`realnumber`* and *`action`* that are defined in the following subchapters.

5.1.2 White space

White space separates other tokens from each other. Valid characters are Space, Tab, Carriage Return and Line Feed. Comments are also treated as white space and is marked by the //-construct (as in C++).

5.1.3 Keywords

In Table 5-2 the keywords are presented mind that OTSE is case sensitive.

file	Section	Keywords defined in the section

protocol specification	**[type]**	`BITFIELD`
		`COMMAND`
		`ENUM`
		`TIME`

	[functions]	`N/A`
	[events]	`N/A`

Test script	**[statemachines]**	`COMPARE`
		`CLEAR`
		`IF`
		`MTIMER`
		`RESCUE`
		`R_SIG`
		`RTIMER`
		`RMTIMER`
		`S_SIG`
		`TERMINATE`
		`TIMER`
		`WAIT`
		`VAR`

	[testscript]	`VAR`

Table 5-2. The sections and their keywords.

5.1.4 Identifiers

An identifier is a sequence of letters and digits. The first character has to be a letter. The underscore _ counts as a letter.

```
    identifier:
         (A letter followed by any sequence of alphanumeric characters)
```

Identifiers have different scope depending on the kind and where it is defined. The most important rule is, that global variables have to be passed as an argument to a state machine in order to be available inside. Transfer of information between state machines is achieved by passing a global variable as argument to both state machines. Since argument passing is *by reference* this allows for information flow between state machines. This is the method used for inter-state machine synchronization.

To avoid name conflicts, it is recommended to use lower case or mixed case when referring to user–defined identifiers (`ThisIsTheFirstState`, `databuffer1`, `flag` etc.). A user–defined identifier cannot have the same name as any of the keywords.

5.1.5 Constants

There are three kinds of constants:

- Integer constants can be expressed in decimal or hexadecimal form –as in `103` and `0x3B`. The terminal symbol is *`number`*.

- Floating constants, real numbers expressed with a decimal point. The terminal symbol is *realnumber*. Only the format 0.23 is allowed, it is not possible to express an exponent, like in 2.3E-1.

  - Strings constants, is surrounded by double quotes as in "This is a string". The terminal symbol is *qstring*. A few of the standard C escape sequences are supported: "\n \r \t \\" namely Carriage Return, Line Feed, Tab and Backslash.

5.2 Protocol Specification

Before the protocol specification file is read, the OTSE does not know what commands and events are available. Neither does OTSE know about the ranges of specific parameters, nor the Endianess of the parameters. All this information is stored in this separate file. The advantage of this solution is that several versions of a protocol, or completely different protocols, can be tested with the same OTSE version: There is no need to recompile the program for different protocol versions.

As mentioned before, the protocol specification file is used to define the protocol to be used in the communication with the EUT. One special aspect of the protocol is addressed:

- Command and event packages are constructed with user–defined data types. This is done in the three remaining sections: [type], [functions] and [events].

The order of sections is fixed. First come [type], then [functions] and finally [events].

There is a simple preparser that can be used to include other files into the protocol specification. Make sure to first include the types that are used in the functions or events.

The use is very simple; below is an example of how to include type.prot file into this protocol specification.

```
    #include "type.prot"
```

An error is generated when a duplicate name is found.

5.2.1 Type section

The OTSE script language has a complex type definition section. The types defined here are the building blocks for the different command and event packages.

All types can be of any integer size of bytes and it is also possible to group bits together using BITFIELD. Allowed type definitions:

```
    typedef:
         typename = [ - ] { size }
         typename = [ - ] { size , min , max }
         typename = [ - ] { size , min , max , timedef }
         typename = [ - ] { size , min , max , enumdef }
```

The first minus sign '-' is optional, by default all types are little Endian. Adding a minus sign in the type definition makes the type to be big Endian. The second minus sign allows that less than the specified size is allowed. This option is used with strings and arrays, so that a string with size of 5 characters is also valid when it contains the characters “tse”. Strings and arrays are explained later on in this chapter.

The typename and size follows the definition:

```
typename:
     identifier

size:
     number
```

The `typename` will be used in other parts of the protocol specification: the command and event definitions. The `size` determines the number of bytes needed to express a parameter of this type.

```
min:
     number

max:
     number
```

The numbers `min` and `max` defines the minimum and maximum value of parameters of this type. The size in bytes of these values must match the `size`. OTSE makes use of the `min` and `max` value to check received parameters and uses these limits when it has to generate a random value.

5.2.1.1 Time definition

```
timedef:
     TIME , timescale

timescale:
     realnumber
```

Parameters of this type are used to indicate a certain time span. A type of the `TIME` kind will help the built–in action `TIMER` to select the correct time base. Using a variable —defined with such a *`timescale`*— as argument to the timer will make OTSE set up a timer of length “value of variable” multiplied with the time scale. Below is an example for HCI-modules:

```
extract from [type] section
t_Interval_with_0 = { 2,0x0000,0xFFFF,TIME,0.000625 }

extract from [statemachines] section
s11: Mode_Change_Event( , , 0x03 , timeinterval); s12.
s12: TIMER(timeinterval) ;s13.
s13: Mode_Change_Event( , , 0x00 , ); s14.
```

Example 5-1. This is an example of the use of `TIMER` *together with a variable. Here the time scale for the timer is set in the type definition for the fourth argument of the HCI_MODE_CHANGE_EVT. The time scale is set to 625µs. If the variable* `timeinterval` *gets the value 200 then the state machine will wait 200x625µs = 125ms in the* `s12` *state.*

5.2.1.2 Enum definition

```
enumdef:
     ENUM , enumlist

enumlist:
     enumeration
     enumeration , enumlist

enumeration:
     number : enumdescription

enumdescription:
     qstring
```

This option is used to assign human readable descriptions to the values of this type. The size in bytes of these enumeration numbers must match the `size` mentioned before. The numbers in an ENUM type must be non-equal, but OTSE does not check this while reading the specification. It is not needed to enter the enumerations in a particular order.

5.2.1.3 String definition

Strings can be defined as:

```
typedef:
     typename = [ - ] { [ - ] size , ASCII }
     typename = [ - ] { [ - ] size , ASCII0 }
     typename = [ - ] { [ - ] size , ASCIIn }
     typename = [ - ] { [ - ] size , UNICODE }
     typename = [ - ] { [ - ] size , UNICODE0 }
     typename = [ - ] { [ - ] size , UNICODEn }
```

Where the ASCII variants indicate 7-bits ASCII encoded in 8-bits units; and the UNICODE variants indicate 7-bits ASCII encoded in 16-bits units. The latter is compatible with the Unicode format.

The variants ASCII0 and UNICODE0 are 0-terminated variants. The last element is an 8 bits or 16 bit '\0' character. The variants ASCIIn and UNICODEn are line-feed terminated variants. They are usually useful in ASCII-based protocols as used on e.g. VT100 terminals.

5.2.1.4 Array definition

Arrays can be defined as:

```
typedef:
   typename = [ - ]{ no elements[ - ], ARRAY, size of element }
```

Where the size of element is the number of bytes of an element and no elements to maximal number of elements in the array.

5.2.2 Functions section

Command packages to the EUT are defined in this section. By specifying the sequence of variables (and their data type – using the `typename` as identifier) the package structure is defined.

```
functiondef:
     functionname = { input }

input:
     (empty)
     parameterdeflist


parameterdeflist:
     parameterdef
     parameterdef , parameterdeflist
     parameterdef UNION { unionlist } , parameterdeflist
     parameterdef UNION { unionlist uniondefault } , parameterdeflist
     parameterdef { '{' parameterdeflist '}' '[' parametername ']' }

unionlist:
     union
     union , unionlist

union:
     ( identifier = number ) parameterdeflist

uniondefault:
     ELSE parameterdeflist

parameterdef:
     parametername    ':' typename
     parametername    ':' typename lenghtindicator
     const

lengthindicator:
     '(' parametername '..' paramtername ')'
     '('               '..' paramtername ')'
     '(' parametername '..'              ')'
```

5.2.2.1 Parameterdef

The definition of 'parameterdef' has three possible expansions. The 1st one, (parameter name: type name) is a parameter that needs to be filled in by test scripts. The 2nd and 3rd (with length indicator, or the const) do not need to be filled in by test scripts. The script engine itself fills in these fields. These parameters are called 'Internally determinable'.

5.2.2.2 Union definition

A union starts with a condition `identifier = number`. The `identifier` must be the parameter directly preceding the union: which element of a union is chosen must depend of that very parameter.

The `number` must be in the range 0 to 10. An `ELSE` can be added to describe default behavior.

5.2.2.3 Repetitive Structures

Some protocols have a list of parameters that is repetitive. OTSE can deal with this under the precondition hat the number of repetitions is directly defined by a field preceding the repetitive part. A good example can be:

```
...
NrOfRecipes    : t_B1,
{
    RecipeNr   : t_B2,
    RecipeName : t_RecipeName
} [ NrOfRecipes],
NextParamter : t_B2,
...
```

In this example, the two parameters RecipeNr and RecipeName can occur multiple times; indicated by the parameter NrOfRecipes. The parameter indicating the number of repetitions **must** be a constant in the corresponding test scripts, because during parsing of a script OTSE must know how many parameters follow for the repetitive structure.

Repetitive structures can occur recursive at any desired level of depth.

5.2.3 Events section

Events are defined similarly to commands.

```
     eventdef:
          eventname = { input }
```

5.3 Script language

A script consists of 3 sections stored in a file; see Table 5-1. They are presented below in the order they can appear in a script file[4]. The order is important because the script is translated in one pass and every identifier must be defined before it is used.

The first section defines data buffers. The second defines state machines and the last “links” state machines to the communication channels.

5.3.1 Statemachines section

In this section the state machines are defined. Every state machine will be linked to a communication channel later on in the [testscript] section. Commands will be sent and events will be received through this single[5] channel.

State machine names must be unique. The same applies for the parameters of the state machine, and all local variables. All these identifiers must be unique, but this is not checked by OTSE. The behavior of the state machine is undefined if there are double defined variables.

Parameters to state machines are passed by reference; this is further discussed in chapter 5.3.2.

[4] Or *files* since the preparser can include other files into one script. More information in chapter 5.3.3

[5] This means that one state machine can not communicate with two HCI-modules, for instance. In this case two state machines have to be defined, probably taking advantage of the synchronisation features of the script language.

The body of a state machine consists of a list of state transitions. Every transition is described by three parameters: origin and destination states and the trigger action. The latter is an event, a command or an internal action.

Except for the first transition, that defines the initial state, there is no need to group the transitions in a special order. However, when several transitions have the same origin it is important to write the transitions in the correct order. There are rules that govern what transition should be performed at runtime, see chapter 5.4.1.

5.3.1.1 Grammar

```
statemachine:
     statemachinename = { localvars rescue statelist }
     statemachinename ( arglist ) = { localvars rescue statelist }

statemachinename:
     identifier

localvars:
     VAR varlist .
     (empty)

varlist:
     vardec
     vardec , varlist

vardec:
     varname
     varname = number

rescue:
     RESCUE statename .

arglist:
     varname
     varname , arglist

varname:
     identifier

statelist:
     statetrans
     statetrans statelist

statetrans:
     statename : action ; statename .
     statename : action -> statename .

statename:
     identifier

action:
     eventname
     eventname ( paramlist )
     functionname
     functionname ( paramlist )
     internalaction
     internalaction ( paramlist )
```

```
paramlist:
     parameter
     parameter , paramlist

parameter:
     varname
     functionname
     number
     (empty)
```

An `action` is thus an event identifier, a command identifier[6] or an internal action. The latter are internal commands or events. If the action is defined to have a parameter list (`paramlist`) it must of course match the parameter definition list as defined in the protocol definition: For events, the number of parameters must equal the number of return parameters as defined. For commands, the number of parameters must equal the number of command parameters.

OTSE inserts a random value when a command parameter is left `empty` to match the type definition in the [data] section. If an event parameter is empty this means that that the actual value of the parameter does not matter for the test – any valid value will be allowed.

All the built-in `internalaction`'s are described in chapter 5.3.1.4.

5.3.1.2 Parameter list of events and functions

Events and functions as defined in the protocol file can have a number of parameters. In the protocol file, 'Internally determinable' parameters can be defined; these parameters are not mentioned in test scripts. These parameters identify the event when it is received; this means that events must consist out of a unique combination of constants parameters. Imagine an event that is defined in the protocol file as:

```
Event = {
  0x00,
  0x01,
  lenght       : t_B1 ( recipename .. ),
  NrOfRecipes : t_B1,
  {
     RecipeNr : t_B2,
     RecipeName : t_RecipeName
  } [ NrOfRecipes ]
}
```

In a test script, this event can be used in a transition. The parameters 0x00, 0x01 and length will not be mentioned in the test script, they can be determined internally. The parameter NrOfRecipes is the first parameter to be mentioned in the script. Possible occurrences are:

```
s0 : Event (0x00 ) ; s1.
s0 : Event (0x01, 0x1234, "Name1" ) ; s1.
s0 : Event (0x02, 0x1234, "Name1" , 0x2345, "Name2" ) ; s1.
```

[6] `functionname` refers to a command definition. In this manual the name "command" is used instead of "function". Note that there is a [functions] section in the Protocol Specification file where the commands are defined.

To improve readability, long parameterlists can be divided over multiple lines:

```
s0 : Event (0x06,
                0x1234, "Name1" ,
                0x2345, "Name2" ,
                0x3456, "Name3" ,
                0x4567, "Name4" ,
                0x5678, "Name5" ,
                0x6789, "Name6"
            )                                ; s1.
```

5.3.1.3 Variable declaration

As indicated in the grammar section it is possible to define local variables. Their scope is in one instance of the state machine – as for local variables in a C function. A big difference is that they do not have to be typed. This is done at runtime.

In the [testscript] section a similar construct exists for variables. The scope for such variables is limited to the same section.

5.3.1.3.1 Constant variables

In the variable declaration it is possible to assign a value to the variable. This variable gets a fixed value and is behaving like a constant. A `CLEAR( )` will not clear this constant variable.

5.3.1.3.2 Dynamic typing

If a `varname` is used as a parameter, that variable gets the type of that parameter as defined in the protocol specification. If the `varname` is used on several places in the same state machine, the needed types must match. Below is an erroneous example code, cut out from a state machine definition that tests HCI-modules:

```
s0 : Command_Complete_Event( ,
        Read_Connection_Accept_Timeout, 0x00, my_var) ; s2.
s0 : Command_Complete_Event( , Read_Page_Timeout, 0x00,
        my_var) ; s1.
s1 : Write_Scan_Enable(my_var) ; s3.
```

The `Command_Complete_Event` has two fixed parameters: the `Num_Hci_Command_Packets`, and the `Command_Opcode`. The variable parts are the return parameters of the `Read_Connection_Accept_Timeout` command and the `Read_Page_Timeout` command. When the first line is parsed, the variable `my_var` gets type `t_Interval`, the type of the parameter as defined in the protocol specification. When the second line is parsed, OTSE tries to assign the type `t_Interval` to the same variable. Because the types match, the second line is accepted. In the third line, the variable `my_var` is used as the parameter of `Write_Scan_Enable`. The type of this parameter is `t_Scan_Enable`, and OTSE tries to assign this type to the variable. Because the variable already has another type, an error message is generated.

All variables used in state transitions must be defined in either the local variable list, or in the parameter variable list of the state machine. Variables that are not used generate a warning.

5.3.1.3.3 Multiple event transitions

The multiple event transition is extremely useful in the following situation: Imagine a test environment with three discoverable devices in the surroundings with known addresses. The test environment is paging them, but cannot predict the order in which they respond. However, we want to proceed only when the response of all of them is received. This can be done with a script like:

```
s00 : Send_Page_Request                  ; s01.

s01 : Page_Response(1, adr1, , , , , )
      Page_Response(1, adr2, , , , , )
      Page_Response(1, adr3, , , , , )   ; s02.
s01 : Page_Response(1,     , , , , , )   ; s01. // from room above us.


s02 : ...
```

In state s01 we wait for three Page_Response messages (with indicated addresses). The transition to s02 is made only then when all Page_Response messages have been arrived at least once. The order in which they arrive has no influence. An alien Page_Response with unknown address is captured by the 2nd transition in state s01. Such an intervening message has no influence on the 1st transition, since state s01 is not left.

However.

If state s01 is left and re-entered later, old events are “forgotten”. Imagine the following test script:

```
s00 : Send_Page_Request                  ; s01.

s01 : Page_Response(1, adr1, , , , , )
      Page_Response(1, adr2, , , , , )   ; s02.
s01 : Page_Response(1, adr3, , , , , )
      Page_Response(1, adr4, , , , , )   ; s03.

s02 : (do something with adr1 and adr2) ; s00.

s03 : (do something with adr3 and adr4) ; s00.
```

Suppose that page responses arrive in this sequence: First from adr1, next from adr3, next from adr4, and finally from adr2. The 2nd transition in state s01 is completed before the response from adr2 has been arrived, so the transition to state s03 will be made. The response from adr2 arrives when the script engine is in state s03. When the script engine enters state s01 again, previous arrivals of Page_Response messages are forgotten, including the arrivals of events that did not lead to a transition.

If a response arrives which is not anticipated then a warning is logged in the logfile. When the user wants to take action when such unanticipated event arrives, the user must use the UnhandledEvent state, for example:

```
s00 : Send_Page_Request                  ; s01.

s01 : Page_Response(1, adr1, , , , , )
      Page_Response(1, adr2, , , , , )   ; s02.
s01 : Page_Response(1, adr3, , , , , )
      Page_Response(1, adr4, , , , , )   ; s03.

s02 : (do something with adr1 and adr2) ; s00.
s03 : (do something with adr3 and adr4) ; s00.

UnhandledEvent: Terminate.
```

5.3.1.4 Internal actions / Keywords

One group of internal actions is CLEAR, R_SIG, S_SIG, TIMER, MTIMER, RTIMER, RMTIMER, TERMINATE and WAIT. These actions have in common that they do not interact with the device under test; they are only used to influence the behavior of the state machines by internal means. Available internal actions are described below.

5.3.1.4.1 CLEAR

One parameter is allowed and it must be a variable.

The value of the variable is cleared – the variable becomes uninitialized. This is useful when the variable is used in an infinite loop to store received values temporarily.

Example: a state machine is used to disconnect all incoming connections:

```
init: CLEAR(handle); s00.
s00 : Connection_Complete_Event(0x00, handle, , ,) ; s01.
s01 : Disconnect(handle,)                          ; s02.
s02 : Command_Status_Event(,,Disconnect)           ; init.
```

Example 5-2. When `handle` *is not initialized, all handles are accepted in state* `s00`*, and the received handle is assigned to the variable.*

During the transition `s00` to `s01`, `handle` obtains the value from the event package. This value is later used in the `Disconnect` command. If `CLEAR` would not have been included in the loop the same value would also be expected in the next round.

5.3.1.4.2 RESCUE

One parameter is allowed to set the rescue state and it must be a state.

At the start of the statemachine a rescue state can be defined. When a RESCUE state is entered all of the statemachines who have a rescue state defined go to that perticulary state.

Example: two statemachines the first is receiving a disconnect event instead of a reply then the RESCUE is used to let the other statemachine to terminate gracefully:

```
S1(bReady) =
{
 RESCUE s100. // Not executed

s00 : Connection_Complete_Event(0x00, handle, , ,) ; s01.
s01 : Send_Command( , ,)                          ; s02.
s02 : Command_Complete_Event(,,,)                 ; s03.
s02 : Disconnect(handle,)                         ; sError.
s03 : Command_Status_Event(,,Disconnect)          ; s110.
SError: RESCUE                                     ; sHalt.

sHalt: S_SIG(bReady)                               ; sWait.
sWait: TIMER(0x100)                                ; sEnd.
sEnd : TERMINATE.
}

S2(bReady) =
{
 RESCUE s100.

…
s100: Shutdown_Link()                              ; sHalt.

sHalt: S_SIG(bReady)                               ; sWait.
sWait: TIMER(0x100)                                ; sEnd.
sEnd : TERMINATE.
}
```

Example 5-3. A disconnect event is received when expecting a reply.

The rescue state can be used to do a gracefull shutdown, after an error occurred in another statemachine.

5.3.1.4.3 IF

Two parameters of the same type are allowed and one or both must be a variable.

The IF statement is used to compare two variables or to compare a variable and a constant. Table 5-3 lists all of the operators that are allowed to compare:

OPERATOR	REMARKS
==	Equal, test if both parameters are equal.
!=	Unequal, test if both parameters are not equal.
<>	Unequal, test if both parameters are not equal.
<	Less than, test if the first parameter is less than the second parameter.
>	Greater than, test if the first parameters is greater than the second parameter.
<=	Less than or equal, test if the first parameters is less than or equals the second parameter.
>=	Greater than, test if the first parameters is greater than or equals the second parameter.
&	Boolean AND, allows two comparisons in a single IF statement, is TRUE

	when both comparisons are TRUE.
|	Boolean |, allows two comparisons in a single IF statement, is TRUE when one of the comparisons is TRUE.
~	Boolean NOT, inverse a comparison when the comparisons was TRUE it becomes FALSE and vice versa.

Table 5-3. The operators of the IF statement

Example: two if statements, one comparing two variables if one is less than the other, the second compares if a variable equal a constant.

```
s112 : IF ( uiCounter < uiEndCondition ) ; s100.
s112 : IF ( uiCounter == 0x0000000A )    ; s120.
```

The IF statement can be used to create loops in a statemachine, in a loop usually variables are incremented or decreased to a certain value. The allowed calculation operators to do so are described in Table 5-4.

OPERATOR	REMARKS
+	Add
-	Subtract
*	Multiply
/	Division

Table 5-4. The calculation operators that can be used in a statemachine.

Example: Calculation with a variable:

```
s110 : uiCounter = uiCounter + 0x01     ; s111.
s111 : uiCounter = uiCounter * 0xA0     ; s112.
s112 : uiCounter = uiCounter / 0xA0     ; s113.
s113 : uiCounter = uiCounter - 0x01     ; DONE.
```

5.3.1.4.4 R_SIG

One parameter is allowed and it must be a variable.

The variable type is set to an internal data type, which cannot be used by other events or commands but S_SIG. If the variable is initialized, R_SIG will de-initialize it, and perform the state transition. If the variable is not initialized the state transition is blocked.

In practice R_SIG is the only action in a state and the state machine will wait in this state until the variable is initialized, de-initialize it and continue.

5.3.1.4.5 S_SIG

One parameter is allowed and it must be a variable.

The variable type is set to an internal data type, which cannot be used by other commands or events but R_SIG. The variable will become initialized, and the state transition will be performed.

S_SIG and R_SIG can be used to synchronize state machines: passing a common global variable, as argument at invocation of the two state machines will furnish the synchronization variable.

5.3.1.4.6 TERMINATE

The TERMINATE action has no parameters.

When a state contains a TERMINATE action, and the state machine enters that state, the script will be stopped and the termination state will be presented. The current states of other state machines do not matter. If the test script is run from a test batch, the name of the termination state will be logged as the reason of termination.

For the TERMINATE transition, a destination state may be specified, but this state is of course meaningless. It is also allowed to use the following construction:

```
    s12 : TERMINATE.
```

Which is a TERMINATE followed by a period.

5.3.1.4.7 TIMER's

One parameter is allowed for the non-random timers, two parameters are required when a random timer is defined. It must be a variable or an integer constant.

TIMER has a time scale is 1.00 seconds unless the variable is of a TIME type and another *`timescale`* is specified in the *`typedef`*.

MTIMER has a fixed time scale of 0.001 seconds.

RTIMER is a timer with a random time-out, which lies between the two limits, specified and uses a fixed time scale of 1.00 seconds.

RMTIMER is a timer with a random time-out, which lies between the two limits, specified and uses a fixed time scale of 0.001 seconds.

On slow computers it can be necessary to make OTSE run faster by disabling the extra logging functionality in the ini file, thus avoiding the file logging. This is the case if timers seem to wait too long. In general, the precision of the timer is depending on the load of the system processor. On standard Linux the system clock runs at 100Hz, which results in that the shortest time out can be 10 ms.

The transition is blocked until the timer expires.

5.3.1.4.8 WAIT

One parameter is allowed and it must be a variable.

The WAIT action can be used when a variable must be initialized before the next state transition.

This prevents the system to generate a random value for uninitialized variables in following commands or events. It is allowed to include more than one WAIT action in the same state, the first variable that becomes initialized, determines the next state transition.

5.3.2 Testscript section

This section instantiate state machines. Arguments can be passed to each of the instances and they are always passed by *reference*[7]. In fact, all arguments to the state machines should be a variable —not a constant value.

[7] This means that if two state machines have been instantiated with the same variable as argument a change of value in one will affect the other instance too.

When a particular instance of a state machine is created it is also linked to one communication channel.

```
testscript:
     globalvars boardlist

globalvars:
     (empty)
     VAR varlist

boardlist:
     board
     board boardlist

board:
     channelname : statemachine_invokelist .

statemachine_invokelist:
     statemachine_invoke
     statemachine_invoke statemachine_invokelist

statemachine_invoke:
     statemachinename
     statemachinename ( arglist )
```

Different instances of state machines can only exchange information via the calling arguments of the state machines. Each invocation has its own local variables and its own current state.

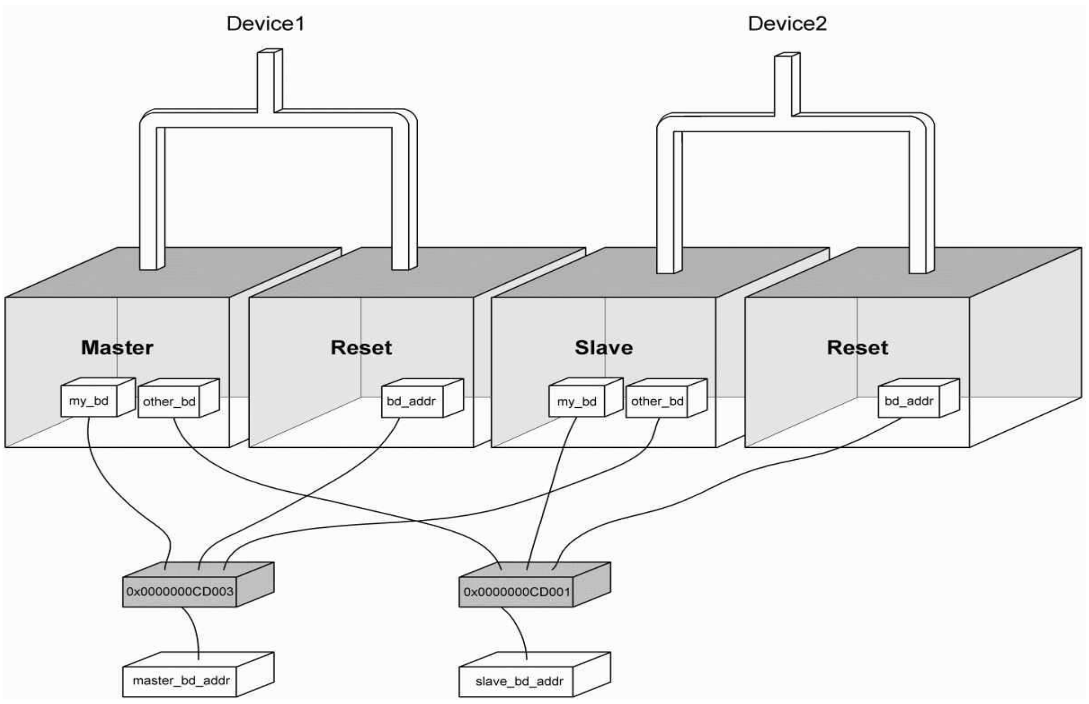
During the test, there will be only one instance of the global variables. State machines instances can be connected to these variables via their arguments.

The `channelname` is the name of the communication channel as defined in the io file.

Example

```
Reset(bd_addr) =
{
 s0 : Reset ; s1.
 s1 : Read_BD_ADDR ; s2.
 s2 : Command_Complete_Event(,Read_BD_ADDR,0x00,bd_addr) ;s2.
}

Master(my_bd,other_bd) =
{
 s0 : WAIT(my_bd) ; s1.
 s1 : WAIT(other_bd); s2.
 s2 : Inquiry(  ...
}

Slave(my_bd, other_bd) =
{
   ...
}

[testscript]
VAR master_bd_addr, slave_bd_addr.

Device1 : Reset(master_bd_addr)
       Master(master_bd_addr,slave_bd_addr).
```

Device2 : Reset(slave_bd_addr)
    Slave(slave_bd_addr, master_bd_addr).



Example 5-4. Example of an HCI-module test.

Four state machines are communicating with two devices. All state machines share the addresses of the two modules. The execution of `Master` and `Slave` is blocked as long as the device's addresses are unknown. It is the task of the two `Reset` instances to furnish those.

5.3.3 Preparser

There is a simple preparser that is used to include other files into the script. A common use is to put state machines definitions that are useful in several test scripts. This will make the code easier to maintain.

The use is very simple; below is an example of how to include the init.tse file into this script.

```
#include "init.tse"
```

Be aware that the order of the different sections is fixed and that it is not possible to divide a section into several parts. The usage is, in practice, to include general state machines into several scripts.

The only other preparser command is the define command:

```
#define timeout_interval 0x06
```

This makes the preparser substitute the string `timeout_interval` with `0x06` further down in the scripts.

5.4 Runtime

5.4.1 Transition selection

One process keeps the state machine instances running. This process has a very simple algorithm for making it look like as all state machine instances are running in parallel.

1. Based on the current state of all state machine instances, it finds all possible transitions. For each instance a decision is taken:

    - If the current state contains commands, one command will be executed.
    - Otherwise, if the current state contains an internal action, this event will be processed.
    - Otherwise, if the current state contains only events (and eventually a timer), the state machine will become pending.

    When OTSE decides to execute a command, one command out of the available commands in that state is chosen on a random base. When OTSE becomes pending, and there is a timer then the timer will be started.

2. Process all events in the input buffer.
3. Check for expiring timers.

These three tasks together form a step as when pressing the “step” button in the user interface. After pressing “Start” OTSE steps in a loop until a state machine performs the `TERMINATE` state transition or “Stop” is pressed.

5.4.2 Processing events

When an event is received via a communication channel, the parameters are checked against the protocol specification. If the parameters are illegal, an error is logged, but processing will continue. The state machine instances linked to the particular channel are selected in the order of appearance in the [testscript] section.

Only the first state machine instance, which is able to process the event, will process the event. If none of the state machines are able to process the event, an error is logged.

When an event is processed, the following happens:

1. The transition that matches the event might have uninitialized variables. These variables are initialized with the values as received in the event.
2. The destination state will become the current state of the state machine.
3. If the new state is different from the original state (in other words: if the state machine goes to a different state) an eventually initialized timer for the state machine is removed.

5.4.3 Expiring timers

When a timer for a particular state machine expires, and the state has not changed, the transition defined in the timer action will be performed. Otherwise, the expiration will be ignored.

5.5 Batch Language

The format of the test batch file is very simple; each line contains the path to one file. The first line contains the filename of the used protocol specification; the rest of the lines contain the filenames of the test scripts. All filenames must include extensions and cannot contain spaces. The last line of the test script file MUST be "`end.`"

Lines starting with // are skipped, and can contain comments.

In the example bellow, the following files are used:

`prottest.testbatch`	*(batch file)*
`prot_1_0.prot`	*(protocol specification)*
`first_test.tse`	*(script file)*
`second_test.tse`	*(script file)*
`MyDir\third_test.tse`	*(script file)*

```
prot_1_0.prot
first_test.tse
second_test.tse
MyDir\third_test.tse
end.
```

Figure 5-1. `prot_test.testbatch`

```
[statemachines]
Master =
{
    s0    : Reset ; s1.

    s1    : Command_Complete_Event ( ,Reset,  0x00) ; ok.
    s1    : TIMER(0x01) ; error.

    ok    : TERMINATE ; ok.
    error : TERMINATE ; error.
}

[testscript]

COM1 : Master.
```

Figure 5-2 `first_test.tse`

```
[statemachines]
Master =
{
  s00 : Write_Scan_Enable(0x03) ; s01.
  s01 : Command_Complete_Event(, Write_Scan_Enable,0x00) ;
ok.
  s01 : Command_Complete_Event(, Write_Scan_Enable,    ) ;
write_failed.
  s01 : TIMER(0x01) ; timeout.

  ok           : TERMINATE ; ok.
  write_failed : TERMINATE ; ok.
  timeout      : TERMINATE ; ok.
}

[testscript]

COM2 : Master.
```

Figure 5-3 `second_test.tse`

The contents of the prot_1_0.prot file are not printed because the file is too long. After starting the `prot_test.testbatch` file, OTSE loads the PROT specification, and starts executing the first test script file. For this test script file, a log file is generated, called `first_test.log`. When the test is completed, the following line of the testbatch file is read, and the next test script is executed. Absolute and relative paths to the .prot-file and the scriptfiles are allowed. When the test is completed, the testbatch file is processed completely, and the system stops. At this point, the following files are added:

```
first_test.log
second_test.log
MyDir\third_test.log
prot_test.log
```

The file prot_test.log contains a summary of the test:

```
Script Engine
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Script Engine 3.0
Copyright (c) 2005
Ordina Technical Automation BV
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Build: Dec 21 2004

Batch started at: 11:14:07

Started at  Status               Script
15:04:27    ok                   first_test.tse
15:04:32    ok                   second_test.tse
15:04:55    ok                   MyDir\third_test.tse
```

Figure 5-4 generated file : `prot_test.log`

If one of the test scripts shows an unexpected result, the corresponding log file can be examined to find out the reason of that result.