

User's Manual - Bluetooth Script Engine R3A



History

Date	Issue	Comments
November 20, 2001	Created	Initial draft
December 3, 2002	Updated	Version 2.5
April 14, 2003	Updated	Version R3A

THE INFORMATION OF THIS DOCUMENT IS PROVIDED AS IS AND ERICSSON MAKES NO REPRESENTATIONS OR WARRANTIES WHATSOEVER, WHETHER EXPRESS OR IMPLIED, FOR THE ACCURACY OF THE INFORMATION PROVIDED HEREIN. IN NO EVENT SHALL ERICSSON BE LIABLE FOR ANY CONSEQUENCES ARISING FROM THE USE OF THIS INFORMATION NOR FOR ANY INFRINGEMENT OF PATENTS OR OTHER INTELLECTUAL PROPERTY RIGHTS OF THIRD PARTIES WHICH MAY RESULT FROM THE USE. NO LICENSE IS GRANTED BY IMPLICATION OR OTHERWISE UNDER ANY INTELLECTUAL PROPERTY RIGHTS OF ERICSSON.

ALL RIGHTS RESERVED. NO PART OF THIS DOCUMENT MAY BE REPRODUCED IN ANY FORM WITHOUT THE WRITTEN PERMISSION OF THE COPYRIGHT HOLDER. THE INFORMATION IN THIS DOCUMENT IS SUBJECT TO CHANGE WITHOUT NOTICE.

CONTENTS

1	INTRODUCTION	1
1.1	ABOUT THIS DOCUMENT	1
1.2	PURPOSE AND USE	1
1.3	DELIVERY OBJECTS	1
1.4	ABBREVIATIONS	1
1.5	GLOSSARY	2
1.6	REFERENCES	2
2	INSTALLATION AND SETUP	3
2.1	INSTALLATION PROCEDURE	3
2.2	INSTALLING THE LICENSE FILE(S).....	3
2.2.1	<i>License Server</i>	<i>4</i>
2.2.1.1	Installing the License Server	4
2.2.1.2	Starting the License Server	4
2.2.1.3	Windows NT/2000 Service	5
2.2.1.4	Windows 95/98 License Server	5
2.2.1.5	License Server Utility	5
2.2.1.6	Compatibility	5
2.2.1.7	Performance Notes	5
2.2.2	<i>Obtaining license code.....</i>	<i>5</i>
2.2.3	<i>Installing the license code.....</i>	<i>6</i>
2.2.4	<i>Running the licensed product.....</i>	<i>6</i>
2.3	UNINSTALLATION	7
3	BLUETOOTH SCRIPT ENGINE IN A SYSTEM.....	8
3.1	VERIFICATION OF BLUETOOTH HOST STACK IMPLEMENTATIONS	9
3.2	BLUETOOTH SCRIPT ENGINE FOR HCI VALIDATIONS	10
3.3	BLUETOOTH SCRIPT ENGINE PROTOCOL STACK VERIFICATION	11
3.4	BLUETOOTH SCRIPT ENGINE COMPONENTS	12
3.4.1	<i>Protocol Specification</i>	<i>13</i>
3.4.2	<i>Script files.....</i>	<i>14</i>
3.4.3	<i>Batch files</i>	<i>14</i>
3.4.4	<i>Script Interpreter.....</i>	<i>15</i>
3.5	OUTPUT FROM BSE	16
3.6	QUICK START	17
3.6.1	<i>Preparation</i>	<i>17</i>
3.6.2	<i>Test17</i>	
4	USER INTERFACE.....	18
4.1	GETTING STARTED.....	18
4.2	SETTING UP THE BLUETOOTH CONTROL PANEL	18
4.3	SETTING UP THE PROGRAM	18
4.4	MENU SECTION	19
4.4.1	<i>File menu</i>	<i>19</i>
4.4.2	<i>Edit menu</i>	<i>19</i>
4.4.3	<i>Tools menu</i>	<i>19</i>
4.4.3.1	Start	19
4.4.3.2	Stop	19
4.4.3.3	Skip.....	19
4.4.3.4	Step.....	19
4.4.3.5	Reset.....	20
4.4.3.6	Device Mapping dialog	20
4.4.3.7	Looping dialog.....	20
4.4.3.8	Change Default HCI Specification File dialog.....	21

4.4.4	View menu.....	21
4.4.4.1	Log Overview	21
4.4.4.2	Script Overview.....	21
4.4.4.3	Batch/Script Tree	21
4.4.4.4	Log/Script Overview.....	22
4.4.4.5	Disable Logging.....	22
4.4.4.6	Transition Log.....	22
4.4.4.7	Clear Overview windows on start.....	22
4.4.4.8	Auto expand tree.....	Error! Bookmark not defined.
4.4.4.9	Clear Overview windows	Error! Bookmark not defined.
4.4.5	Window menu	22
4.4.6	Help menu	22
4.5	TOOLBAR SECTION	22
4.6	TREE VIEW SECTION	23
4.7	WINDOW SECTION	23
4.8	LOG OVERVIEW SECTION	24
4.9	SCRIPT OVERVIEW SECTION	25
4.10	STATUS BAR SECTION	25
4.11	LOG FILES.....	26
5	EXECUTION OF PASS/FAIL TESTS.....	27
5.1	INTRODUCTION	27
5.2	TEST PROCEDURE.....	27
6	SCRIPT LANGUAGE INTRODUCTION.....	29
6.1	STATE MACHINES	29
6.2	STATE MACHINE EXAMPLE.....	30
6.3	PROTOCOL SPECIFICATION.....	31
6.4	STORAGE AND SYNCHRONIZATION.....	32
6.5	USE OF FILES	32
7	SCRIPT LANGUAGE REFERENCE	33
7.1	INTRODUCTION	33
7.1.1	Conventions used in this chapter.....	33
7.1.2	White space.....	33
7.1.3	Keywords	34
7.1.4	Identifiers.....	34
7.1.5	Constants.....	35
7.2	PROTOCOL SPECIFICATION	35
7.2.1	Type section	36
7.2.1.1	Time definition	36
7.2.1.2	Enum definition.....	37
7.2.1.3	Bitfield definition	37
7.2.1.4	Command definition.....	37
7.2.2	Functions section	38
7.2.2.1	Union definition	38
7.2.2.2	Array parameters	39
7.2.3	Events section.....	39
7.3	SCRIPT LANGUAGE	39
7.3.1	Data section	39
7.3.2	Statemachines section	40
7.3.2.1	Grammar	40
7.3.2.2	Variable declaration.....	42
7.3.2.3	Smart strings	42
7.3.2.4	Advanced flow control.....	44
7.3.2.5	Internal actions / Keywords.....	47
7.3.3	Testscript section	50
7.3.4	Preparser.....	51

7.4 RUNTIME 52

 7.4.1 *Transition selection* 52

 7.4.2 *Processing events*..... 52

 7.4.3 *Expiring timers* 53

7.5 BATCH LANGUAGE 54

TABLE OF TABLES

TABLE 1-1. ABBREVIATIONS. 1

TABLE 7-1. THE MANDATORY SECTIONS FOR THE SCRIPT AND PROTOCOL SPECIFICATION FILES..... 33

TABLE 7-2. THE SEVEN SECTIONS AND THEIR KEYWORDS. 34

TABLE OF FIGURES

FIGURE 2-1. COMPUTERS USING STAND-ALONE LICENSES	3
FIGURE 2-2. COMPUTERS USING A LICENSE SERVER	4
FIGURE 2-3. SETTING ENVIRONMENT VARIABLE LSFORCEHOST FOR WIN2000	7
FIGURE 3-1. TESTING BLUETOOTH HOST STACK IMPLEMENTATIONS	9
FIGURE 3-2. USE OF TWO HCI MODULES IN TESTING	10
FIGURE 3-3. PROTOCOL TESTING OF BLUETOOTH HOST STACK	11
FIGURE 3-4. SCRIPT ENGINE COMPONENTS	12
FIGURE 3-5. EXAMPLE OF A HCI FILE	13
FIGURE 3-6. EXAMPLE ON A STATE MACHINE	14
FIGURE 3-7. EXAMPLE OF A BATCH FILE	14
FIGURE 3-8. MAIN GUI FOR SCRIPT ENGINE	16
FIGURE 3-9. THE BUTTONS YOU NEED TO GET STARTED HAVE BEEN MARKED IN RED.	17
FIGURE 4-1. BLUETOOTH SCRIPT ENGINE USER INTERFACE	18
FIGURE 4-2. MAPPING OF DEVICES TO CHANNELS	20
FIGURE 4-3. LOOPING DIALOG	21
FIGURE 4-4. BSE TOOLBAR	23
FIGURE 4-5. BSE TREE VIEW	23
FIGURE 4-6. OVERVIEW WINDOW. EVERY LINE IS PREFIXED BY A TIME STAMP (PRESENT TIME) IN THE FORMAT HOUR : MIN : SEC : MILLISECOND.	24
FIGURE 4-7. BSE STATUS BAR	25
FIGURE 5-1. BLUETOOTH VERIFICATION TEST PROCEDURE	27
FIGURE 6-1. 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.	29
FIGURE 6-2. A STATE MACHINE THAT SENDS OUT A COMMAND AND WAITS FOR AN EVENT FROM THE HCI- MODULE. THE FIGURE IS SIMPLIFIED; THE CONNECTION BETWEEN THE STATE MACHINE AND THE EUT IS ACTUALLY INDIRECT, PASSING THROUGH A COMMUNICATION INTERFACE.	30
FIGURE 6-3. THE RELATION BETWEEN ABSTRACT COMMANDS/EVENTS AND I/O IS CONTROLLED BY THE PROTOCOL SPECIFICATION FILE.	31
FIGURE 6-4. TWO STATE MACHINES THAT ARE SYNCHRONISED VIA THE VARIABLE FLAG. THE TRANSITION B1→B2 IS BLOCKED UNTIL ANY OTHER STATE MACHINE EXECUTES A 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.	32

1 INTRODUCTION

1.1 About this Document

This document is the user's manual for Bluetooth Script Engine (BSE) R3A. It is written for users with knowledge of Bluetooth technology and describes the interface and functionality provided by the BSE. It also covers the set-up procedure of the tool.

This chapter explains the purpose and the benefits of using the Bluetooth Script Engine by Ericsson. 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 BSE is to provide a development tool that makes it possible to verify Bluetooth adaptations and porting of products. BSE can be a useful tool in the development phase as it can automatically perform multiple test cases by the use of scripts.

With BSE you can perform PASS/FAIL tests by using pre-validated scripts from Ericsson. BSE also provides the ability to execute customer-defined scripts. Testing can be done towards HCI modules, or for verification of the Ericsson Bluetooth HOST Stack [2] and layers within the stack. The Bluetooth Script Engine can also be a component when verifying Bluetooth Single Processor solutions where the Bluetooth HOST Stack is one building component.

Intended customers for Bluetooth Script Engine are, OEM customers, licensees and partners that adapt/ports the Bluetooth HOST Stack. The Bluetooth Script Engine will also be used internally within Ericsson when developing Bluetooth technology.

1.3 Delivery Objects

A delivery contains the following objects:

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

1.4 Abbreviations

Abbreviations	Explanation
BCP	Bluetooth Control Panel
BHT	Bluetooth HCI Toolbox
BLA	Bluetooth Log Analyzer
BNF	Bachus Naur Form
BSE	Bluetooth Script Engine
EUT	Equipment Under Test
GUI	Graphical User Interface

Table 1-1. Abbreviations.

1.5 Glossary

1.6 References

- [1] Title: Bluetooth Special Interest Group, Specification of the Bluetooth System, version 1.0B, volume 1, available from www.bluetooth.com, Dec 1999.
Doc no.:
- [2] Title: Ericsson, Bluetooth HOST Stack.
Doc no.: LZT 108 5392 3.10
- [3] Title: Users Manual – Bluetooth Log Analyzer.
Doc no.: 198 17-CXC 132 2168 Uen
- [4] Title: Users Manual – Bluetooth HCI toolbox.
Doc no.: 198 17-CXC 125 51 Uen
- [5] Title: Ericsson, Bluetooth HCI Script Kit.
Doc no.: 198 17-CXC 101 053 Uen
- [6] Title: Users Manual – Bluetooth Application Script Kit.
Doc no.: 198 17-CXC 109 666 Uen
- [7] Title: Bluetooth Control Panel User's Manual
Doc no.: 198 17-CXC 125 390 Uen

2 INSTALLATION AND SETUP

2.1 Installation Procedure

BSE is delivered with an installation program. This helps you to both install and uninstall. Start the installation by executing the file "Setup Bluetooth Script Engine R3A.exe", and follow the on-screen instructions.

2.2 Installing the license file(s)

This product is protected by a License Manager, you need to obtain a license from Ericsson to use this product.

Both Stand-alone and Networked licensing is available in this product:

- Stand-alone licensing. The product is licensed to run on a single computer without using a computer network. No license server is needed.

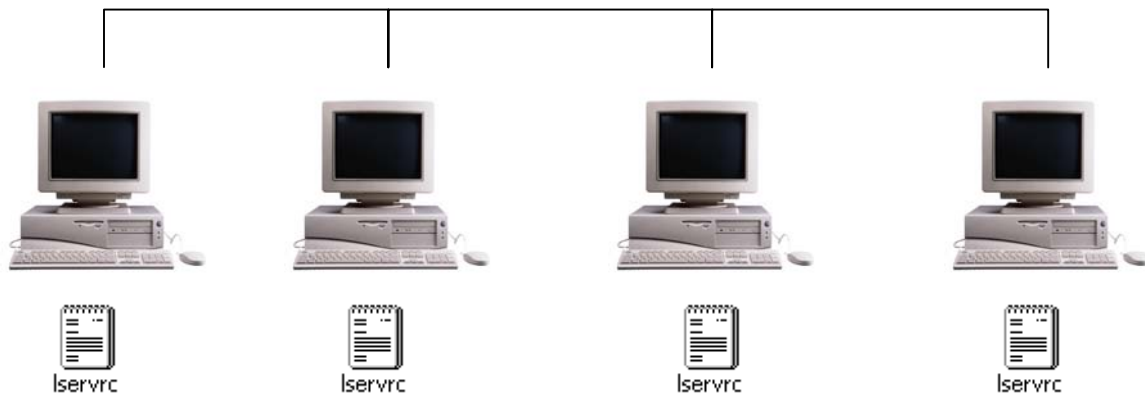


Figure 2-1. Computers using stand-alone licenses

- Networked licensing. The product is licensed to run on multiple computers connected in a network. In this case, a computer in the network must be designated as a *license server* computer on which the license server is running to authorize the licenses.

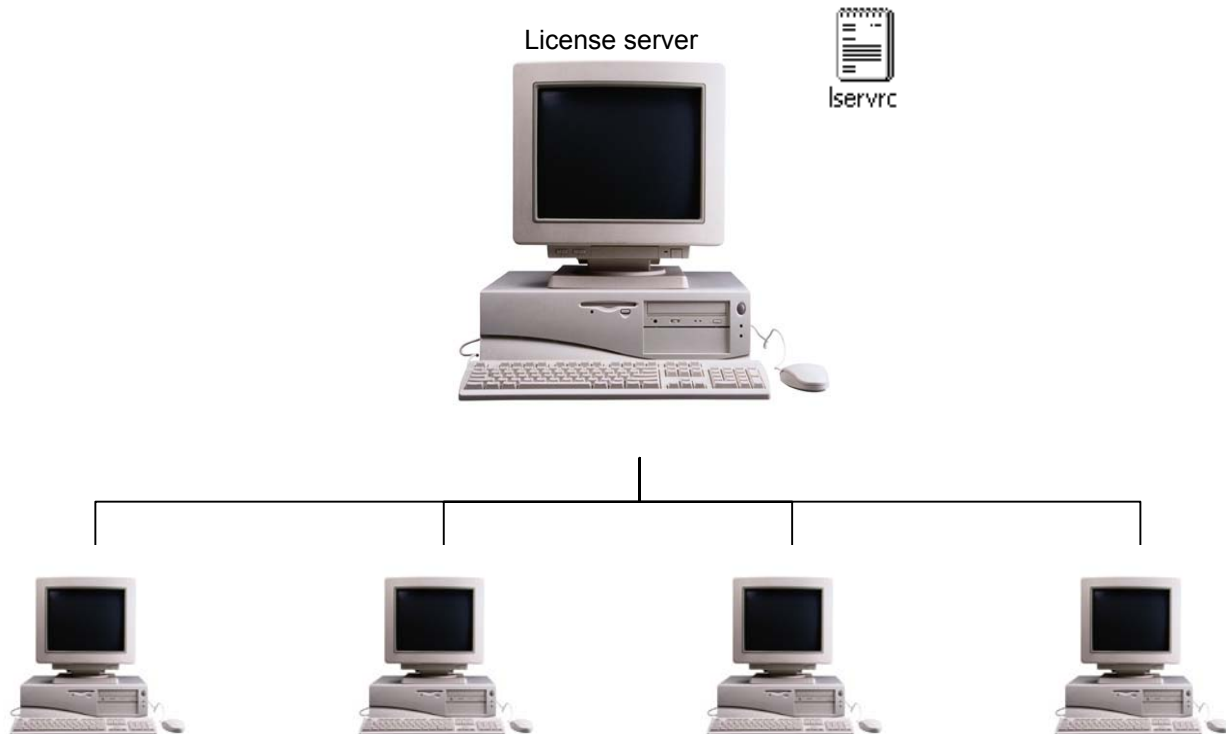


Figure 2-2. Computers using a license server

2.2.1 License Server

2.2.1.1 Installing the License Server

Select a computer as server computer. The server computer must be available on the network that the client computer (the computers where you want to run license protected application) connects to. The server computer can also be one of the client computers. The license server is installed using a separate installation program, called *License Server Setup*, located in the Start-menu under *Ericsson | Bluetooth Control Panel R1C | License Tools*. Run *License Server Setup* to install license server. The license server is installed with program file named *lserv9x.exe* for Windows 9x or *lservnt.exe* for Windows NT/2000.

2.2.1.2 Starting the License Server

License server installation and startup vary depending upon the operation system on the license server computer.

- Windows 95/98 User Privileges: The same user who starts the license server may also configure the license server and stop the license server. Before allowing these actions, SentinelLM not only checks that the user name is the same, but also makes sure that the user is in the same network domain as the license server.
- Windows NT/2000 User Privileges: Any user who has administrator privileges may start the license server and may also configure the license server regardless of who started it. Any user with administrators privileges can also use *lsrvdown* (located in the same folder as license server) to shut down the license server unless the license server was started

by a user with the user name "Administrator", in which case only "Administrator" can use *lsrtdown* to shut it down.

2.2.1.3 Windows NT/2000 Service

The license server installation is done using the *loadls* utility (located in the same folder as license server). Once loaded, the license server can be configured and started from the Control Panel | Services group. Automatic or manual license server startup can be selected.

2.2.1.4 Windows 95/98 License Server

A windows 95/98 license server can be configured to be started automatically when Windows 95/98 starts or to be started manually. To enable automatic license server startup, create a shortcut of the license server program, *lserv9x.exe*, and add it to the Windows 95/98 system StartUp folder.

To start the license server manually, you simply run *lserv9x.exe*.

2.2.1.5 License Server Utility

- *lsmon*

lsmon.exe is used to display the information about the licenses on the license server. To use it, just run License Server Monitor under *Ericsson | Bluetooth Control Panel R1C | License Tools*. It works for windows 95/98, Windows NT/2000.

- *loadls*

loadls.exe is used to install the license server as an NT/2000 service, it is for Windows NT/2000 only. To use it, just run it.

- *lsrtdown*

lsrtdown.exe is used to shut down the license server. It works for windows 95/98, Windows NT/2000.

Format : *lsrtdown host-name*

host-name specifies the name of the computer that is running the license server that you want to shut down.

2.2.1.6 Compatibility

A license server running on a computer with one operating system can communicate with a client running on another operating system.

2.2.1.7 Performance Notes

High performance can be achieved by the followings settings:

- The computers on which license servers run should use static IP address rather than dynamically allocated IP address (DHCP).
- Client computers should be configured to directly contact the license server that services the licenses used by that client. The LSHOST and LSFORCEHOST environment variables can be used to do this (Assign LSFORCEHOST or LSHOST with value of the host name of server computer).
- Better performance can be achieved by running license servers on Windows NT/2000 rather than Windows 95/98.

2.2.2 Obtaining license code

- Stand-alone licensing. First, you must generate the locking code for the computer where you want to run this product. To generate the locking code, run *echoid.exe*, located in the start-menu under Ericsson\Bluetooth Control Panel. Send the locking code to Ericsson techsupport.bluetooth@ebt.ericsson.se. In return, you will get the corresponding license code.

- Networked licensing. The basic procedure for this license type is the same as for Stand-alone licensing, except that you only need to generate the locking code for the license server computer.

2.2.3 Installing the license code

The license code will come in the form of a license file.

- Stand-alone licensing. Rename the license file to */servrc*, put it in the same directory as the BLA executable.
- Networked licensing. Put */servrc* to the same directory as the license server on the server computer. Every time */servrc* is updated, license server must be restarted to reload the updated license codes to the server.

If you get multiple licenses for this product, simply add them to license file */servrc* by using a normal text editor, where each license code uses one line.

2.2.4 Running the licensed product

Stand-alone licensing. Just run the product directly after above procedures are finished.

Networked licensing. First you need to set the environment variable depending on the operative system according to the steps below:

- Windows98: add *SET LSFORCEHOST=HostName* to the file autoexec.bat and reboot the computer
- WindowsNT/2000: set LSFORCEHOST environment variable to the license server computer's host name ,see Figure 4-3

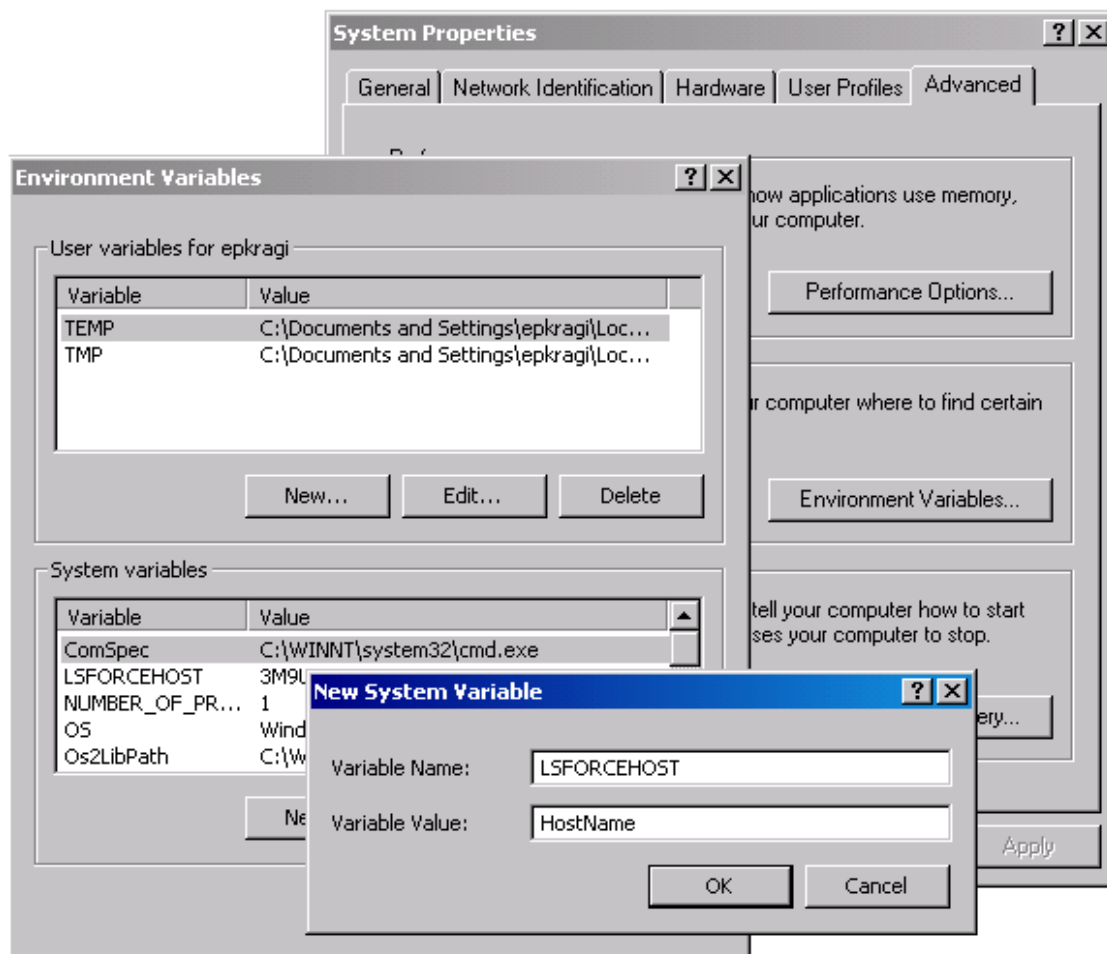


Figure 2-3. Setting Environment variable LSFORCEHOST for Win2000

When license server is running, run the licensed product from the client computer.

2.3 Uninstallation

To uninstall Bluetooth Script Engine start your control panel and select "Add/Remove programs". Select Bluetooth Script Engine R3A and press the button "Add/Remove...". Follow the instructions on screen to remove (uninstall), repair or modify your installation.

3 BLUETOOTH SCRIPT ENGINE IN A SYSTEM

Bluetooth Script Engine is a software tool that runs on Windows 98/NT/2000. BSE will be one central component when developing and testing new Bluetooth systems.

The Bluetooth Foundation Specification proposes two different solutions representing two different architectures:

- **The Single-processor Solution** (or One-processor Solution)
The entire Bluetooth protocol stack executes in one processor.
- **The Two-processor Solution** (or HOST Solution)
The Bluetooth protocol stack is split in two parts. Each part executes in a separate piece of hardware (or processor). The lower level Bluetooth protocols executes in a piece of hardware referred to as the **HCI Module**.
The higher-level protocols executes in another piece of hardware, usually called HOST, which also runs the application. The two parts communicates via HCI. In this context **Bluetooth HOST Stack** represents the upper part.

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

3.1 Verification of Bluetooth Host stack implementations

Bluetooth Script Engine will take the central role in verifying portings of the **Bluetooth HOST stack** into customer environment. BSE will run in the PC environment; the ported protocol stack is running in customer HOST environment. The customer's HOST environment is connected via serial ports. This test set-up also requires Bluetooth Application Script Kit [6].

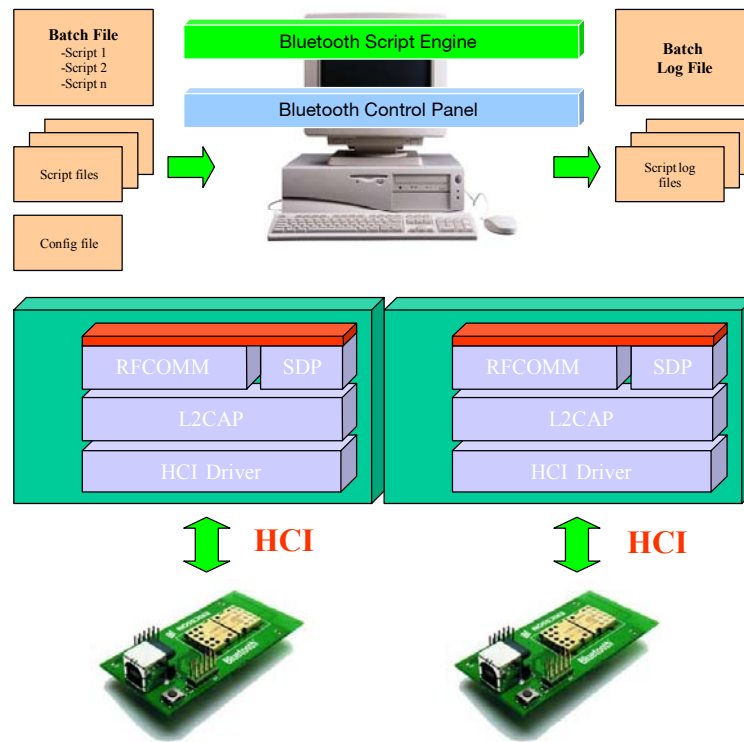


Figure 3-1. Testing Bluetooth HOST stack implementations

Input files for the test are Batch and Script files (Bluetooth Application Script Kit) and a configuration file. Output files are batch and script loggings. Communication with Equipment Under Test (EUT) is done via the Bluetooth Control Panel, see ref [7] for more information.

In this example the script will generate Bluetooth HOST stack API commands. Incoming data (events) are received and analyzed by BSE. The purpose of this test set-up is to verify the functionality of the whole chain — from Host Stack API via air interface to Host Stack API on the other device — after having adapted/porting the stack.

The Bluetooth Application Script Kit is covered in its own manual [6].

3.2 Bluetooth Script Engine for HCI validations

The main purpose for this two-processor configuration is to verify new hardware (for example a new HCI module) and firmware by using validated scripts from Ericsson.

The script files can automatically send HCI commands and validate the data received. Bluetooth HCI Script Kit [5] is also required.

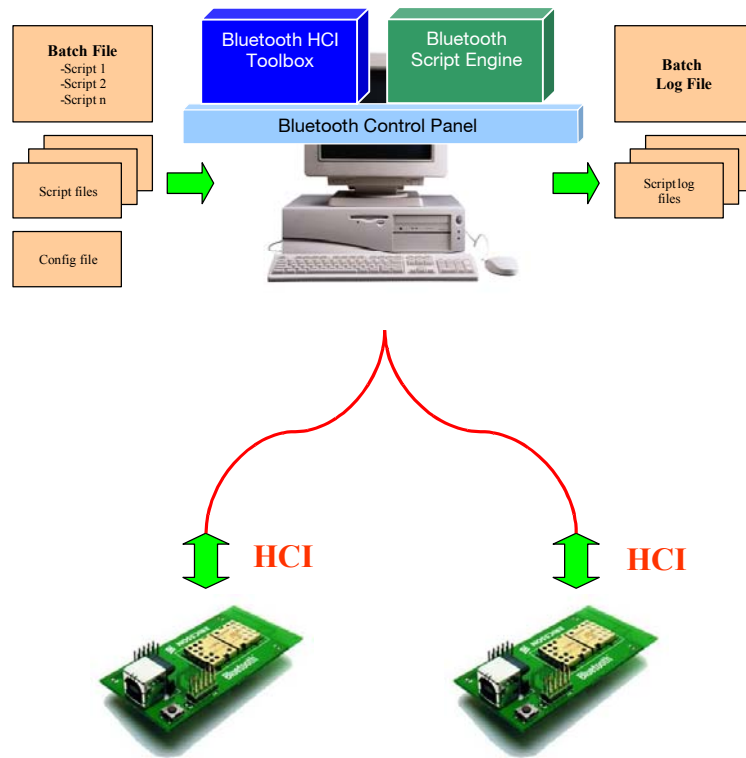


Figure 3-2. Use of two HCI modules in testing

In this test set-up the batch and script files are validating the HCI and radio interface of the modules. The configuration can for example be used for PASS/FAIL tests on new hardware.

Output files are batch and script loggings. Communication with EUT is done Bluetooth Control Panel (BCP) [7].

3.3 Bluetooth Script Engine protocol stack verification

In this configuration the Bluetooth HOST Stack is the target for the test. The test can be executed towards individual layers of the stack. The scripts and the debug components are further described in dedicated manuals, see ref. [6].

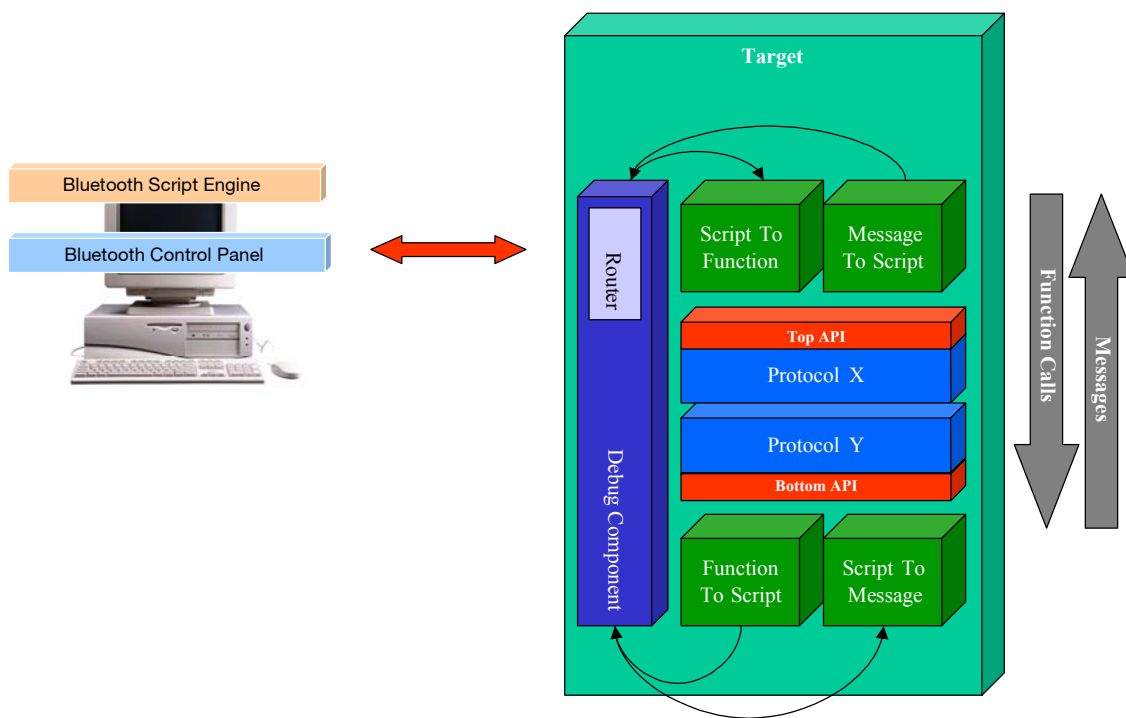


Figure 3-3. Protocol testing of Bluetooth HOST Stack

3.4 Bluetooth Script Engine components

Bluetooth Script Engine consists of the following parts:

- HCI-file (Protocol Specification)
- Scripts
- Batch files
- Script Interpreter executable, called Script Engine

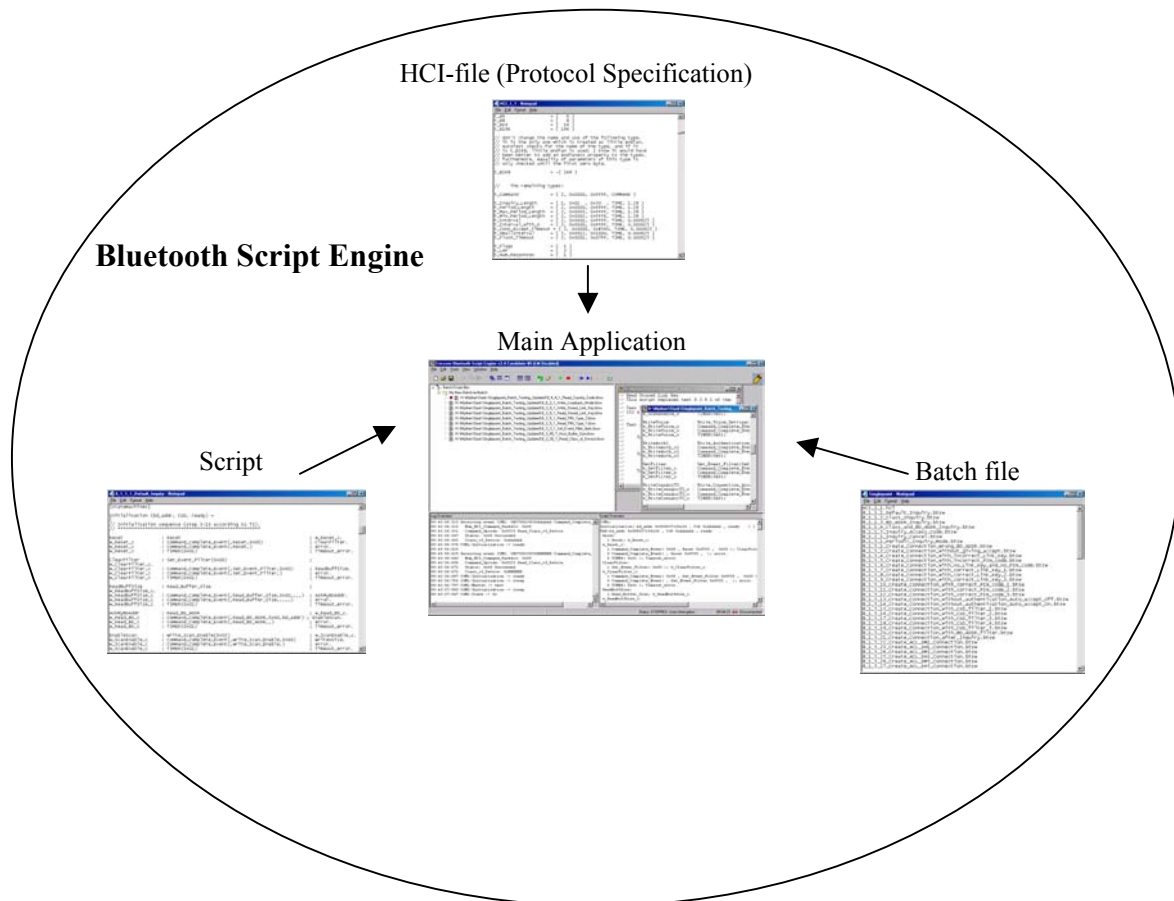


Figure 3-4. Script engine components

Protocol Specification, scripts and the batch file are stored in separate external text files, and can be modified by the user. The Script Interpreter is an executable.

3.4.1 Protocol Specification

The Protocol Specification¹ consist of three parts:

A **type** definition part. Data types are defined to have a specified size and are the building block for commands and events.

A **function** definition part. Commands are defined with their parameters. In this section all possible output to the I/O-ports are given.

An **event** definition part. Events are defined with their parameters. Here all valid input from I/O-ports are presented.

The scripting and the layout of the configuration layout will be further described in chapter 6.

```
[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 3-5. Example of a HCI file

¹ In some documents called “configuration file” or “HCI file”

3.4.2 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 3-6. 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 6 and 7.

3.4.3 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 7.5.

```
hci 1 0 l2cap.hci
timer_null2.bts
timer_null.bts
end.
```

Figure 3-7. Example of a Batch File

In the example above is **hci_1_0_l2cap.hci** the configuration file, **timer_null2.bts** and **timer_null.bts** are two scripts, which will be executed, in sequential order.

3.4.4 Script Interpreter

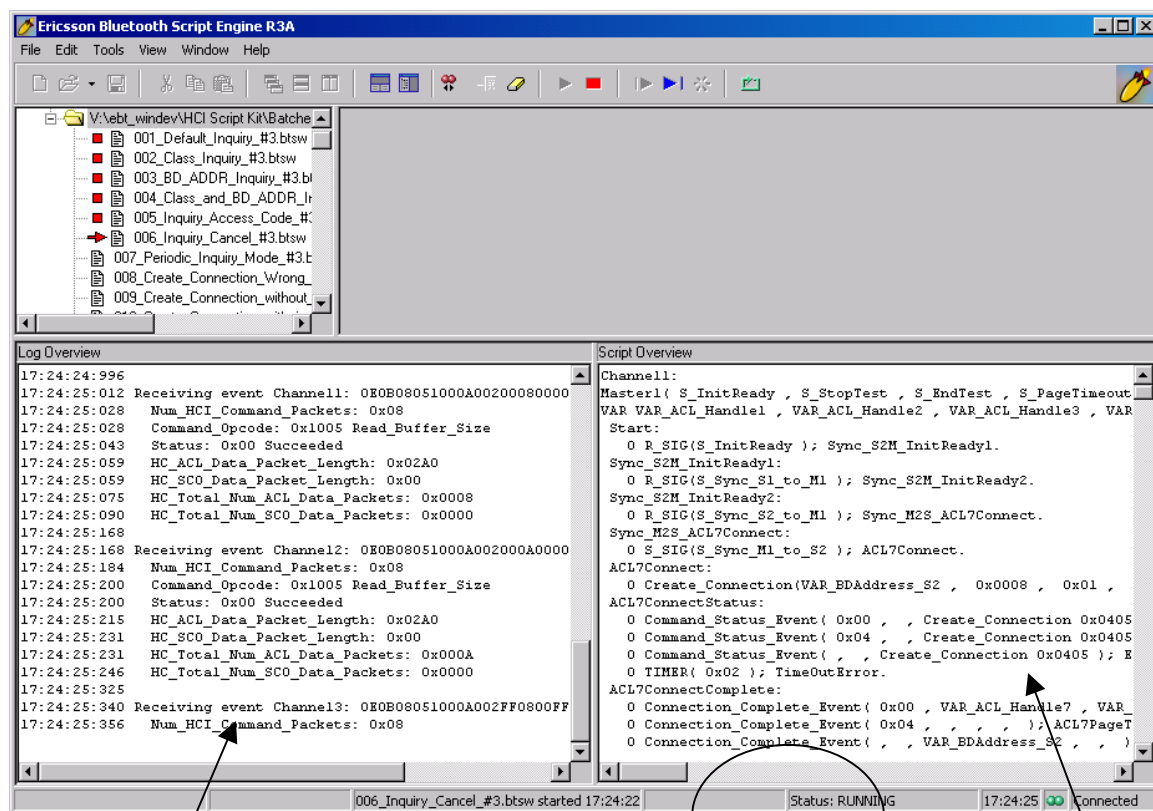
The script interpreter “Bluetooth Script Engine” is an executable .exe file. It can be started from the Start menu.

3.5 Output from BSE

The output from Bluetooth Script Engine is logging information from state transitions, commands and received events. You can receive the information in the output logging window² of the tool in real time. It is also possible to save all the information in log files, for later analyze.

In the script overview window is it possible to see the current state of the state machines. There are several possibilities to combine and select the level of information logged. This will further be discussed in chapter 4.9.

There is also a state panel in the status bar where you can see the state of the running script. The information in this panel is useful in PASS/FAIL tests as it can present the result of the execution of the script, for example "ok" or "error". The "stop" information is however dependent of the design of the script, the tester have to read the documentation for the script in order to know the "PASS" or "FAIL" conditions for a specific script.



Logging Window

Status Indicator

Overview window

Figure 3-8. Main GUI for Script Engine

² The logging in the output window can be disabled.

3.6 Quick start

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

3.6.1 Preparation

Before starting BSE, make sure you have configured Bluetooth Control Panel by assigning EUTs to channels. For more information how to use Bluetooth Control Panel, see ref [7].

Start BSE, and if it is the first time you are using BSE after the installation, you have to change the default HCI-file. This is done by clicking Tools->Change Default HCI Specification File.

If you have modified the EUT-channels in the Bluetooth Control Panel, you may have to change the mapping of devices to channels in BSE. This is done by clicking Tools->Device Mapping or by clicking the corresponding button in the toolbar. See chapter 4.4.3.6 for further information.

3.6.2 Test

Start with either open batch/script file(s) manually by pressing the File->Open menu item or by clicking the corresponding button in the toolbar or by drag-and-drop the file(s) from Windows Explorer.

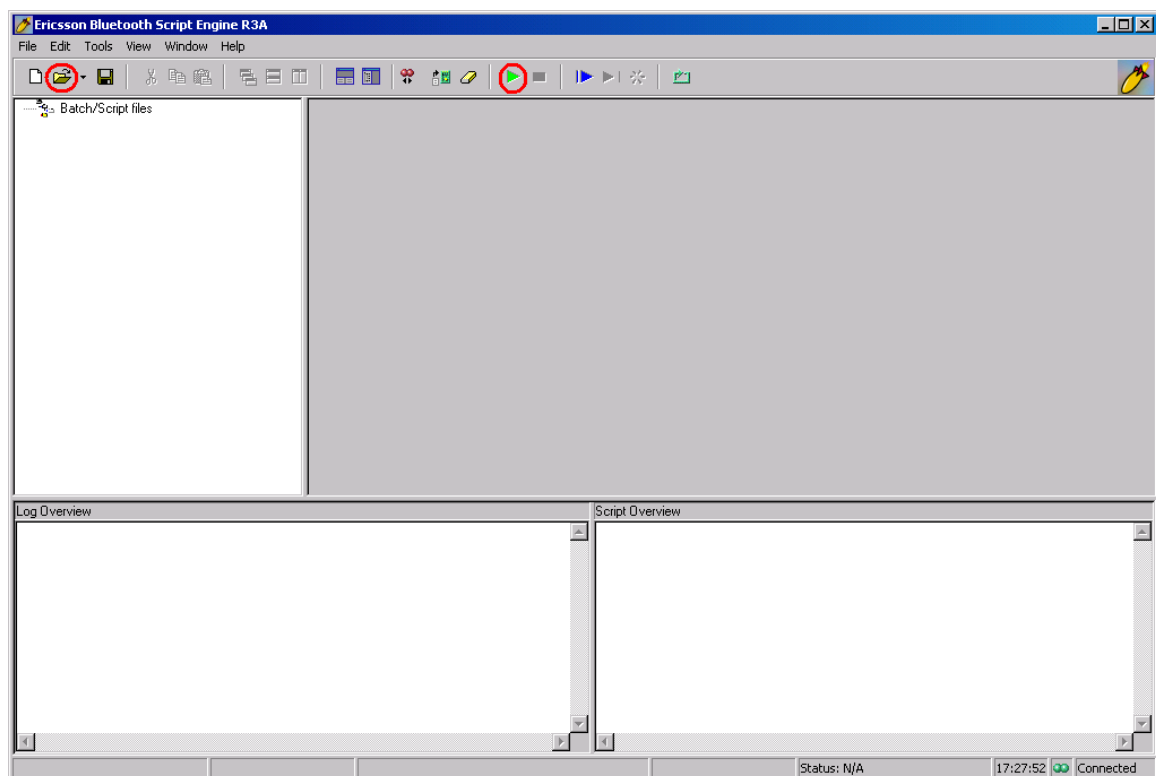


Figure 3-9. The buttons you need to get started have been marked in red.

The last step before actually starting the batch/script file is to highlight the batch/script file in the tree view to be executed. If not highlighted, BSE will automatically use the last loaded batch/script file as default. By pressing the menu item Tools->Start, key F5, or by clicking the corresponding button in the toolbar, the batch/script starts running.

4 USER INTERFACE

4.1 Getting started

BSE is a Windows program, making use of a typical Windows graphical user interface (GUI). It is based on a multi-document interface, where the application is able to spawn child windows residing inside the workspace. The application is designed to support any screen resolution, but is best viewed at resolutions above 1024*768.

4.2 Setting up the Bluetooth Control Panel

Prior to using the BSE you must configure how you communicate with the devices you want to connect. This could be done using COM-ports or USB. This is done in the BCP. Use the user's manual of BCP [7] first and then return to this manual.

4.3 Setting up the program

An example of the Bluetooth Script Engine main window could look like this when being used:

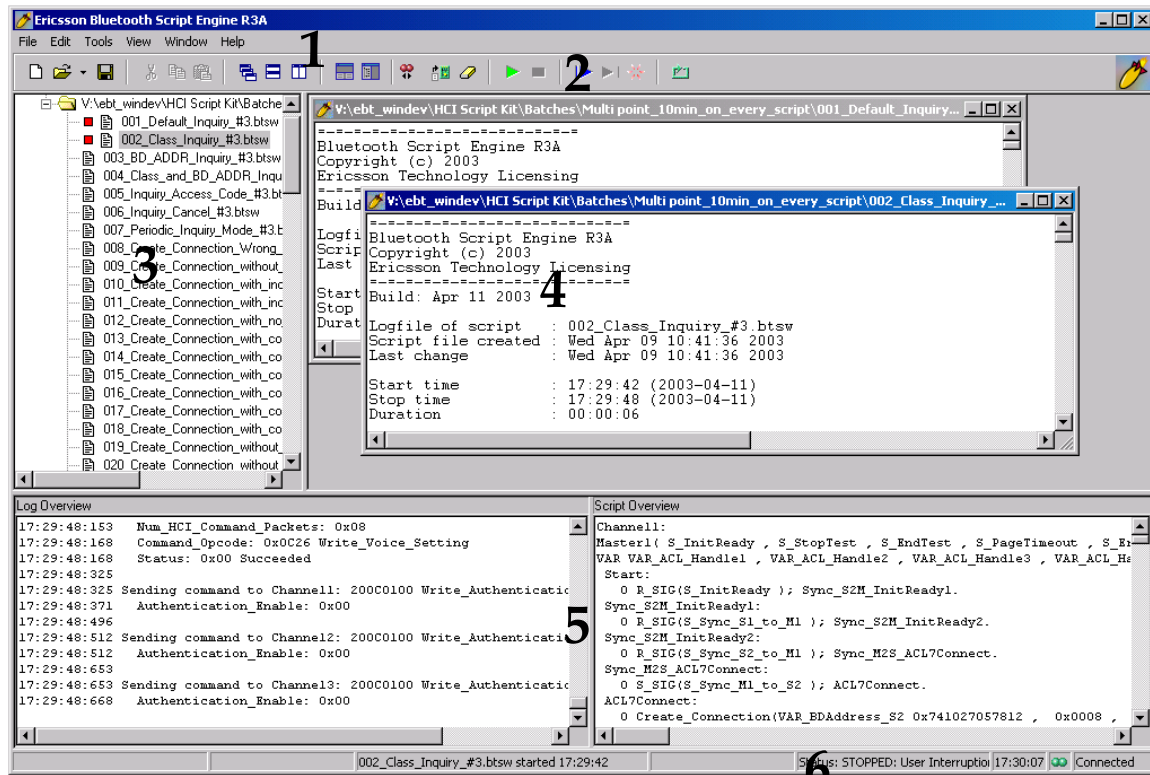


Figure 4-1. Bluetooth Script Engine user interface

The user interface has six major parts: The “menu” section (1), the “toolbar” section (2), the “tree view” section (3), the “window” section (4), the “overview” section (5) and the “status bar” section (6).

4.4 Menu section

4.4.1 File menu

The file menu includes standard Windows file related menu items such as New, Open, and Close, Close All Windows, Save File and Exit.

4.4.2 Edit menu

The edit menu includes standard Windows file related menu items such as Undo, Cut, Copy, Paste, Delete, and Select All.

4.4.3 Tools menu

This menu includes BSE specific menu items such as script controlling and settings dialogs. In the script control section, a script can be started, stopped, skipped, single-stepped and reset.

4.4.3.1 Start

When “Start” is pressed BSE loads the script from file before the actual execution. This does not happen if the script already is loaded. However, if the script is modified BSE recognizes this by looking on the time stamp and load the new script.

When a script is started, all state machines are activated and all incoming events are processed. Incoming events that arrived before a script is started are flushed.

The active state of the state machines is not initialized before starting, which means that when a stopped script is restarted, state machines continue to run from their last active state. The same applies for the value of variables. Variables are not reinitialized before a start.

4.4.3.2 Stop

When a script is stopped, the state machines generate no outgoing commands anymore. Incoming events are flushed³.

The script log file is closed upon script stop.

4.4.3.3 Skip

When running batch files, invoking skip will stop the current running script and the batch processing will continue.

4.4.3.4 Step

Single stepping through a script has only limited use. One single-step click will only influence state machines that are about to perform a command. State machines waiting for an event or having a timer cannot make a single step.

After the single step, script processing stops and incoming events are flushed.

³ The first incoming event after a stop generates a warning, consecutive events are flushed silently.

4.4.3.5 Reset

Resetting a script has two major effects: The actual state will be reset to the first state mentioned in the state machine definitions — and all signals and variables, that are not pre-initialized in the script, will be cleared. Additionally, all state images of the tree view will be cleared.

4.4.3.6 Device Mapping dialog

This dialog is used for the mapping of devices to channels. In this dialog 16 different devices could be mapped to 16 different channels. The channels are set prior to starting BSE using BCP, see ref [7] for further information. Only available and free (not reserved) channels are visible in the drop-down list. The user is able to name the devices according to the device names in the [testscript] section of the script files. See section 7.3.3 for further information regarding the [testscript] section. Also, the user can choose from five different protocols.

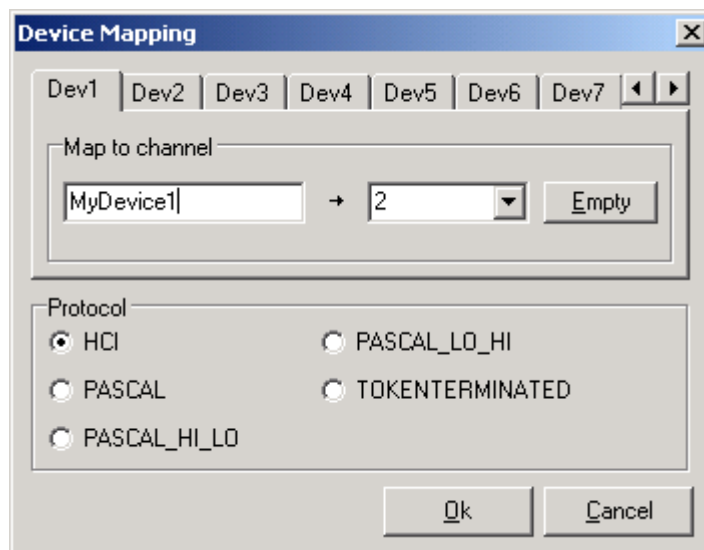


Figure 4-2. Mapping of devices to channels

4.4.3.7 Looping dialog

Using the looping dialog, the user is able to specify the number of times a certain batch/script file is to be executed. If radio button item “Indefinitely” is chosen, the selected batch/script will execute until execution counter reaches INT_MAX⁴. If checkbox “Stop on errors” is checked, the looping of script/batch will be stopped when error occurs, otherwise BSE will continue looping regardless.

⁴ INT_MAX = 2147483647

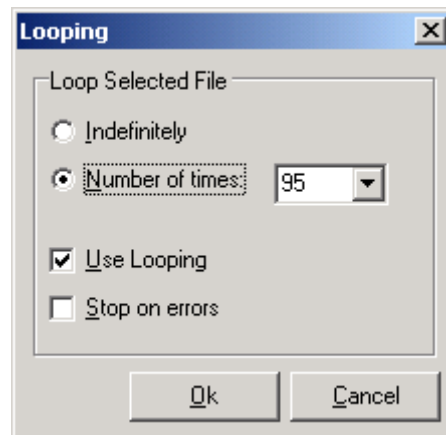


Figure 4-3. Looping dialog

4.4.3.8 Change Default HCI Specification File dialog

This is an ordinary open dialog where the user can open and load a new HCI-file. If there are errors in the HCI-file, the user is prompted.

4.4.3.9 OSE Crash Decoder

This feature translates an OSE crash event.

4.4.3.10 Use new fatal error format

When this option is checked, the BSE will search for the magic sequence that comes before any fatal error event.

4.4.4 View menu

The view menu contains BSE viewing options such as log/script overview selection, hide/unhide views and disabling/enabling of viewing options.

4.4.4.1 Log Overview

Using this menu item, the user can set the logging mode to be used. The different modes the user can select from are Compact log, Extensive log and String log. More information of the log overview window can be found in chapter 4.8.

4.4.4.2 Script Overview

Using this menu item, the user can set the script overview mode to be used. The different modes the user can select from are Show bodies, Show headers and Show test case. More information of the script overview window can be found in chapter 4.9.

4.4.4.3 Batch/Script Tree

Hides or shows the batch/script tree.

4.4.4.4 Log/Script Overview

Hides or shows the overview view.

4.4.4.5 Disable Logging

It is possible for the user to disable the logging, if BSE runs to slow. In some scripts, where a lot of log text is produced, the screen updates can cause timers to lose precision.

4.4.4.6 Transition Log

When this option is checked, a log line of each transition is generated. More information about transition log option can be found in chapter 4.8.

4.4.4.7 Clear Overview windows on start

When start button is pressed the log overview window is cleared if this option is checked.

4.4.4.8 Parse Batch file on start

When this option is checked, the BSE will verify that all scripts included in the batch are available before it starts executing the batch file.

4.4.4.9 Stop on error

When this option is checked, the current execution of a batch will stop if an error occurs.

4.4.4.10 Include script header in log

When this option is checked, the BSE will include all comment-lines in the top of the script file in the produced log file.

4.4.4.11 Clear overview windows

Clear the overview windows.

4.4.5 Window menu

The window menu includes ordinary window handling routines such as cascade, tile horizontally, tile vertically, minimize all and arrange all.

4.4.6 Help menu

In the Help menu the user can access the User's Manual, as well as an *About* entry with information about the application.

4.5 Toolbar section

This section includes the most commonly used menu items. Figure 4-4 illustrates the toolbar with the following buttons in left-to-right order: New, Open, Save File, Cut, Copy, Paste, Cascade, Tile Horizontally, Tile Vertically, Batch/Script Tree, Log/Script Overview, Device Mapping, Start, Stop, Step, Skip, Reset and Looping.



Figure 4-4. BSE Toolbar

Note that when pressing the button for looping, the “use looping” checkbox of Figure 4-3 is marked/unmarked; it does NOT open the looping dialog.

4.6 Tree View section

A conceptual structure of the tree view of BSE is illustrated in Figure 4-5. The user is able to create and save new batch- and script files in the tree view. It is possible to drag-and-drop script files between batch files and additionally it is possible to drag-and-drop multiple selections of script/batch files from Microsoft Windows Explorer.

When double clicking a script file a window is spawned in the window section showing the contents of the script. By using the popup menu as illustrated in Figure 4-5 it is possible to view a script's/batch's corresponding log file. Additionally, the same popup menu allows the user to delete the selected node in the tree. If the chosen node is a batch file, the user is also able to view the contents of the batch file by selecting “View Batch file”. By selecting either of the “select/hide” popup menu items, the user is given the possibility to modify which scripts/batches should be executed. If the batch file is edited outside the BSE editor, the user is given the possibility to “reload” the batch file by clicking the “Refresh Batch file” menu item or by pressing the F5-key.

After an execution of e.g. a batch file, the individual scripts are marked with either an “Ok”-sign (green ball), “Error”-sign (red square), or a “Warning”-sign (yellow triangle). See Figure 4-5 for an illustration of this example.

The tree view section can at some times (e.g. batch running) be redundant, and hence the user is given the possibility to hide this view as described in chapter 4.4.4.3.

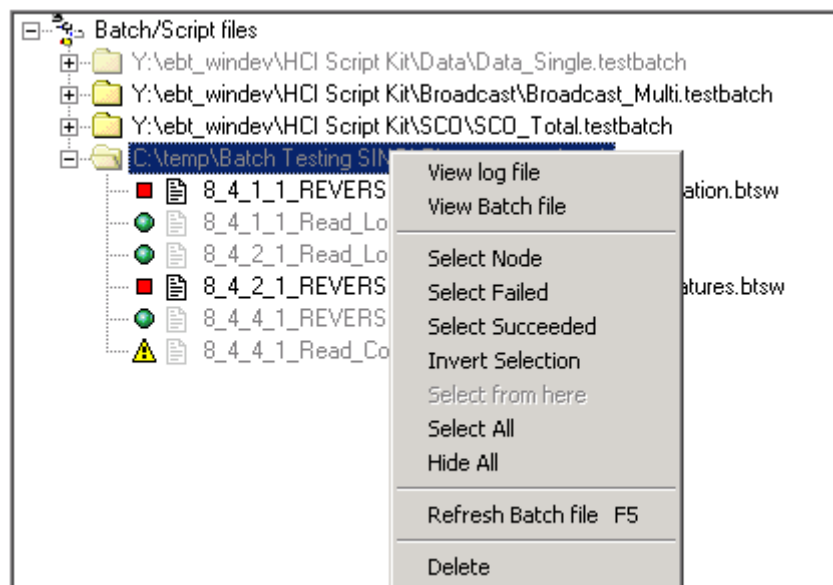


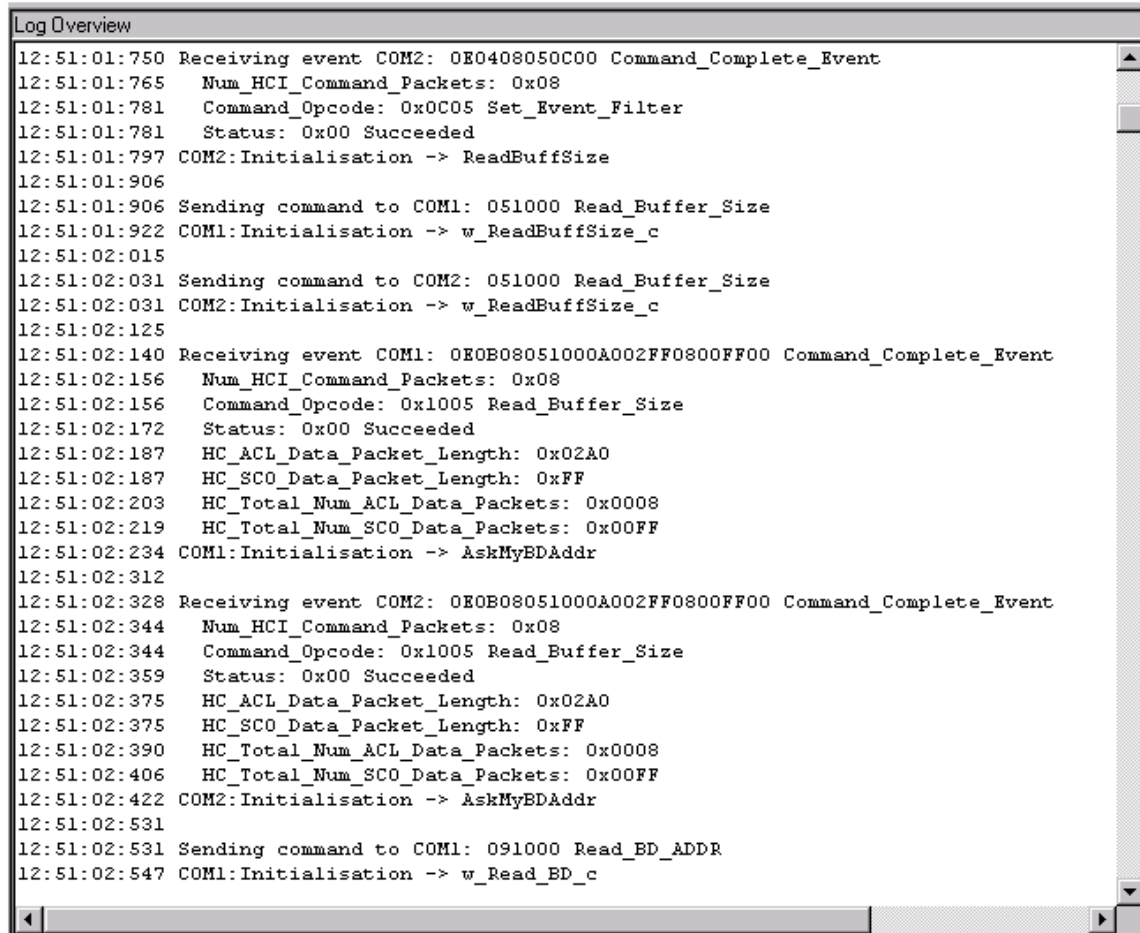
Figure 4-5. BSE Tree view

4.7 Window section

This area is where all windows are spawned and where the user is able to view/edit individual script files. The windows can be arranged according to Windows GUI standards using the windows controls. The windows are resizable and the workspace can be expanded using the

built-in scrollbars. By right clicking on a window the user gets access to the following editing functions by using the popup menu: Cut, Copy, Paste, Delete, Find and Replace.

4.8 Log Overview section



```

Log Overview
12:51:01:750 Receiving event COM2: 0E0408050C00 Command_Complete_Event
12:51:01:765 Num_HCI_Command_Packets: 0x08
12:51:01:781 Command_Opcode: 0x0C05 Set_Event_Filter
12:51:01:781 Status: 0x00 Succeeded
12:51:01:797 COM2:Initialisation -> ReadBuffSize
12:51:01:906
12:51:01:906 Sending command to COM1: 051000 Read_Buffer_Size
12:51:01:922 COM1:Initialisation -> w_ReadBuffSize_c
12:51:02:015
12:51:02:031 Sending command to COM2: 051000 Read_Buffer_Size
12:51:02:031 COM2:Initialisation -> w_ReadBuffSize_c
12:51:02:125
12:51:02:140 Receiving event COM1: 0E0B08051000A002FF0800FF00 Command_Complete_Event
12:51:02:156 Num_HCI_Command_Packets: 0x08
12:51:02:156 Command_Opcode: 0x1005 Read_Buffer_Size
12:51:02:172 Status: 0x00 Succeeded
12:51:02:187 HC_ACL_Data_Packet_Length: 0x02A0
12:51:02:187 HC_SCO_Data_Packet_Length: 0xFF
12:51:02:203 HC_Total_Num_ACL_Data_Packets: 0x0008
12:51:02:219 HC_Total_Num_SCO_Data_Packets: 0x00FF
12:51:02:234 COM1:Initialisation -> AskMyBDAddr
12:51:02:312
12:51:02:328 Receiving event COM2: 0E0B08051000A002FF0800FF00 Command_Complete_Event
12:51:02:344 Num_HCI_Command_Packets: 0x08
12:51:02:344 Command_Opcode: 0x1005 Read_Buffer_Size
12:51:02:359 Status: 0x00 Succeeded
12:51:02:375 HC_ACL_Data_Packet_Length: 0x02A0
12:51:02:375 HC_SCO_Data_Packet_Length: 0xFF
12:51:02:390 HC_Total_Num_ACL_Data_Packets: 0x0008
12:51:02:406 HC_Total_Num_SCO_Data_Packets: 0x00FF
12:51:02:422 COM2:Initialisation -> AskMyBDAddr
12:51:02:531
12:51:02:531 Sending command to COM1: 091000 Read_BD_ADDR
12:51:02:547 COM1:Initialisation -> w_Read_BD_c
  
```

Figure 4-6. Overview window. Every line is prefixed by a time stamp (present time) in the format hour : min : sec : millisecond.

Log overview shows the output of the execution of scripts. Select Transition log to generate a log line of each state transition. The format is:

```
COM2: Init -> w_Reset_c
```

In this logline the name of the device and the name of the state machine (Init) is mentioned. If multiple instances of the same state machine are running on the same channel, it is not possible to distinguish which state machine that has changed state.

Three different logging modes is possible:

- Compact log
- Extensive log
- String log

User chooses which mode to use by either using the menu (i.e. View->Log Overview->Extensive Log) or by using the popup menu. The popup menu is invoked by right clicking on the log overview window.

The Compact Log/Extensive Log/String log modes determine the way incoming events are logged. If **Compact Log** is selected, only a hex dump of the first bytes of incoming events and outgoing commands are logged. This logging has the following format:

```
Sending command to COM1: 050C03020002
Receiving event COM1: 0E0408050C00
```

Which state machine caused the command is not logged. The device sending commands or receiving events and the bytes sent out are logged.

If **Extensive log** is selected, detailed logging will be added:

```
13:58:14:047 Sending command to COM1: 050C03020002 Set_Event_Filter
13:58:14:062 Filter_Type: 0x02
13:58:14:078 Connection_Setup: 0x00
13:58:14:094 Auto_Accept_Flag: 0x02
13:58:14:125 COM1:Initialisation -> w_SetFilter_c
```

If **String log** is selected, the values of the parameters are not printed in hex but in ASCII.

4.9 Script Overview section

In the script overview window, the actual state of the script is shown. Choosing one of the script overview modes selects a particular presentation format. The different script overview modes the user can choose from are:

- Show Bodies. Will view the body of all active state machines and the current state of each, variables and their values will also be displayed.
- Show Test Case. In batch processing, the terminal states of the individual script files will be displayed.
- Show Headers. Shows the current state of each of the state machines.

User chooses which mode to use by either using the menu (i.e. View->Script Overview->Show Bodies) or by using the popup menu. The popup menu is invoked by right clicking on the script overview window.

4.10 Status Bar section

The status bar shows the status of different aspects of the application. BSE makes use of a standard Windows GUI status bar and is illustrated in Figure 4-7.



Figure 4-7. BSE status bar

The status bar consists of five sections described below in left-to-right order:

- Hints. When user places the mouse over menu/toolbar items, a brief explanation of the item is shown in this status bar panel.
- Caret position. When editing a script file, the position of the caret is shown in this status bar panel. The format is Line:Column.
- Looping iteration. This status bar panel shows the increment of the currently running batch/script file.
- Batch/script status. Shows the status of the batch/script file (e.g. RUNNING, ok, error, user interruption etc).
- Current time.

- Communication status. The two remaining panels show the current status of the communication, that is, if BSE is connected to BCP or not.

4.11 Log files

As soon the start button is pressed, a log file is opened, new information printed in the logging window will also be written into the log file. When a script stops (no matter what reason), the log file is closed automatically.

In batch mode also log files are created automatically, one log file per script file is created. One batch log file is also created, this log file contain a summary of the whole batch execution. This is the same information as printed in the script overview window when "Test Case View" is selected. The files are saved in the same directory from where the batch file was loaded, and the name of the batch log will have the same name as the batch file with the extension ".log". The script log files have the same name as the script files with the extension ".log". Note that if you re-run the batch file, the old log files are overwritten!

5 EXECUTION OF PASS/FAIL TESTS

5.1 Introduction

The following example describes how to use BSE in a PASS/FAIL test. The EUT can be either of the examples described in chapter 3. The procedure will also show the connection to other tools: Bluetooth Log Analyzer (BLA) and Bluetooth HCI Toolbox (BHT).

5.2 Test procedure

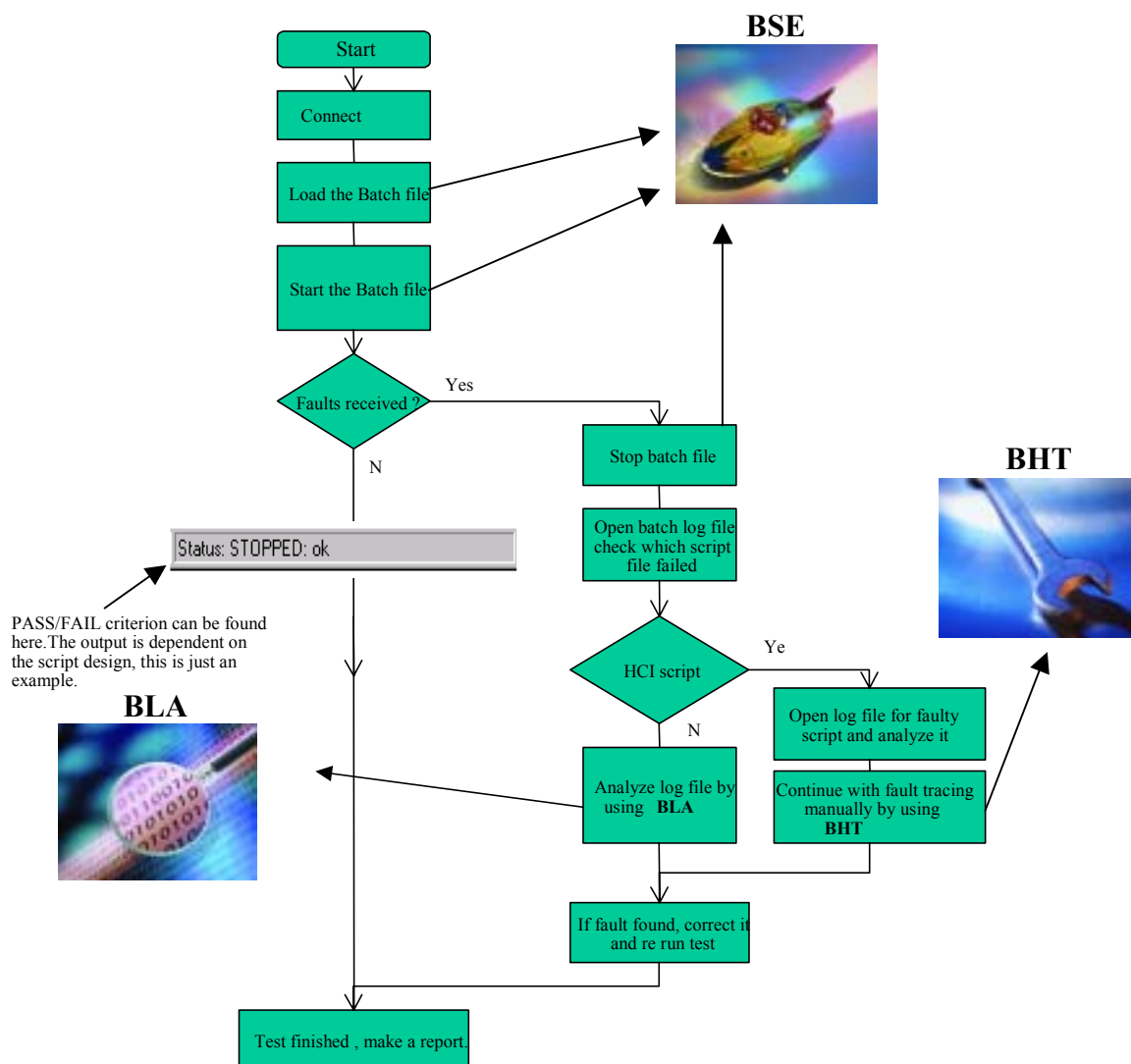


Figure 5-1. Bluetooth verification test procedure

The equipment under test (EUT) can be hardware to which the Ericsson Host Stack has been ported, or a HCI module.

If a batch file fails, the batch overview log tells which script failed. There is also a script log file which can be used for further fault analyze.

If the EUT is the Host Stack, can Bluetooth Log Analyzer [3] be useful. On the other hand — if an HCI-module is the EUT — will the Bluetooth HCI Toolbox [4] be useful.

6 SCRIPT LANGUAGE INTRODUCTION

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

BSE has a script language specially designed for test purposes. Initially it was developed to automate the tests made by Ericsson 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 BSE 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 BSE: The communication between the equipment under test (EUT) and BSE had to be adapted to test conditions — as little runtime input from the test team, and good communication logs should be produced.

6.1 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.

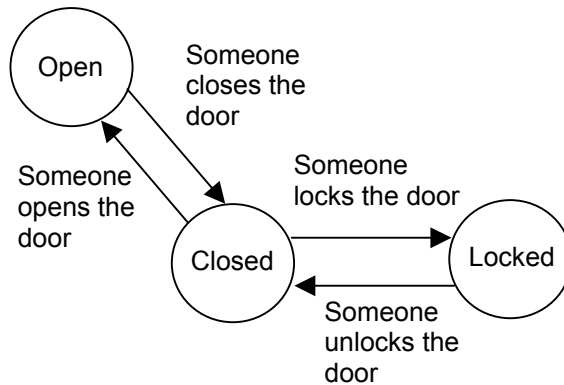


Figure 6-1. State chart representing a door. The current state of the door changes depending on external stimuli: every arrow represents a valid state-transition and has the reason for the transition marked beside.

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

The script language in BSE 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.

6.2 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 from Ericsson. 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. A state machine that tests the HCI-module could look like in

Figure 6-2:

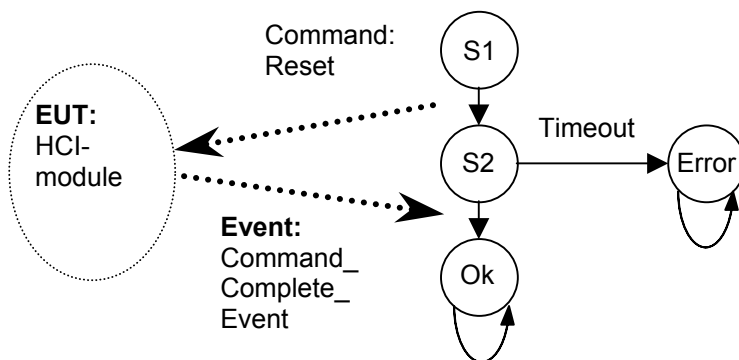


Figure 6-2. A state machine that sends out a command and waits for an event from the HCI-module. The figure is simplified; the connection between the state machine and the EUT is actually indirect, passing through a communication interface.

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

Each of the state transitions (the arrows in Figure 6-2) 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 seconds⁵.

The action `TERMINATE` tells BSE 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.

⁵ BSE uses by default a smallest unit of time of 1.28 seconds.

6.3 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 (*.hci).

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.

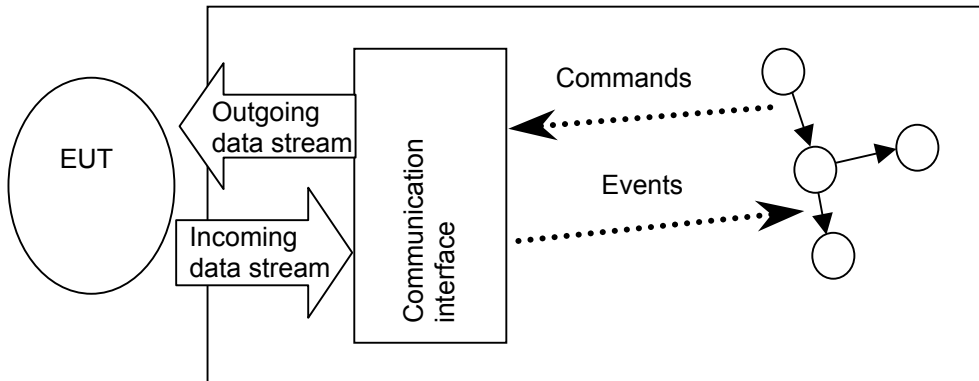


Figure 6-3. The relation between abstract commands/events and I/O is controlled by the Protocol specification file.

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.

6.4 Storage and Synchronization

It is possible to activate several state machines in a single script file. They can, 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 technique⁶ that is used for sharing data.

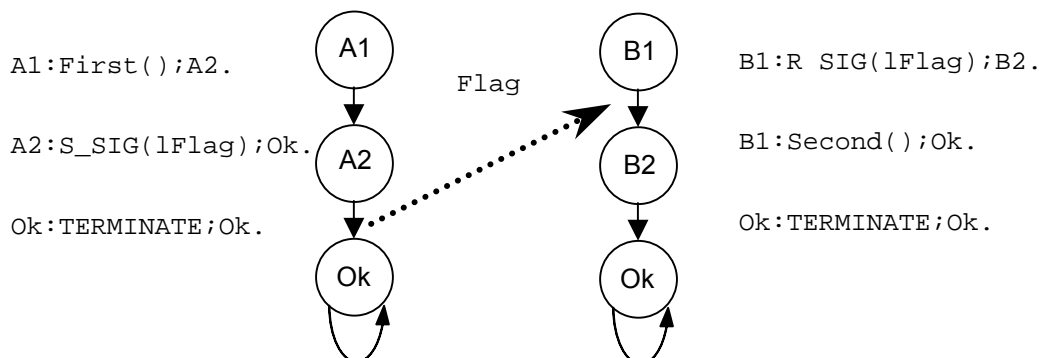


Figure 6-4. Two state machines that are synchronised via the variable flag. The transition B1→B2 is blocked until any other state machine executes a S_SIG-action. lFlag is actually two different local variables but they are assigned, upon invocation of the two state machines, to the same Flag variable in the [testscript]-section.

In BSE 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.

6.5 Use of files

In order to run BSE a few files has to be present:

- The protocol specification (file name extension “.hci”).
- 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 BSE produces to different types of log files (extension “.log”). They reflect the log information that also is shown on screen.

⁶ See chapter 7.3.3.

7 SCRIPT LANGUAGE REFERENCE

7.1 Introduction

This manual describes the BSE script language. There are three types of input files: .btsw, .hci and .testbatch that has different syntax. The first two types are divided in sections, see Table 7-1 below, and will be described in depth in this chapter. The syntax for batch processing is described in chapter 7.5.

Script file .btsw	Protocol Specification file .hci
[data]	[type]
[statemachines]	[functions]
[testscript]	[events]

Table 7-1. 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.

7.1.1 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.

7.1.2 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++).

7.1.3 Keywords

In Table 7-2 the keywords are presented. BSE is case sensitive.

Section	Keywords defined in the section
[data]	
[statemachines]	COMPARE CLEAR HCI_PROTOCOL HCI_READDATA HCI_WRITEDATA RESCUE R_SIG S_SIG TERMINATE TIMER WAIT
[testscript]	VAR
[io]	COM1, COM2, ... COM16 HCI PASCALSTRING PASCAL_HI_LO PASCAL_LO_HI TOKENTERMINATED USB (reserved for future use)
[type]	BITFIELD COMMAND ENUM TIME
[functions]	COMMAND_PACKET COMMAND_SPECIFIC
[events]	COMMAND_PACKET COMMAND_SPECIFIC

Table 7-2. The seven sections and their keywords.

7.1.4 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.

7.1.5 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.

BSE script language lacks the *Expression* symbol. This means that it is not possible to do any calculations in the source code. Comparison of variables and strings is possible and for the latter there is a feature of smart strings (see chapter 7.3.2.3). This feature makes it possible to do advanced pattern matching on string data.

A *qstring* as a command or event argument will be treated as a smart string. This means that the characters ~ and { } will have special meaning. The string " " (a single space) will also be treated in a special way, refer to 7.3.2.3.

7.2 Protocol Specification

BSE is designed in such a way that it contains as little knowledge as possible about protocol properties. BSE knows the following properties of the HCI specification:

- The format of the HCI Command Packet.
- The format of the HCI Event Packet.
- The format of the HCI Data packet.
- The sequence of elements of arrays in case of array parameters.

Those properties are hard coded into BSE but are only activated when the HCI base protocol⁷ is selected. For the moment it is not possible to select any other base protocol.

Not only communication with HCI-modules is done with the HCI base protocol, but also the communication to the Debug component of the Bluetooth HOST stack. In this case the communication consists of text strings. They are just merely wrapped into HCI packages. This protocol specification is found in the file "cit_definitions.hci".

Before the protocol specification file is read, does BSE not know what commands and events are available. Neither does BSE know about the ranges of specific parameters, nor the endianness of the parameters. All this information is stored in this separate file. The advantage of this solution is that several versions of the HCI protocol, or completely different protocols, can be tested with the same BSE 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].

⁷There is a name confusion here since the protocol specification file name extension is ".hci". This extension name has not changed of compatibility reasons. HCI base protocol refers to the above four points. Other base protocols like token terminated, fixed length and pascal string can be chosen. Only HCI base protocol is implemented in BSE v2.5.

7.2.1 Type section

The BSE 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 }
    typename = [ - ] { size , min , max , bitmapdef }
    typename = [ - ] { size , min , max , commanddef }
```

The 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. Strings in the HCI protocol are big endian and thus defined as:

```
t_String = -{ 248 }
```

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`. BSE makes use of the `min` and `max` value to check received parameters and uses these limits when it has to generate a random value.

7.2.1.1 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 BSE 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 7-1. 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 $625\mu\text{s}$. If the variable `timeinterval` gets the value 200 then the state machine will wait $200 \times 625\mu\text{s} = 125\text{ms}$ in the `s12` state.

7.2.1.2 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 BSE does not check this while reading the specification. It is not needed to enter the enumerations in a particular order.

7.2.1.3 Bitfield definition

```
bitmapdef:
    BITFIELD , bitmaplist

bitmaplist:
    bitmapdef
    bitmapdef , bitmaplist

bitmapdef:
    number : bitmapdescription
    ( number , number ) : bitmapdescription

bitmapdescription:
    qstring
```

Two types of bitmaps exist: one-bit and multi-bit bitmaps. The one-bit bitmaps can be used when each bit of parameters of this type has a particular meaning. When several bits together form an enumeration, multi-bit bitmaps have to be used. The first number is the bitmask, indicating the interesting bits; the second number specifies the element in question in the enumeration. An example:

<i>multibit bitmap</i>	<i>meaning (in bits)</i>
(0x0300, 0x0000)	xxxx xx00 xxxx xxxx
(0x0300, 0x0100)	xxxx xx01 xxxx xxxx
(0x0300, 0x0200)	xxxx xx10 xxxx xxxx
(0x0300, 0x0300)	xxxx xx11 xxxx xxxx

The sequence of the enumerations within one bitmask is free. The sequence of the elements themselves is free too, but it is not allowed to mix up enumerations of different bit masks. BSE does not check this, but makes use of this rule when it has to calculate a random, allowed value. When BSE has to generate a random value of this type, it chooses for every bitmask one random element.

7.2.1.4 Command definition

```
commanddef:
    COMMAND
```

This data type is used when a parameter indicates a command number. These parameters are only used in events, especially general-purpose events that contain a small fixed list, and a variable list of return parameters of previous sent commands.

Parameters of this kind are only useful in HCI-protocol.

```
s1 : Inquiry(0x000000, 0x00, 0x00); s2.
s2 : Command_Status_Event(0x12,,Inquiry); s3.
```

In this example, the third argument of the event is of a 2 byte-COMMAND type. Inquiry is a command defined in the [functions] section with an OGF=0x01 and OCF=0x0001 combined to a 2 byte word 0x0401. Thus the transition to s3 will only be done if the third argument is 0x0401.

Also see the keyword COMMAND_SPECIFIC.

7.2.2 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.

In HCI packages two parameters OGF and OCF form one 2-byte word. Those parameters have to be specified.

```
functiondef:
    functionname = { OGF , OCF , input - output }

input:
    (empty)
    parameterdeflist

output:
    (empty)
    parameterdeflist

parameterdeflist:
    parameterdef
    parameterdef , parameterdeflist
    parameterdef UNION { unionlist } , parameterdeflist
    parameterdef UNION { unionlist uniondefault } , parameterdeflist

unionlist:
    union
    union , unionlist

union:
    ( identifier = number ) parameterdeflist

uniondefault:
    ELSE parameterdeflist

parameterdef:
    parametername : typename
    parametername : typename [ ]
    parametername * : typename
```

7.2.2.1 Union definition

Unions make life hard for BSE script developers, but are needed to completely describe the HCI specification. There is only one function that makes use of this facility: the Set Event Filter.

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.

7.2.2.2 Array parameters

The second and third definition of `parameterdef` is used for defining array parameters. Knowledge about HCI data structure is hard coded into BSE itself: BSE knows the sequence of the array elements in the HCI packet. The parameter indicating the number of elements in the arrays must be marked with an asterisk.

7.2.3 Events section

Events are defined similarly to commands, but the structure is simpler. The following definitions are allowed:

```
eventdef:
    eventname = { eventnumber , parameterdeflist }
```

The eventnumber is a one-byte number that has to be specified.

Events may return a parameter containing a function number. If the type of this parameter is defined as a `COMMAND` type, BSE will recognize this, and the following parameter is allowed to be of type `COMMAND_SPECIFIC`. BSE will receive the `outputsection` of the command indicated by the previous parameter. This follows the description of HCI found in [1]

7.3 Script language

A script consists of 3 sections stored in a file, see Table 7-1. They are presented below in the order they can appear in a script file⁸. 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 to be used to send HCI data packets to the EUT. The second defines state machines and the last “links” state machines to the communication channels.

7.3.1 Data section

This section is optional. It defines global⁹ string variables to be used with the two state machine keywords `HCI_READDATA` and `HCI_WRITEDATA`. They can be initialized.

The scope for the [data] variables is unlimited. They can be used in a state machine without having it passed as an argument to the state machine. This is different from variables defined in the [testscript] section.

The syntax is very simple, so an example presents it clearly. Below is an extract of a script file with a data section defining three data buffers.

```
[data]
databuffer1 = "Test string \n\r"
databuffer2 = "Say hello"
emptydatabuffer
```

The two first buffers are initialized and can be used to send out strings to the EUT. This is not possible with the data buffer `emptydatabuffer`. It can on the other hand be used to match *any* incoming string from the EUT.

⁸ Or *files* since the preparer can include other files into one script. More information in chapter 7.3.4

⁹ They can be used in all state machines in the script.

7.3.2 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 single¹⁰ 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 BSE. 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 7.3.3.

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 7.4.1.

7.3.2.1 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 .
```

¹⁰ 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.


```
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 identifier¹¹ 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.

If a command parameter is *empty* this means that BSE should insert a random value fulfilling the type definition in the [data] section. If an event parameter is empty this means 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 7.3.2.5.

7.3.2.1.1 Variable number of arguments

Note! This section describes functionality that is not used when connecting BSE to a HOST stack. In the case that the EUT is a HCI-module, variable length of argument lists is of importance.

Some commands and events have a variable number of parameters¹²: The parameter list has a fixed part and a variable part. The variable part always depends on the value of a specific argument in the fixed part. That very parameter **must** be a *number* or a *functionname*. It is **not** allowed to use a *varname* or *empty* here, because only a *number* or *functionname* makes it possible to check the number of parameters while parsing the state machine. A *number* is typically used in case the variable part is defined as a union. A *functionname* is used when the event contains values for "return parameters" of a command. In the latter case, the type of the parameter determining the variable part of the parameter list must be a *COMMAND* type.

7.3.2.1.2 Variable length of parameters

All the types defined in the [type] section have fixed length. Often a string type is defined there in a manner similar to one of these:

```
[type]
t_B248      = -{ 248 }
t_String    = -{ 240 }
```

¹¹ *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.

¹² See chapter 7.2.2.1, 7.2.2.2 and 7.2.3.

```
[functions]
CIT_Command = { 0x00, 0x0080,
  CommandString : t_String
-
}
```

It is still possible to make a call like this,

```
s10: CIT_Command("SCM.FNC:HCI_CmdReset()") ; s11.
```

even if the length of the quoted string is not 240. The *last* parameter in a packet is allowed to be shorter in size than the specified in the command/event definition; no zero-padding is made.

7.3.2.2 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.

7.3.2.2.1 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.

7.3.2.2.2 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, BSE 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 BSE 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 generates a warning.

7.3.2.3 Smart strings

Note! This section describes functionality that is not used when connecting BSE to a HCI-module. In the case that the EUT is the Ericsson HOST Stack smart strings is of great importance.

Smart strings are added to BSE to be able to conduct tests on equipment that do not support an HCI interface. They do not really fit in the structure of BSE, but were added to allow communication via text strings to the EUT.

The HOST Stack sends out strings of information, which look like:

```
YYY.FNC:L2CA_ReqVersion(0x02)
YYY.L2CA_VERSION_CNF(0x02,L2CA_POS_RESULT,CAX103422 X1C )
```

BSE has poor facilities to use the contents of these strings, because BSE was not designed for this matter. However, to be able to use BSE in these environments, so called "smart strings" are implemented.

7.3.2.3.1 Wildcards

Quoted strings can contain one or more wildcard characters '~'. When a quoted string contains a '~' any character can be used instead. This is useful if the EUT uses sequence-numbers or time-stamps in the response-string.

An isolated space in a string (" ") will match any string in an incoming event. A modified example from Bluetooth Application Script Kit:

```
s10 : CIT_Response ( " " ) ; s10.
```

In this state any response from the Host Stack [2] will invoke the transition back to s10. This construction is useful when information messages are received that should be logged into the log file.

7.3.2.3.2 Substrings in events

Smart strings in an event can contain substrings like {n,varname}. The first parameter n indicates the number of characters that will be extracted from the received string; varname indicates the variable in which those n characters will be stored.

Other variants exist:

{n,=varname}

The received characters must match the value of varname. If the variable is uninitialized, a normal assignment will happen.

{n,<varname}

The received value must be smaller than the value of varname. If the variable is uninitialized, the test will fail. No assignment will take place.

{n,>varname}

The received value must be bigger than the value of varname. If the variable is uninitialized, the match will fail. No assignment will take place.

{n,!varname}

The received value must be unequal to the value of varname. If the variable is uninitialized, the match will fail. No assignment will take place.

7.3.2.3.3 Substrings in commands

Strings in a command can contain substrings like:

{varname}

If varname is uninitialized, no characters will be inserted in the output string. Otherwise, the value of the variable will be inserted.

7.3.2.3.4 Example

This is an example of a state machine showing the use of smart strings:

```
HCI_Simulator(ready) = {
  VAR SeqNr.
  s0 : CIT_Response("L2C.HCI_ReqStart({4,SeqNr})") ; s1.

  s1 : CIT_Command("HCI.MSG:L2C[HCI_START_CNF({SeqNr},HCI_NO_ERROR)]") ; s2.

  s2 : CIT_Response("L2C.HCI_CmdRegister(0x~~)") ; s3.
  s2 : TIMER(0x01) ; timeout.

  s3 : CIT_Response("L2C.HCI_ReqDataInfo(0x8888)") ; s4.
  s3 : TIMER(0x01) ; timeout.

  s4 : CIT_Command("HCI.MSG:L2C[HCI_DATA_INFO_CNF(0,0,04,10,10,10,02,01)]") ; s5.
  s5 : S_SIG(ready) ; s0.

  timeout : TERMINATE ; timeout.
}
```

In state `s0` the state machine is waiting for a response to be received. This response contains a `SeqNr` of 4 bytes that must be used in following confirm message.

7.3.2.4 Advanced flow control

In the previous release 1.1 of BSE two new concepts were introduced, and they apply to the R3A release as well. The first is *atomic* state transitions, and the second is an interrupt procedure using the new keyword `RESCUE`. Additionally, in version R3A of BSE, a script language extension to allow multiple event transitions is introduced. These concepts give enhanced control on state transitions and are described in the following subchapters.

7.3.2.4.1 Atomic state transitions

In a situation where several actions *has* to be performed *without* any other actions handled in between, the atomic transition can be used. Such a transition not only blocks other state machines, but also the processing of incoming events.

The arrow token marks the atomic transition, as seen in the code below:

```
s10 : Event(,,) -> s11.
s11 : Command() ; s20.
```

The intended use for this transition is when a number of events, in *unspecified* order, have to be notified before the flow continues. If the script did not use this feature, quite tiresome constructions would be necessary.

The following example illustrates how a script can be written to enforce that both `rsp1` and `rsp2` has been received before the transition to state `continue`:

```
s0 : inquiry ; s1.

s1 : rsp1      ; s2.
s1 : rsp2      ; s3.

s2 : rsp2      ; continue.
s3 : rsp1      ; continue.

Continue : inq_complete ; Finished.
```

This construction becomes very complex if the number of required responses is higher. A better solution is to use signals as flags and later verify that all signals are set. An example with four required responses is given below. Note that `inq_complete` marks the end an inquiry here:

```
s0 : inquiry ; s1.
```

```
s1 : rsp1      -> s1a.
s1 : rsp2      -> s1b.
s1 : rsp3      -> s1c.
s1 : rsp4      -> s1d.
s1 : inq_complete ; s2a.

s1a : S_SIG(x1) ; s1.
s1b : S_SIG(x2) ; s1.
s1c : S_SIG(x3) ; s1.
s1d : S_SIG(x4) ; s1.

s2a : R_SIG(x1) ; s2b.
s2a : TIMER(0) ; error.
s2b : R_SIG(x2) ; s2c.
s2b : TIMER(0) ; error.
s2c : R_SIG(x3) ; s2d.
s2c : TIMER(0) ; error.
s2d : R_SIG(x4) ; Finished.
s2d : TIMER(0) ; error.
```

Upon reception of `inq_complete` event the four signals are verified in sequence and a transition to `Finished` is performed, if all signals were set.

Note that “arrow transitions” is used in state `s1`. This means that upon entry to states `s1a`, `s1b`, `s1c` and `s1d`, the action is processed immediately, and incoming events remain unprocessed in the input buffer. Once BSE is returned to state `s1`, incoming events are processed again. The use of atomic transitions is needed here, since incoming events must not be handled in `s1a`, `b`, `c`, and `d`. If `rsp2` would arrive in state `s2a`, it would be discarded since there is not a matching transition in that state.

7.3.2.4.2 Interrupt states

A state machine can have one interrupt state. If a `RESCUE` action is performed, an immediate transition to this interrupt state is made. The rescue action and the interrupt states can be located in different state machines.

The interrupt state is specified after the local variables as in:

```
StatemachineA() =
{
  VAR a,b,c.
  RESCUE s12.

  s0 : ...
  s1 : ...
  s12 : ...
}
```

In another state machine a line similar to this can be found:

```
problem_state: RESCUE : s45.
```

Whenever such a `RESCUE` command is performed, all state machines – that have rescue states specified – will transfer to those rescue states. In this case will `StatemachineA` transfer to state `s12`.

7.3.2.4.3 Multiple event transitions

The multiple event transition is extremely useful in the following situation: Imagine a test environment with four discoverable Bluetooth devices with known BD addresses. One of the devices is running this script:

```
s00 : Inquiry( , , ) ; s01.
s01 : Command_Status_Event(0x00, , Inquiry) ; s02.
```

```
// now we want to receive a response from the 3 other devices
// The variables bd1, bd2 and bd3 have been initialized already
// with the correct bd addresses. We want to be sure we've
// received these 3 responses, but we don't know the order in
// which they receive. Furthermore, we can receive a response
// from unknown devices (maybe in the room above us, on the next
// floor). The first response can origin from #1, #2 or #3:

s02 : Inquiry_Result_Event(1, bd1, , , , , ) ; s03.
s02 : Inquiry_Result_Event(1, bd2, , , , , ) ; s04.
s02 : Inquiry_Result_Event(1, bd3, , , , , ) ; s05.
s02 : Inquiry_Result_Event(1, , , , , , ) ; s02. // from room above
us.

// In state s03, we received #1. Wait for #2 and #3.
s03 : Inquiry_Result_Event(1, bd2, , , , , ) ; s03b.
s03 : Inquiry_Result_Event(1, bd3, , , , , ) ; s03c.
s03 : Inquiry_Result_Event(1, , , , , , ) ; s03. // from room above
us.

s03b: Inquiry_Result_Event(1, bd3, , , , , ) ; s06.
s03b: Inquiry_Result_Event(1, , , , , , ) ; s03b. // from room above
us.

s03c: Inquiry_Result_Event(1, bd2, , , , , ) ; s06.
s03c: Inquiry_Result_Event(1, , , , , , ) ; s03c. // from room above
us.

// In state s04, we received #2. Wait for #1 and #3.
s04 : Inquiry_Result_Event(1, bd1, , , , , ) ; s04b.
s04 : Inquiry_Result_Event(1, bd3, , , , , ) ; s04c.
s04 : Inquiry_Result_Event(1, , , , , , ) ; s04. // from room above
us.

s04b: Inquiry_Result_Event(1, bd3, , , , , ) ; s06.
s04b: Inquiry_Result_Event(1, , , , , , ) ; s04b. // from room above
us.

s04c: Inquiry_Result_Event(1, bd1, , , , , ) ; s06.
s04c: Inquiry_Result_Event(1, , , , , , ) ; s04c. // from room above
us.

// In state s05, we received #3. Wait for #2 and #1.
s05 : Inquiry_Result_Event(1, bd2, , , , , ) ; s05b.
s05 : Inquiry_Result_Event(1, bd1, , , , , ) ; s05c.
s05 : Inquiry_Result_Event(1, , , , , , ) ; s05. // from room above
us.

s05b: Inquiry_Result_Event(1, bd1, , , , , ) ; s06.
s05b: Inquiry_Result_Event(1, , , , , , ) ; s05b. // from room above
us.

s05c: Inquiry_Result_Event(1, bd2, , , , , ) ; s06.
s05c: Inquiry_Result_Event(1, , , , , , ) ; s05c. // from room above
us.

s06 : Inquiry_Result_Event(1, , , , , , ) ; s06. // from room above
us.
s06 : Inquiry_Complete_Event(0x00, ) ; s07.
```

Reading this script makes one feel that things are unnecessarily complex. Using the prior versions of BSE, this can not be simplified. In this version of BSE, an extension of the script language is introduced to allow multiple event transitions. By using this language feature, it is possible to create simplified scripts. The following script has **exactly** the same behavior as the script above:

```
s00 : Inquiry( , , ) ; s01.
s01 : Command_Status_Event(0x00, , Inquiry) ; s02.

// now we want to receive a response from the 3 other devices
// The variables bd1, bd2 and bd3 have been initialized already
// with the correct bd addresses. We want to be sure we've
// received these 3 responses, but we don't know the order in
// which they receive. Furthermore, we can receive a response
// from unknown devices (maybe in the room above us, on the next
// floor). The first response can origin from #1, #2 or #3:

s02 : Inquiry_Result_Event(1, bd1, , , , , )
      Inquiry_Result_Event(1, bd2, , , , , )
      Inquiry_Result_Event(1, bd3, , , , , ) ; s06.
s02 : Inquiry_Result_Event(1, , , , , , ) ; s02. // from room above
us.

s06 : Inquiry_Result_Event(1, , , , , , ) ; s06. // from room above
us.

s06 : Inquiry_Complete_Event(0x00, ) ; s07.
```

7.3.2.5 Internal actions / Keywords

One group of internal actions is CLEAR, COMPARE, R_SIG, S_SIG, TIMER, 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 behaviour of the state machines by internal means. Others like HCI_READDATA and HCI_WRITEDATA, exist for sending/receiving HCI data packets.

Available internal actions are described below.

7.3.2.5.1 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 7-2. When handle is uninitialized, 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.

7.3.2.5.2 COMPARE

This action has two parameters: both must be data buffers defined in the [data] section.

Unless the data buffers are equal the state transition is blocked. Leaving one of the variables empty will force a transfer to the destination state.

Since the data buffers have global scope any state machine can do a comparison.

This command is normally used after a HCI_READDATA. This action can be avoided by using an initialized data buffer as argument for HCI_READDATA.

Example:

```
compare : COMPARE(SendBuffer,ReceiveBuffer) ;ok.  
compare : COMPARE(SendBuffer,) ;err.
```

Example 7-3.

7.3.2.5.3 HCI_PROTOCOL

The HCI_PROTOCOL action has no parameters.

This action can be used to switch the communication base protocol to HCI protocol. This results in BSE sending HCI packets and expects HCI packets from the EUT.

7.3.2.5.4 HCI_READDATA

This action has four parameters:

```
HCI_READDATA(handle, pbFlag, bcFlag, databuffer)
```

This transition works similar to all other events: variables that are of no interest can be omitted. Until a HCI ACL data packet arrives that matches the requirements the transition is blocked. If `databuffer` is empty it will be filled with the packet data.

The last parameter is compulsory and must be a variable defined in the [data] section.

7.3.2.5.5 HCI_WRITEDATA

This action has four parameters:

```
HCI_WRITEDATA(handle, pbFlag, bcFlag, databuffer)
```

This will result in one HCI ACL data packet being sent to the EUT. This data packet is filled, according to [1], with `handle`, `pbFlag`, `bcFlag` and the `databuffer`. The last argument must be a variable defined in the [data] section.

7.3.2.5.6 RESCUE

The RESCUE action has no parameters.

When a rescue action is performed, all state machines that have a rescue state will go to there.

This can, for instance, be used in case a certain command procedure fails, and the EUT has to recover from this failure before next script file is run.

7.3.2.5.7 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.

7.3.2.5.8 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.

7.3.2.5.9 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.

7.3.2.5.10 TIMER

One parameter is allowed. It must be a variable or an integer constant. The time scale is 1.28 seconds unless the variable is of a TIME type and another *timescale* is specified in the *typedef*.

On slow computers it can be necessary to make BSE run faster by pressing "Disable logging", thus avoiding the screen 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.

The transition is blocked until the timer expires.

7.3.2.5.11 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.

7.3.3 Testscript section

This section instantiate state machines. Arguments can be passed to each of the instances and they are always passed by *reference*¹³. In fact, all arguments to the state machines should be a variable —not a constant value.

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 device as defined in the device mapping dialog, see chapter 4.4.3.6.

¹³ 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.

Example

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

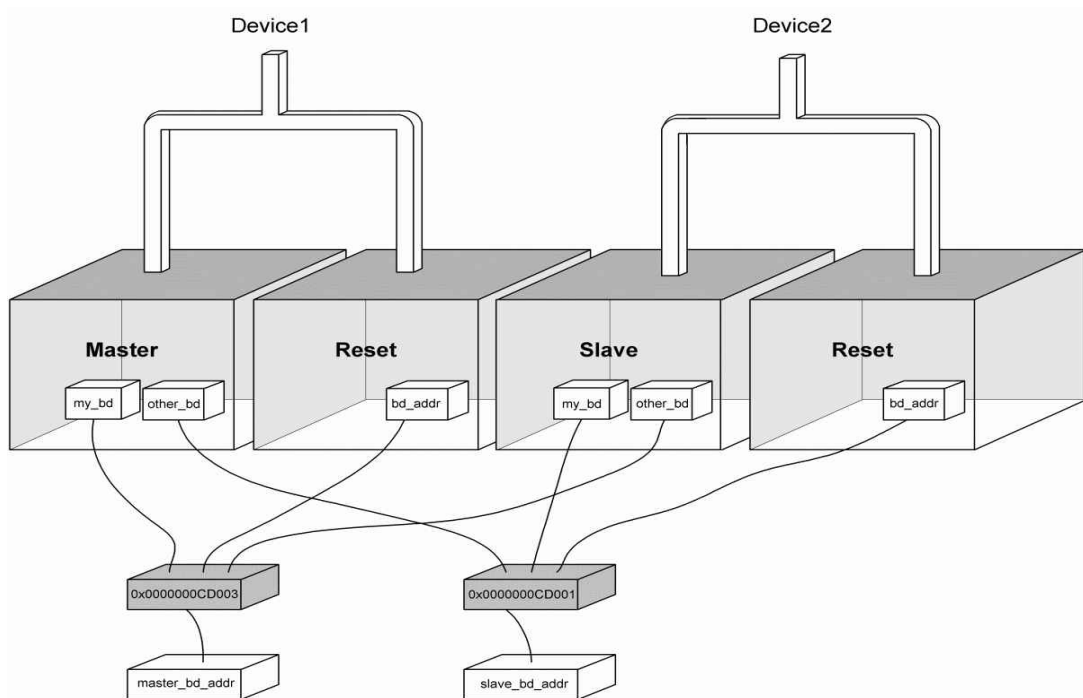
```
Master(my_bd,other_bd) =
{
  s0 : WAIT(my_bd) ; s1.
  s1 : WAIT(other_bd); s2.
  s2 : Inquiry( ...
}
```

```
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 7-4. Example of an HCI-module test. Four state machines are communicating with two devices. All state machines share the addresses of the two HCI-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.

7.3.4 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.btsw` file into this script.

```
#include "init.btsw"
```

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.

7.4 Runtime

7.4.1 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 BSE decides to execute a command, one command out of the available commands in that state is chosen on a random base. When BSE decides to become pending, and there is a timer, 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” BSE steps in a loop until a state machine performs the `TERMINATE` state transition or “Stop” is pressed.

7.4.2 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.

7.4.3 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.

7.5 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:

hctest.testbatch	(batch file)
hci_1_0.hci	(protocol specification)
first_test.btsw	(script file)
second_test.btsw	(script file)
MyDir\third_test.btsw	(script file)

```
hci_1_0.hci
first_test.btsw
second_test.btsw
MyDir\third_test.btsw
end.
```

```
[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 7-2 first_test.btsw

```
[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 7-3 second_test.btsw

The contents of the `hci_1_0.hci` file are not printed because the file is too long. After starting the `hci_test.testbatch` file, BSE loads the HCI 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 `.hci`-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
hci_test.log
```

The file `hci_test.log` contains a summary of the test:

```
Bluetooth Script Engine
=====
Bluetooth Script Engine R3A
Copyright (c) 2003
Ericsson Technology Licensing
=====
Build: Dec 21 2001

Batch started at: 11:14:07

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

Figure 7-4. generated file : hci 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.