STTool User's and Developer's Manual

Version 2.0

Sudhanshu Shukla, Venkata Rao Pedapati, Rajat Moona,

Department of CSE, IIT Kanpur

Version 2.0 April 2010

Table of Contents

Scope and purpose

STTool, or Smartcard Test Tool, is a piece of software used to test the behavior of test scripts on cards that use SCOSTA/ SCOSTA-CL/ SCOSTA-PKI Smart Card Operating System. STTool uses test scripts which are executed on the smart card with the help of command library using PC/SC interface. It facilitates easy and quick way of performing a sequence of complex related operations on smart cards supporting SCOSTA/SCOSTA-CL/SCOSTA-PKI OS. This document explains the architectural overview of STTool and elucidates how to use it to actually test the smart cards. This document shall also help future developers of STTool to understand the implementation and architecture of STTool and will enable them to add new components or features. This document shall also help users of STTool, write test scripts to test a smart card.

References

1. SCOSTA, SCOSTA-CL & SCOSTA-PKI Specifications (http://scosta.gov.in/)
2. ISO7816 documentations (http://www.iso.org)
3. Command Library
4. PCSC library reference - MSDN and Muscle PCSC lite manual

Introduction

STTool is a tool used for executing test scripts on an ISO7816 smart card. The primary purpose of this tool is to perform tests on a SCOSTA or SCOSTA-CL or SCOSTA-PKI compliant Smart card Operating System. This tool supports the transport layer in T=0, T=1 and T=CL protocol. The APDUs can be specified in a well understood scripting language which can be interspersed with the C++ program fragments.

The STTool also supports secure messaging. In this mode, the APDUs can be specified in usual manner but are transported to the smart card using the secure messaging. The script writer has absolute control on the format of the SM on the command and can use it to verify the format on the received response APDUs.

STTool also supports error condition checking. It is possible for the script writer to intentionally provide wrong APDUs or wrong SM specifications and check for the correct behavior of the card (which in this case must be a return SW1 and SW2 to indicate errors).

STTool can also check for the expected values and can report mismatch in case the expected values are not returned by the card.

STTool also handles automatically the warning conditions returned by the card and can handle getting data using GET RESPONSE command if needed.

Features of STTool

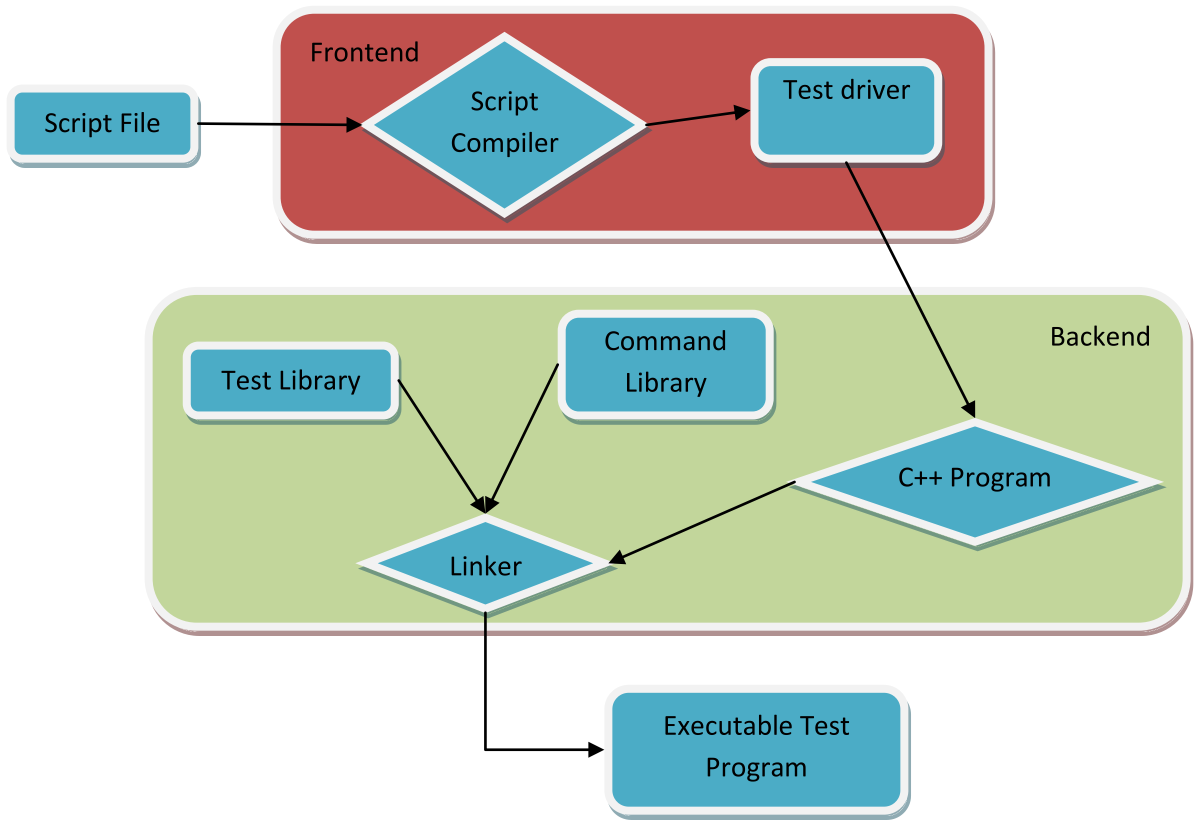
The primary purpose of STTool is to execute test scripts that can be written in an easy-to-understand language on an ISO7816 smart card. The script writer shall write ISO commands in a language that is explained in later sections of this document. When built and executed, STTool shall execute these commands on a real smart card one by one. The following list provides a non-exhaustive list of features it provides for user’s convenience.

- Status checking – The expected status of the command can be mentioned along with the command specification. This expected status shall be compared against actual status bytes received from the card and a message shall be printed in case of a mismatch. This helps users to give commands with wrong parameters or wrong commands and make sure that the card is returning correct error codes. The status can be checked against the equality, non-equality and may include wild cards.
- Return value checking – The expected return values of the command can also be mentioned along with the command. This expected value shall be compared against the actual output and an error message shall be printed in case of a mismatch. The return value can be checked against the equality, non-equality and may include wide cards. The return value may be specified as a hexadecimal character string with a leading “\x” for each byte or a character string that includes the wild card characters, hexadecimal digits and white spaces.
- STTool automatically resolves warning conditions. For example, if the status of a command execution is 0x61XX, STTool automatically executes **GET RESPONSE command** to retrieve the remaining data.
- STTool supports Secure Messaging – An SM environment can be set up by issuing a special command in the script. The SM environment so defined shall apply to all commands that follow till another SM environment is defined. All commands that follow a SM command are wrapped up in SM data objects and sent to the card after translation (as mentioned in the SM command). This translation does not require any extra specifications from the script writer. On the response side, STTool processes the SM response obtained from the transport layer and recovers the original response and the original status. Both of these operations are performed as per the SM specifications.
- Script writes can inter-mix script language with C++ language constructs. This allows users to be able to perform the necessary calculations to obtain expected values etc. Users can even use the variables defined in these C++ constructs as part of the script. STTool automatically resolves them.
- The same code works for both Linux and Windows platforms. On the Linux platform it uses MUSCLE PC/SC Lite, while on the Windows platforms it uses PC/SC API.
- STTool provides the functionality of various cryptographic algorithms that are exported from command library. Command library already contains software implementations of various DES based algorithms, SHA1 & RSA algorithms (PKCS1 version2.1). Thus these algorithms can be used while writing scripts. Adding support for newer cryptographic algorithms is trivial.

- Compiler for STTool is written using well established Lex and Yacc infrastructure. This facilitates addition of new language constructs in an easy manner.

Architectural Overview

The design of STTool can be depicted by the following figure.



STTool comprises of two distinct major components.

- Frontend or the script compiler
- Backend or the Test library

The test suite developer shall write a test script in a language dictated by the script compiler. The script file consists of a sequence of commands separated by a semicolon. The frontend or the Script compiler parses the script file, and compiles it into a C++ program (test driver). All commands in the test script are converted into corresponding function calls of command library functions. Parameters of the command are converted as arguments to these functions.

The backend contains the implementations of routines in a library which are called by the frontend generated C++program. The .cpp file generated by the script compiler, the backend library and command library are compiled and linked together to produce the final test executable. This executable test program on execution, executes each of the commands in the script one after another on the real smart card.

Syntax of Script Files

In this section, we shall see in detail, syntax of the scripting language to be used in script files. All scripts follow the following basic rules.

- White Spaces do not matter.
- Each command is terminated by ';' character and can be spawned over multiple lines of the file. Similarly a line in the file can contain one or more commands, each terminated by a ';'.
- A general syntax of the command is the following.
  CommandAPDU expected_status;
  Here "CommandAPDU" is an ISO command. Upon its execution, the received status is checked for the "expected_status". "expected_status" is optional and when not specified, it defaults to checking for no errors (i.e. **OK**).
- "CommandAPDU" is the command name followed by parameters that the command has.
- "expected_status" is the string of kind "SW?=<value>". The "expected_status" is optional. When not specified it is equivalent to specifying the string as "SW?=OK". The expected status enables STTool to compare the obtained status bytes with the expected status bytes and report an error message if there is a mismatch. The following are the valid ways in which the expected status can be provided.
  - ***SW ?= ERROR***. This causes the STTool to compare the returned status to be an error. The error values is indicated by SW1 being of type 0x6X where X is a number other than 0 and 1.
  - ***SW ?= OK***. This causes the STTool to compare the returned status to be non-error type. If the returned status is of type 0x61xx, the STTool automatically issues a GET RESPONSE command to get xx number of bytes. The values of the status must be 0x9000 (after the GET RESPONSE or after the command as the case may be).
  - ***SW ?= ANY.*** This causes the STTool to ignore the status returned by the card upon execution of the command. The status is only reported and no matching is done for the error reporting purposes.
  - ***SW ?=*** "<STRING>". An expected value of the SW can be specified using a string. The string must be a hexadecimal number prefixed by "0x" and must contain four hexadecimal digits or a '*' to indicate that the corresponding digit need not match. Some examples of this pattern are the following. SW ?= "0x65**" (to indicate that SW1 must be 0x65 while SW2 can be any value). SW?="0x6CF*" (to indicate that SW1 must be 0x6C while the SW2 can be any number between 0xF0 to 0xFF).
- There are some special commands like ***SM*** and ***RESET.*** SM Command does not execute anything on the card but it defines an SM environment with a tag list. All the commands that follow this command until the end of the file or another SM command are sent using secure messaging as

per the structure of the tags given in this SM command. Syntax of the SM command is described later. RESET command resets the card.

"CommandAPDU" stands for a single ISO command and the format depends on the command itself. The following list describes the syntax of each command.

Notations used in the document

The following notations are used while describing the syntax of the test scripts.

- ***Bold Italic Text*** provides the syntax of the command language.
- ***UPPERCASE*** words in bold and italic indicate keywords. The keywords are case-sensitive in the script file and can only be typed in all capitals.
- Constructs enclosed inside ***[square brackets]*** are optional.
- Constructs enclosed inside ***{curly brackets}*** are mutually exclusive – that is only one of them can exist.
- Constructs enclosed between ***<angular brackets>*** are the values – typically integers and strings.
- Constructs separated by a ‖ represent any one of the list.

File Based Commands

Create File

```
CREATEFILE FILEID=<fileid> {
        DF [DFNAME=<dfname>] [SE=<se>] [SEFILE=<sefile>] ‖
         {INTERNAL ‖ WORKING}
                 {TRANSPARENT FILESIZE=<filesize> ‖
                         {FIXEDLENGTH ‖ VARLENGTH ‖ CYCLIC} MNR=<mnr> MRL=<mrl> [SIMPLETLV]
                }
                [SFI=<sfid>]
                 [DATACODING={WRITE_OR ‖ WRITE_AND ‖ WRITE_ONCE}]
}
[LCSI=<lcsi>][COMPACT_ATTR=<compact_access_rule>][EXPANDED_ATTR=<expanded_access_rule>]
```

The command shall create a new DF or EF inside the card with the FCP dictated by the options given in this command. If it is a DF, optional DFNAME, SE and SEFILE can be specified. If it is an EF, several options can be specified – whether it is internal or working, whether it is transparent or record oriented file. If transparent, file size needs to be given. For record oriented files, one of options FIXEDLENGTH, VARLENGTH and CYCLIC, has to be chosen. <mnr> is the Maximum number of records in the record file, while <mrl> is the Maximum record length. Optionally, Short File identifier can be specified in <sfid> and data coding byte can be specified using the construct DATACODING = <value>.

The LCSI for the file to be created can be specified using the construct LCSI=<lcsi>. Compact Access Rule and Expanded Access Rule can be specified as string values using the constructs COMPACT_ATTR=<compact_access_rule> and EXPANDED_ATTR=<expanded_access_rule> respectively.

Select File

{SF || SELECTFILE} {DFNAME || CDF || CEF || PDF || MF || MFPATH || CDFPATH}
[= {<string> || <id>}] {FIRST || LAST || NEXT || PREVIOUS || TOLAST || FROMLAST}
[{FCP || FCI || FMD} [<numbytes>]]
[EXPVAL{?= || !=} {<hexstring> || <string>}]

This command selects a file inside the card. Selection can be done by various methods – by DFNAME of the DF, or id of child DF, id of the child EF, selecting Parent DF or MF directly, by specifying path from MF, by specifying path from the current DF. The next option enables us to choose whether to select first or last or next or previous file when multiple entries are returned by the above selection. The next option indicates if either the FCP, FCI or FMD are required to be retuned in the response and if so the number of bytes expecting is given in <numbytes>. Optionally an expected value may be present.

If the <numbytes> is provided as a non - Zero(0) value then Ne is set as <numbytes>. Otherwise if EXPVAL is provided then Ne is set as the length of the EXPVAL. Finally if <numbytes> is not provided or provided as Zero(0) and EXPVAL is not provided then Ne is set as EMPTY or Zero(0).

Delete File

{DF || DELETEFILE} {DFNAME || CDF || CEF || PDF || MF || MFPATH || CDFPATH} [= {<string> || <id>}] {FIRST || LAST || NEXT || PREVIOUS || TOLAST || FROMLAST}

This command deletes a file whose address is indicated by one of the options given above and explained in previous commands.

Read Binary

{RB || READBINARY} [SFI =<sfid>] <offset> <numbytes> [EXPVAL{?= || !=} {<hexstring> || <string>}]

<sfid> is the value of SFID of the file from which we wish to read. If it is not specified, the data is read from the currently selected file. If <sfid> is provided, the corresponding file becomes the currently selected file after the execution of the command.

<offset> is the offset in the file from where the data is read.

<numbytes> is the number of bytes to read.

Optionally, one can also specify Expected value of the read data by using the **EXPVAL** construct. The expected value can be specified as a Hexadecimal string or a normal string – which can even be a c variable. A hexadecimal string is a string of hexadecimal bytes separated by spaces.

This command results in reading file starting from <offset> bytes from the beginning of the file given by SFID <sfid> or currently selected file and checks against the expected value if specified.

If the <numbytes> is provided as a non - Zero(0) value then Ne is set as <numbytes>. Otherwise if EXPVAL is provided then Ne is set as the length of the EXPVAL. Finally if <numbytes> is not provided or provided as Zero(0) and EXPVAL is not provided then Ne is set as EMPTY or Zero(0).

Write Binary

{WB || WRITEBINARY} [SFI =<sfid>] <offset> <data>

<sfid> is the value of SFID of the file to which we wish to write. If it is not specified, the data is written to the currently selected file. If <sfid> is provided, the corresponding file becomes the currently selected file after the execution of the command.

<offset> is the offset in the file from which writing has to take place.

<data> is the actual data to write.

This command results in writing <data> starting from <offset> bytes from the beginning of the file given by SFID <sfid> or currently selected file.

Update Binary

{UB || UPDATEBINARY} [SFI = <sfid>] <offset> <data>

The meaning and operation of this command is same as that of the Write Binary. This command performs the update binary operation which are not subjected to Write-AND, Write-Or or Write-Once behavior of the file.

Erase Binary

{EB || ERASEBINARY} [SFI=<sfid>] <offset> [<endoffset>]

<sfid> and <offset> have same meaning as in the case of **UB** and **WB**. <endoffset> is the end offset – until which the contents shall be erased. The command shall erase contents of the currently selected file or the file given by SFID <sfid> starting from <offset> to <endoffset>. When <endoffset> is not specified, it defaults to the end of the file.

Read Record

{RR || READRECORD} [SFI=<sfid>] {RNO || RID} = <record>[{FIRST || LAST || NEXT || PREVIOUS || TOLAST || FROMLAST}] <numbytes> [EXPVAL{?= || !=} {<hexstring> || <string>}]

Reads a record from the currently selected file or from a file with SFI <sfid>. Records can be indicated by either a record number (RNO) or record identifier (RID). Keywords (FIRST, LAST, NEXT, PREVIOUS, TOLAST & FROMLAST) can be specified to tell which occurrence of record identifier to be read and in what order to read records. The <numbytes> indicate number of bytes expecting from the record.

If the <numbytes> is provided as a non - Zero(0) value then Ne is set as <numbytes>. Otherwise if EXPVAL is provided then Ne is set as the length of the EXPVAL. Finally if <numbytes> is not provided or provided as Zero(0) and EXPVAL is not provided then Ne is set as EMPTY or Zero(0).

Write Record

{WR ॥ WRITERECORD} [SFI=<sfid>] {RNO=<recordno> {FIRST ॥ LAST ॥ NEXT ॥ PREVIOUS ॥ TOLAST ॥ FROMLAST}} <data>

<sfid> stands for SFID and has same meaning as in other file related commands. Either a record no. can be specified by using the construct **RNO=<recordno>** or a record can be indicated in relation to the last read record using one of the keywords (FIRST, LAST, NEXT, PREVIOUS, TOLAST, FROMLAST). <data> is the data to be written in the record. The command writes <data> to a record either given by <recordno> or by one of the keywords in the currently selected file or the file given by <sfid> if SFI is mentioned.

Update Record

{UR ॥ UPDATERECORD} [SFI=<sfid>] {RNO=<recordno> {FIRST ॥ LAST ॥ NEXT ॥ PREVIOUS ॥ TOLAST ॥ FROMLAST}} <data>

This command is exactly of same format as that of Write Record.

Append Record

{AR ॥ APPENDRECORD} [SFI=<sfid>] <data>

sfid stands for SFID and has same meaning as in the above cases. The data present in <data> is appended to the last operated record.

Activate File

{AF॥ ACTIVATEFILE} {DFNAME ॥ CDF ॥ CEF ॥ PDF ॥ MF ॥ MFPATH ॥ CDFPATH}
[= {<string> ॥ <id>}] {FIRST ॥ LAST ॥ NEXT ॥ PREVIOUS ॥ TOLAST ॥ FROMLAST}
[{FCP ॥ FCI ॥ FMD} [<numbytes>]]
[EXPVAL{?= ॥ != } {<hexstring> ॥ <string>}]

The command activates a file whose address is specified by one of the keywords given above which mean the same as in the case of Select file. Either the FCP or FCI or FMD can be requested as in the case of Select file. The number of bytes expected as response can be specified in <numbytes>.

If the <numbytes> is provided as a non - Zero(0) value then Ne is set as <numbytes>. Otherwise if EXPVAL is provided then Ne is set as the length of the EXPVAL. Finally if <numbytes> is not provided or provided as Zero(0) and EXPVAL is not provided then Ne is set as EMPTY or Zero(0).

Deactivate File

DAF {DFNAME ‖ CDF ‖ CEF ‖ PDF ‖ MF ‖ MFPATH ‖ CDFPATH}
[= {<string> ‖ <id>}] {FIRST ‖ LAST ‖ NEXT ‖ PREVIOUS ‖ TOLAST ‖ FROMLAST}
[{FCP ‖ FCI ‖ FMD} [<numbytes>]]
[EXPVAL{?= ‖ != } {<hexstring> ‖ <string>}]

The command deactivates a file on the card. The meaning of all options is same as in the case of Activate file.

Terminate EF

{TERMEF ‖ TERMINATEEF} {DFNAME ‖ CDF ‖ CEF ‖ PDF ‖ MF ‖ MFPATH ‖ CDFPATH}
[= {<string> ‖ <id>}] {FIRST ‖ LAST ‖ NEXT ‖ PREVIOUS ‖ TOLAST ‖ FROMLAST}
[{FCP ‖ FCI ‖ FMD} [<numbytes>]]
[EXPVAL{?= ‖ != } {<hexstring> ‖ <string>}]

The command transforms the life cycle of EF into termination stage. The meaning of all options is same as in the case of Activate file.

Terminate DF

{TERMDF ‖ TERMINATEDF} {DFNAME ‖ CDF ‖ CEF ‖ PDF ‖ MF ‖ MFPATH ‖ CDFPATH}
[= {<string> ‖ <id>}] {FIRST ‖ LAST ‖ NEXT ‖ PREVIOUS ‖ TOLAST ‖ FROMLAST}
[{FCP ‖ FCI ‖ FMD} [<numbytes>]]
[EXPVAL{?= ‖ != } {<hexstring> ‖ <string>}]

The command transforms the life cycle of a DF into Termination stage. The meaning of all options is same as in the case of Activate file.

Security Related Commands

Verify

VERIFY {GRD ‖ SRD} =<refdata> [<passwd>]

Whether the reference data is global or not is indicated by putting GRD and SRD according to the situation. <refdata> is the reference data no. which is used by SCOSTA to perform the password authentication. The optional value <passwd> is the password that has to be verified. The command verifies a password given by an external entity and updates the security status accordingly.

Change Reference Data

CRD {X ‖ C} {GRD ‖ SRD}=<refdatano> NEWDATA=<new_value> [OLDDATA=<old_value>]

This command shall change a reference data with serial no. <refdatano>, present in the card from <old_value> to <new_value>. The option X or C specifies whether reference data has to be exchanged or just changed.

Reset Retry Counter

RRC {GRD || SRD}=<refdatano> [RCODE=<resetcode> NRD=<newrefdata>]

The command resets the retry counter for a key or password with reference data no. <refdatano>. Optionally, upon completion of resetting retry counter, the reference data can be changed and the new reference data to be used is given in <newrefdata>.

Enable Verification Requirement

EVR {GRD || SRD}=<refdatano> [DATA=<data>]

This command switches on the requirement to compare reference data with the verification data. The verification data can be provided in <data> and the exact reference data is pointed to by <refdatano>.

Disable Verification Requirement

DVR *{GRD || SRD}=<refdatano> [DATA=<data>]*

The semantics of the command is same as that of Enable Verification Requirement. The command switches of the need to verify a reference data pointed to by <refdatano>.

Get Challenge

{GETC || GETCHALLENGE} [ALGO=<algo>]<numbytes> [EXPVAL{?= || !=} {<hexstring> || <string>}]

Algorithm reference can be specified in <algo>. If <algo> is not specified then it is taken as Zero(0). <numbytes> indicates the number of bytes of challenge required. Optionally expected value can be specified.

If the <numbytes> is provided as a non - Zero(0) value then Ne is set as <numbytes>. Otherwise if EXPVAL is provided then Ne is set as the length of the EXPVAL. Finally if <numbytes> is not provided or provided as Zero(0) and EXPVAL is not provided then Ne is set as EMPTY or Zero(0).

The command retrieves a challenge of at most length Ne in either plain text or encrypted using IFD's certified public key by the algorithm specified in <algo>, for later use in an external authenticate command.

Internal Authenticate

{IAUTH || INTERNALAUTHENTICATE} ALGO=<algo> {GRD || SRD}=<refdatano> CHALLENGE=<challenge> [<numbytes>][EXPVAL{?= || !=} {<hexstring> || <string>}]

The algorithm reference is specified in <algo> while the key to be used for internal authentication is specified in <refdatano>. The challenge being issued to the card is specified in <challenge>. Optionally an expected value of the encrypted challenge can be mentioned which is compared against the actual obtained output and an error message reported if there is a mismatch. The command performs an internal authentication of the card with algorithm specified in <algo>, key specified in <refdatano>, and challenge in <challenge>. Optionally the length of the expected response can be specified in <numbytes>.

If the <numbytes> is provided as a non - Zero(0) value then Ne is set as <numbytes>. Otherwise if EXPVAL is provided then Ne is set as the length of the EXPVAL. Finally if <numbytes> is not provided or provided as Zero(0) and EXPVAL is not provided then Ne is set as EMPTY or Zero(0).

External Authenticate

{EAUTH || EXTERNALAUTHENTICATE} ALGO=<algo> {GRD || SRD}=<refdatano>[RESPONSE =] <response>

The algorithm reference is specified in <algo> as before and the key to be used is specified in <refdatano>. The response to the challenge is passed to the card in <response>. The command performs an external authentication using algorithm <algo> and key <refdatano> and the <response> is the answer being given to the card by the external entity. If the key is correct at the outside, the command should succeed.

Mutual Authenticate

{MAUTH || MUTUALAUTHENTICATE} ALGO=<algo> {GRD || SRD}=<refdatano> {CHALLENGE=<challenge> [RESPONSE =]<response> || CHALLENGE_RESPONSE=<data>} [<numbytes>] [EXPVAL{?= || != } {<hexstring> || <string>}]

This command is a combination of Internal Authenticate and External Authenticate. If the CHALLENGE and RESPONSE are explicitly specified in <challenge> & <response> respectively then command data send to the card is <response> concatenated with <challenge>, alternatively the command data can be specified using CHALLENGE_RESPONSE in <data>. Optionally the length of the expected response can be specified in <numbytes>.

If the <numbytes> is provided as a non - Zero(0) value then Ne is set as <numbytes>. Otherwise if EXPVAL is provided then Ne is set as the length of the EXPVAL. Finally if <numbytes> is not provided or provided as Zero(0) and EXPVAL is not provided then Ne is set as EMPTY or Zero(0).

MSE Set

MSESET {CDA || VEA || CDAVEA} <tagvalue> <valuevalue>

The options CDA stands for Computation, Decipherment and Authentication while VEA stands for Verification, Encipherment and Authentication. One of these have to be specified to specify the operation for which this SE is going to be used. <tagvalue> is the value of the tag whose value has to be set in the current SE. <valuevalue> is the value of the value in the TLV structure that is being changed. The command modifies the current SE, replacing the value of an existing tag <tagvalue> by <valuevalue>.

MSE Store

MSESTORE <seno>

<seno> is the number of the SE to which the current SE has to be stored. The command writes current SE to EEPROM overwriting SE number <seno>

MSE Erase

MSEERASE <seno>

This command shall erase an SE with number <seno> from the EEPROM.

MSE Restore

MSERESTORE <seno>

This command shall restore an SE with number <seno> into current SE.

Perform Security Operation

PSO
{CCC // VCC // ENCIPHER // DECIPHER // HASH // COMPUTE_DIG_SIG // VERIFY_DIG_SIG // VERIFY_CERT} <in_data> [<tag_encrypted_data>] [<numbytes>] [EXPVAL {?= //   !=} {<hexstring> // <string>}]

This command shall perform a security operation in the card. CCC is for Computing Cryptographic Checksum, VCC is for Verifying Cryptographic Checksum, ENCIPHER and DECIPHER are for enciphering and deciphering arbitrary length data respectively, HASH is to compute a HASH of arbitrary length data, COMPUTE_DIG_SIG and VERIFY_DIG_SIG are for computation and verification of digital signature inside the card and VERIFY_CERT is for certificate verification inside the card.

For all the PSO commands, <in_data> represents the input data - the data on which the operation has to be performed. For VERIFY_CERT, the <in_data> should be a BER encoded X.509 certificate. For VCC and VERIFY_DIG_SIG, however, the <in_data> includes the signature or the checksum that has to be verified against. The PSO ENCIPHER and DECIPHER commands can take multiple forms of the encrypted data. The tag for this can be specified as <tag_encrypted_data>. For PSO Encipher, this tag is put in P1 while for PSO Decipher, this tag is put in P2. Optionally expected number of bytes and expected response can be provided in <numbytes> and EXPVAL respectively.

If the <numbytes> is provided as a non - Zero(0) value then Ne is set as <numbytes>. Otherwise if EXPVAL is provided then Ne is set as the length of the EXPVAL. Finally if <numbytes> is not provided or provided as Zero(0) and EXPVAL is not provided then Ne is set as EMPTY or Zero(0).

Other Commands

Get Data

{GD || GETDATA} {BERTLV || APPL || STLV || BER2TLV} <tagvalue> <numbytes>[EXPVAL{?= || !=} {<hexstring> || <string>}]

The command retrieves TLV data from the card corresponding to the tag <tagvalue>. The type of tag being requested is indicated by one of the four keywords given. <tagvalue> is the value of the tag whose value is to be retrieved. <numbytes> is the number of bytes expected to be read from this data object. Optionally, one can also specify Expected value of the read data by using the **EXPVAL** construct. The expected value can be specified as a Hexadecimal string or a normal string – which can be a c variable. A hexadecimal string is a string of hexadecimal bytes separated by spaces.

If the <numbytes> is provided as a non - Zero(0) value then Ne is set as <numbytes>. Otherwise if EXPVAL is provided then Ne is set as the length of the EXPVAL. Finally if <numbytes> is not provided or provided as Zero(0) and EXPVAL is not provided then Ne is set as EMPTY or Zero(0).

Put Data

{PD || PUTDATA} {BERTLV || APPL || STLV || BER2TLV} <tagvalue> <data>

The syntax is almost same as get data. <tagvalue> stands for value of the tag that is being inserted, and <data> stands for the value of TLV structure that is being written. The command writes a data object given by <tagvalue>, length of <data> and <data> to the current DF.

Get Response

{GETR || GETRESPONSE} <numbytes> [EXPVAL{?= || !=} {<hexstring> || <string>}]

All terms mean same as that of Get Challenge. The command retrieves any data that is remaining in the output buffer and is indicated by the status bytes of the previous command. Expected number of bytes can be specified in <numbytes>. Optionally expected response can be provided in EXPVAL.

If the <numbytes> is provided as a non - Zero(0) value then Ne is set as <numbytes>. Otherwise if EXPVAL is provided then Ne is set as the length of the EXPVAL. Finally if <numbytes> is not provided or provided as Zero(0) and EXPVAL is not provided then Ne is set as EMPTY or Zero(0).

Command APDU

CAPDU [CLA=<cla>] [INS=<ins>] [P1=<p1>] [P2=<p2>] [Lc=<lc>] [Le=<le>] [INDATA=<indata>] [EXPVAL{?= || !=} {<hexstring> || <string>}]

If a particular command cannot be constructed using all other constructs, this command can be used to construct it from scratch. All the components of a Command APDU – CLA, INS, P1, P2, Lc, Le and Input data can be specified individually. The APDU shall be passed to the card as it is. Optionally expected response can be provided in EXPVAL.

Secure Messaging

Secure Messaging can be applied to any set of commands as described above. When the secure messaging is applicable, the APDUs are first converted to the SM based APDUs and then sent to the card.

The script file starts the execution with no secure messaging. Secure messaging is activated by **SM** command. The **SM** command by itself is not executed on the card. It activates the secure messaging for all the commands that follow the **SM** command.

SM command in addition may provide expected SW1 and SW2 values like in other commands. These are then treated as the SW1 and SW2 of the APDUs as returned from the card for all subsequent commands. The SW1 and SW2 so specified refer to the SM related SW1 and SW2.

SM command takes various arguments which specify the method of translation of an APDU to a secure APDU. The arguments are divided into two groups. The first group provides the details about the DOs that must be provided in the command APDU by the IFD. The second argument provides the DOs that might be present in the response APDU. In the second case, it is an error that is reported if there is an unspecified DO in the response ADPU. A general syntax of the **SM** command is the following.

SM <commandAPDU_tagslist> RESPONSE <responseAPDU_taglist> {SW ?= <value>}

The <value> fields at the end provide possible expected values of SW1 and SW2. The value of expected SW is given in a manner as given earlier.

The <commandAPDU_taglist> provides the specifications of the DOs that must be provided in the command APDU by the STTool when it presents the APDU to the ICC. The <responseAPDU_taglist> provides the specifications of the DOs that might be present in the response. The following are the valid types of items in the tag lists.

- ***NULL***. When ***NULL*** is specified, the tag lists can not contain any other items. Both the commandAPDU_taglist and the responseAPDU_taglist must have only ***NULL***. The following is the only valid SM command when ***NULL*** is used.

  SM NULL RESPONSE NULL;

  This is also the SM command in effect when the scrip execution is started in the beginning.

- ***CC (<cryptospec>)***. This tag is used to indicate the cryptographic checksum related parameters. When used in the commandAPDU_taglist, it indicates that the cryptographic checksum must be computed for all the DOs in the command APDU which start with the odd tag. The <cryptospec> indicates the algorithm, key and IV for the computation (by the IFD in the commandAPDU_taglist) and verification (by the IFD in the responseAPDU_taglist) of the cryptographic checksum. The ***CC*** can only be the last tag of the <taglist>.

- ***CDATA***. This tag can occur in the commandAPDU_taglist and responseAPDU_taglist. It indicates that the corresponding APDU may include the data (command or response) of the APDU tagged under 0x80 data object.

- ***CDATA_AUTH***. This tag is behaviorally same as ***CDATA*** except that when it occurs in an APDU, it is also included in the computation of cryptographic checksum. Correspondingly a DO with tag 0x81 is used.

- ***CH_AUTH***. This tag is meaningful only in the commandAPDU_taglist where it indicates that the command header must be included as a DO in the command APDU. Accordingly tag 0x89 is used. When this tag is used, it is always included in the computation of cryptographic checksum if it is used in the commandAPDU_taglist.

- ***CHINCLUDE***. This tag can only occur in the commandAPDU_taglist. When ***CHINCLUDE*** tag is specified, it indicates that the command header must be used in the authentication. This also indicates that CLA=0x0C shall be used while passing the APDU to the ICC.

- ***CRYPTODO (<cryptospec> <taglist>)***. This tag is used to create a 0x82 template in the corresponding APDU. The algorithm, key and IV for encryption and decryption are specified as <cryptospec>. In the commandAPDU_taglist, this tag provides the details for the encryption (on the IFD side) and the <taglist> can include only those tags which are valid for the commandAPDU_taglist. In the responseAPDU_taglist, this tag provides the details for the decryption (on the IFD side) and the <taglist> can include only those tags which are valid for the responseAPDU_taglist. Occurrence of this DO in the response is optional.

- ***CRYPTODO_AUTH (<cryptospec> <taglist>)***. This tag is used in a similar manner as the ***CRYPTODO*** when it occurs in an APDU, it is also included in the computation of cryptographic checksum. Correspondingly a DO with tag 0x83 is used.

- ***DO (<string>)***. This tag is used to include a DO in the corresponding APDU. The tag is not processed at all except for looking into the tag. The first byte of the string provides the tag, second byte the length and the remaining bytes provide the value. When the tag is an odd number, it is also included in the computation of the cryptographic checksum. When ***DO*** is used in the responseAPDU_taglist, it indicates that the response must include this DO in exactly the same format as specified.

- ***ENCDATA (<cryptospec>)***. This tag is used by the script to encrypt the command data under tag 0x86 in the commandAPDU_taglist. When it occurs in the reponseAPDU_taglist, it is used to indicate the decryption (on the IFD side) of the response data. Accordingly tag 0x86 is used. The <cryptosepc> describes the algorithm, key and IV for the encryption or decryption. Occurrence of this DO in the response is optional.

- ***ENCDATA_AUTH (<cryptospec>)***. This tag is used in a similar manner as the ***ENCDATA*** when it occurs in an APDU, it is also included in the computation of cryptographic checksum. Correspondingly a DO with tag 0x87 is used.

- ***LE***. This tag is meaningful only in the commandAPDU_taglist where it indicates that the Le must be encapsulated under tag 0x96.

- ***LE_AUTH***. This tag is meaningful only in the commandAPDU_taglist where it indicates that the Le must be encapsulated under tag 0x97. This DO is also used in the computation of the cryptographic checksum if used.

- ***PLAINDO (<taglist>)***. This tag is used to create a 0xB0 template in the corresponding APDU. The data field is composed of the tags as given in the <taglist>. This tag when it occurs in the commandAPDU_taglist can include only those items in the <taglist> which are valid for the commandAPDU_taglist. Similarly when it occurs in the responseAPDU_taglist, only those items which are valid for the responseAPDU_taglist can be included in the <taglist>. Occurrence of this DO in the response is optional.

- ***PLAINDO_AUTH (taglist)***. This tag is similar to the ***PLAINDO*** tag except that it is also included in the computation of the cryptographic checksum.

- ***STATUS***. This tag can only occur in the responseAPDU_taglist. When included it indicates that the response APDU must have the SW1 and SW2 given under tag 0x99. If the response APDU does not include this tag, or when the length of this tag is 0, it is considered as 0x99, 0x02, 0x90, 0x00. If the cryptographic checksum is included in the response, this DO in the response is included in the computation of the cryptographic checksum.

Specifying an Algorithm

An algorithm and its related parameters for cryptographic applications are specified using the ***<cryptospec>*** constructions in various SM related tags such as ***ENCDATA***, ***ENCDATA_AUTH***, ***CRYPTODO***, ***CRYPTODO_AUTH***, and ***CC***. The ***<cryptospec>*** includes one or more of the following.

- ***KEYVAL = <string>.*** This provides the key for the algorithm. It is a required argument for the ***<cryptospec>*** and assumes no default value.

- ***IV = <string>.*** This provides the initial value for the algorithm. If not specified, the default value is assumed as per the algorithm. For the **tdes-cbc** and **tdes-mac** algorithms, the default value of the ***IV*** is all-zeros. For the **epp-tdes-mac-nossc**, algorithm, the IV is not used. Any value given for this is disregarded by the algorithm to compute the cryptographic checksum. For the **epp-tdes-mac** algorithm, ***IV*** provides the values of the SSC which is first incremented before computation of the checksum. The default value of the ***IV*** is zero (i.e. SSC is zero, and it is incremented prior to using). Since no storage for the IV is specified, the incremented value of SSC (i.e. 1) is discarded. For this algorithm, since the SSC is first incremented, only a C++ variable must be specified as an input to the IV. A constant string is accepted but results in a runtime error since it can not be a proper r-value in a C++ program.

- ***ALGO = <const_string>*** The following are the possible values for the algorithms.
    - ***tdes-cbc:*** This algorithm is used for the encryption and decryption. It is also the default under ***ENCDATA***, ***ENCDATA_AUTH***, ***CRYPTODO***, ***CRYPTODO_AUTH*** tags. This algorithm encrypts the plaintext using the specified IV and key (16-byte long).
    - ***tdes-mac****:* This algorithm is used for the computation of the cryptographic checksum. The basic core of this algorithm is identical to the ***tdes-cbc*** except that it provides only the last block of the ciphertext. This algorithm is the default algorithm in ***CC*** tag.
    - ***epp-tdes-mac***: This algorithm is used for the computation of the cryptographic checksum. The value of the IV is used as a SSC (Send Sequence Counter). The SSC is first incremented prior to the computation of the checksum. Therefore the IV must be a read write location. It would be a run-time error to specify the IV as a constant.
    - ***epp-tdes-mac-nossc***: This algorithm is essentially same as the ***epp-tdes-mac*** algorithm except that no SSC is used. Accordingly the value specified in the IV is disregarded.

Cryptographic Algorithm Functions useable in the scripts

STTool exports the strong cryptographic mechanisms of command library. The script writers can use these cryptographic mechanisms through the following C++ function.

```
int CryptoFunc(int op,
        const unsigned char *indata, int indatalen,
        unsigned char *outdata, int outdatalen,
        const char *algo,
        const unsigned char *keyval, int keylen,
        unsigned char *ivval, int ivlen);
```

The value of the *op* can be one of the following constants (ENCRYPT, DECRYPT, CC_COMPUTE, CC_VERIFY, HASH, HASH_VERIFY, DSIG_COMPUTE & DSIG_VERIFY). The *indata* and *indatalen* provide the input message. The *outdata* and *outdatalen* provides the pointers where the result will be stored and the length of the memory buffer. The *keyval* and *keylen* provide the key for the algorithm. The *ivval* and *ivlen* provide the IV for the algorithm. The *algo* is the character string for the algorithm as specified earlier. In addition to the list of crypto algos given earlier, following algorithms can also be used:

- **sha1:** This algorithm is SHA-1 algorithm. It is used for the computation of hash. The *keyval, keylen, ivval and ivlen are* disregarded for this algorithm.

- **RSA Cryptographic Algorithms:** These algorithms are Asymmetric Key algorithms as defined in PKCS#1 v2.1. They provide methods for encryption, decryption, digital signature generation and digital signature verification (ENCRYPT, DECRYPT, DSIG_COMPUTE & DSIG_VERIFY respectively). For encryption and digital signature verification the key provided should be a BER encoded PKCS #1 RSA Public key. For decryption and digital signature verification the key provided should be a BER encoded PKCS #1 RSA Private key. The *ivval* and *ivlen* are disregarded for these algorithms.
    - **rsa_pkcs1_v1_5:** This algorithm supports encryption (RSAES-PKCS1-v1_5), decryption (RSAES-PKCS1-v1_5), digital signature generation (RSASSA-PKCS1-v1_5) and digital signature verification (RSASSA-PKCS1-v1_5).
    - **rsa_pkcs1_v2_1:** This algorithm supports encryption (RSAES-OAEP), decryption (RSAES-OAEP), digital signature generation (RSASSA-PSS) and digital signature verification (RSASSA-PSS).

How to execute a script

A script written according to the syntax given in the previous section will go through the following steps to obtain an executable test.

1) This script is compiled using script compiler. For this, the following commands need to be executed while inside the work directory.

   ```
   $ ../frontend/testtool  –s  <path to the script>  [-c  <output_file>]  [–h <header file>]
   ```

   The structure of paths to the files may be different for linux and windows platforms. The options -c and -h are optional and if not specified the default names shall be out.cpp and header.h respectively. This shall produce a C++ file in <output_file> **and** a header file <header_file>.
2) The final executable test is created by the following command

   ```
   $ ./make_final
   ```
   for linux and
   ```
   $ make_final.bat
   ```

   for windows. This shall compile the out.cpp program in combination with the backend and the command library and produces the final executable **TestTool** on linux and **TestTool.exe** on windows
3) The resulting executable can be run by the following command:
   ```
   $ ./TestTool  [-p T0|T1|RAW|T0RAW|T1RAW|T0T1|ANY] [–e <logfile_name>]  [-r  <reader name>]
   ```
   -p option is for specifying the protocol to be used (ANY means any one of T0, T1 or RAW). **–e** option specifies the file name for the log file to be used and **–r** specifies the reader name to be

used in particular. All options are optional. If **–e** option is not specified, the log appears on the screen. If **–p** and **–r** options are not specified, then the user is given a list of protocols or readers and is asked to choose one.

4) The above command shall start executing the script and the prompt returns after the execution is complete. The command, status of the response and the response bytes received are printed in the log file for each command.
5) The script file shall contain a message flagged with text starting with "**FAILURE"** for all those commands whose expected status did not match with the obtained status OR whose expected value did not match with the actual obtained response bytes. So, if a search for the above text in the log file does not have any results, then it is highly probable that the script command is exactly what we intend and is being executed correctly on the card.

Developer's Manual

STTool uses the **command library** for communicating with Smart Cards. More details about command library can be found in the command library documentation.

List of files and functionality

The following are the source files present in the frontend.

- testlib.c – This contains routines which print the command library function corresponding to each command in the script in out.cpp. These routines are called from inside the yacc file (testtool.y).
- testtool.l – This is the standard lex file containing possible tokens in the script file.
- testtool.y – This is the standard yacc file, containing the grammar, rules and corresponding actions. For most of the rules, actions comprise of collecting the parameters of the command and call the corresponding function in testlib.c which shall print out the command library function in out.cpp.
- Makefile.linux and Makefile.win – Two different Makefiles for linux and windows platforms. The code is platform independent and works on both platforms without any modifications.

The following are the source files present in the backend directory.

- isolib.h and isolib.cpp – This file contains the routines which are called from out.cpp. The file provides routines for initializing STTool, connecting to the desired reader and logging STTool script execution results.

Building the STTool frontend

The frontend or the script compiler can be built by using the Makefiles present in the frontend directory.

$**make –f Makefile.linux**

on linux and

$nmake /f Makefile.win
on windows.

The output shall be an executable with name **testtool** in linux and **testtool.exe** in windows in the same directory.

Building the STTool backend

The backend can be built by using the Makefiles present in the backend directory.
$make –f Makefile.linux [CMDLIBPATH=<path_of_Command_Library>]

on linux and
$nmake /f Makefile.win [CMDLIBPATH=<path_of_Command_Library>]
on windows.

The output shall be a library with name **loadlibrary.so.1.0** in linux and **loadlibrary.lib** in windows.

Adding new constructions

To add new constructs to the scripting language used, the following needs to be done.

- Add the corresponding tokens in testtool.l file if there are any new ones.
- Add the rules for the grammar and the actions for each rule.
- If the rule is taking care of a new operation, add a function to testlib.c to handle it. This function merely collects the parameters of the command and prints out the command library code for handling the command into output c++ file. The action for the rules in testtool.y shall call this function with proper arguments.

Examples of generated code from the script

The following example shows a sample command in the script and the corresponding code generated by the frontend or the script compiler.

CREATEFILE FILEID = 0x5f01 INTERNAL VARLENGTH MNR = 07 MRL = 08 SFI = 1 DATACODING = WRITE_ONCE;

The code generated is as follows:

```
checksw = (char*)"????";
T_lineno = 1;
T_command = (char*)"CREATEFILE FILEID = 0x5f01 INTERNAL VARLENGTH MNR = 07 MRL = 08 SFI = 1
DATACODING = WRITE_ONCE";

fcp = new FCP(0x5F01);
process_return_code(fcp->setFD(FDB_INTERNAL_EF | FDB_LINEAR_VARIABLE, DCB_WRITE_ONCE, 8,
7));
```

```
process_return_code(fcp->setShortfid((BYTE)1));
apdu = new CreateFileAPDU(fcp, logCommand);
delete fcp;
fcp = NULL;
RespLen = 0;
fprintf(stderr, "\nLine %d:\t%s\n", T_lineno, T_command);
process_return_code(apdu->sendAPDU(*reader, SM));
if(apdu != NULL)      delete apdu;
apdu = NULL;
```

An example of an SM command

The following is an example of the SM specifications.

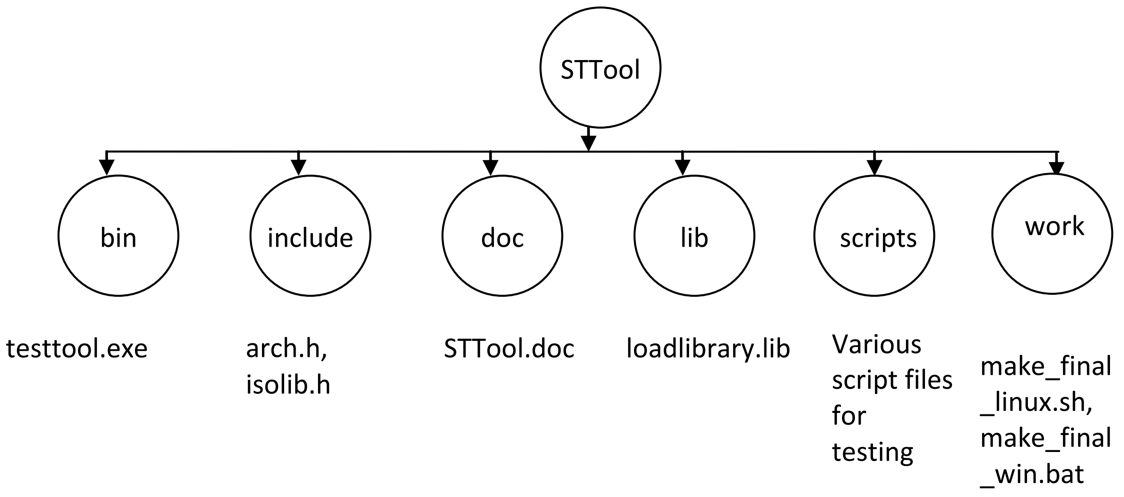
SM CHINCLUDE ENCDATA_AUTH (ALGO=”tdes-cbc” KEYVAL=deskey IV=zero) LE CC (ALGO=”tdes-epp-mac” KEYVAL=Mackey IV=zero) RESPONSE ENCDATA_AUTH(ALGO=”tdes_cbc” KEYVAL=deskey IV=zero) STATUS SW?=OK;

This SM command indicates that the command APDU is formed using tag 0x87, 0x96 and 0x9E. In the computation of the CC, tdes-epp-mac algorithm is used with the key being in a C++ variable Mackey and IV being in a C++ variable zero. The tag 0x87 is formed by encrypting the command data using tdes-cbc algorithm for which the key and the IV are given in C++ variables deskey and zero. The CC computation includes command headers and encrypted data but does not include the DO for the Le.

Distribution of the STTool

The STTool is distributed for Windows XP in a single file called **STTool.zip**.

This file should be unzipped in a temporary directory and **install.bat** should be executed for the installation. After the installation is completed, a directory structure is created under the **Program Files** directory of the standard Windows XP installation.



How to run a script

The script distribution is for standard Windows XP distribution. Prior to running a script ensure that the following are available.

1. MicroSoft Vistual C++ Compiler with Visual Studio .NET 2003.
2. The PATH environment variable should contain path to the cryptographic algorithm libraries provided with Command Library.

In order to run a script, issue the following commands on the command prompt of Windows XP command tool from inside the work directory.

1. **..\bin\testtool.exe -s ..\scripts\script_file_name**
2. **make_final_win.dat <path_of_Command_Library>**
3. **Testtool.exe**