# Tester Bench For jQuery

## User Manual

Document status: draft.

# Table of Contents

## Revision history

| ***Revision Date*** | ***Comments*** |
|---|---|
| 15.07.2010 | Initial Draft |
| 31.08.2010 | Proxy and *localhost* added |
| 03.10.2010 | XML logging, console filters, snapshot(). |
| 12.10.2010 | Release 0.9.0 |

## General Disclaimer

This version of the "User Manual" still may have gaps in presentation of the features, even though the features are already available; some important topics may be missing from this document.

When any of such topics are concerned, please user Forum http://forum.testerbench.com/ or send a question to testerbench@mouseclicktechnologies.com.

# 1. Installation

This chapter will guide you through the installation process.

### 1.1 Pre-requisites

Tester Bench can be installed on any recent Windows: Windows XP, Vista, or Windows 7. Service pack level should not be important, as soon as your browser is stable enough to open the web pages you are testing.

Beta version is distributed in form of MSI package, which may cause dependency on the msiexec in certain non-upgraded systems. Final release will have also EXE distributable which will resolve these dependencies for the affected systems.

Tester Bench installer requires administrative rights in order to complete the installation.

### 1.2 Installing Tester Bench For jQuery

- Execute TesterBench.MSI

The installation takes less than 5 minutes.

Tester Bench will be installed locally, and the file extension ".jsq" will be associated with the main application TesterBench.exe.

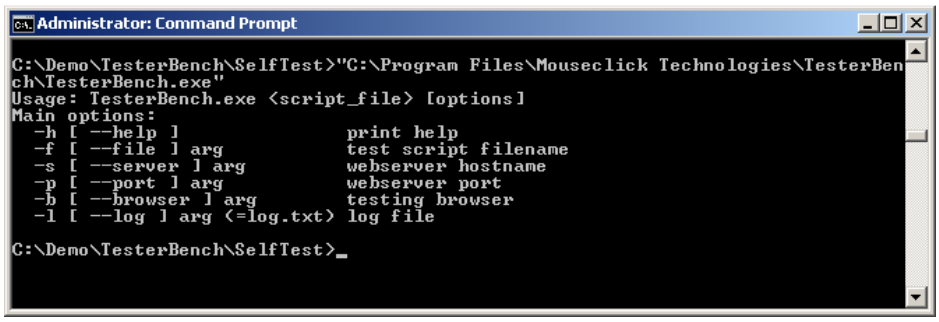
Syntax of the ".jsq" script files is described in the "API reference" chapter.

## 2. Running Tester Bench

In the Beta version release there are two ways of running Tester Bench application: in batch mode, or in a test editor. Even though it is possible to redirect standard output into a file, and therefore use batch mode quite efficiently, we expect that most users will prefer running test scripts directly from the test editor. Both options are briefly described below.

### 2.1 Tester Bench in batch mode

This is the first and the easiest way to execute Tester Bench. If you execute Tester Bench without arguments, for example:
"C:\Program Files\Mouseclick Technologies\Tester Bench\TesterBench.exe"
you will see something like this:




When you are writing your first test script, which will be probably called test001.jsq, you may execute this script in different browsers using command line:

"C:\Program Files\Mouseclick Technologies\TesterBench\TesterBench.exe" test001.jsq –bchrome
"C:\Program Files\Mouseclick Technologies\TesterBench\TesterBench.exe" test001.jsq -bexplorer
"C:\Program Files\Mouseclick Technologies\TesterBench\TesterBench.exe" test001.jsq -bsafari
"C:\Program Files\Mouseclick Technologies\TesterBench\TesterBench.exe" test001.jsq -bfirefox

You can also execute your script using system default browser:

"C:\Program Files\Mouseclick Technologies\TesterBench\TesterBench.exe" test001.jsq

## 2.2 Tester Bench integration with a text editor

Most of the popular text editors allow for a compiler execution, with logging into a separate editor window. This allows to execute test scripts directly from the editor, and have all major browsers pre-configured in the compiler command line.

For example, if you are using notepad++, you would typically use NppExec plugin, and create compiler shortcuts with a command line as follows:

```
"C:\Program Files\Mouseclick Technologies\TesterBench\TesterBench.exe" "$(FULL_CURRENT_PATH)" -bchrome
"C:\Program Files\Mouseclick Technologies\TesterBench\TesterBench.exe" "$(FULL_CURRENT_PATH)" -bexplorer
"C:\Program Files\Mouseclick Technologies\TesterBench\TesterBench.exe" "$(FULL_CURRENT_PATH)" -bsafari
"C:\Program Files\Mouseclick Technologies\TesterBench\TesterBench.exe" "$(FULL_CURRENT_PATH)" -bfirefox
"C:\Program Files\Mouseclick Technologies\TesterBench\TesterBench.exe" "$(FULL_CURRENT_PATH)" -bopera
```

Similar configurations are possible for TextPad

## 2.3 Using Tester Bench IDE

Tester Bench IDE is not available with the Beta public preview release. In the final release this option will provide native GUI for test management and test result analysis.

## 2.4 Using Tester Bench with a proxy

In order to control the page within the browser, Tester Bench installs a local proxy, and configures the system to use this local proxy by default. Port of the local proxy is defined by the configuration setting in the config.xml file. Default port number is 8098. Port number can be changed if necessary.

If browser connection to the tested systems was “direct”, then no other actions are required. Tester Bench proxy will set the required proxy port for the browser on startup, and restore the browser settings on exit.

NB: Firefox must have “Use system proxy settings” enabled.

If any of the tested pages require another proxy to be used, then Tester Bench should be configured to use the required proxy. Tester bench configuration settings allow to specify proxy manually, and to use several automatic methods. Tester Bench can use proxy.pac or pad methods, either specified directly, or as inherited from the Internet Explorer settings.

NB: Manually configured proxy is recommended, other options are still experimental. Automatic proxy detection may fail under Windows XP.

## 2.5 Using Tester Bench with *localhost*

Tester Bench can test pages on *localhost* transparently, the same way as on any other host (web server). However, internally, this requires a minor trick: you will notice http://localhost./ instead of http://localhost/ in the URL line. Testing host does this rewrite to work around *localhost* direct access. Similarly, “127.0.0.1“ will be rewritten as “127.0.0.1.” (trailing dot). This behaviour is transparent for the user in the sense that the original notation can and should be used:

open(http://localhost:8855/mypage.html);

or

open(http://127.0.0.1:8855/mypage.html);

# 3. Testing and Scripting

## 3.1 Scripting language

Scripting language is native JavaScript (windows script host) with extensions. Built in extensions allow page control and jQuery interoperability. User extensions are allowed provided CommonJS specification is respected.

## 3.2 Executing scripts: tests, scripts, suites

Test: single test is executed in a single sequence of actions planned in linear fashion in one browser, on a single page, or, possibly, on a number of pages, where user actions cause navigation from one page to another.

Script: single ".jsq" file, which can contain one or more tests, as above.

Suite: single ".jsq" file which references script files by calling host "run" method, e.g.

*run("scripts/script.jsq")*

All scripts in a suite are executed using the same test settings and logging will be collected in a single logfile (using the same console).

## 3.3 Creating your own extensions

Library of component testing can be shared between testers and across pages deploying component libraries. In order to do so, test libraries should be implemented following CommonJS standard.

# 4. Browser quirks

With new versions of browsers being released every month, we are far from being able to guarantee that nothing will be ever broken by the new release, or that final version of Tester Bench will be final in such a way that it will handle all forthcoming browsers in a pre-emptive fashion.

On the contrary, we feel very lucky that at this moment we can promise to support testing of as many browsers, as the web development can offer. In other words, if you can produce a website that seamlessly works in a certain number of (stable) browsers, we will be able to provide seamless testing experience for the same browser base. At this moment we are targeting jQuery support list, therefore choosing jQuery as our reference industry standard, however, not limiting usage of Tester Bench with other development frameworks.

# 5. API Reference

## 5.1 Host Methods

***assert(value)***

returns Assertion class

***echo(text)***

outputs text to console window and log file.

***require(name)***

includes external script module/library.
(see CommonJS module - http://commonjs.org/specs/modules/1.0/)

***run(path)***

runs test script. used in test suites.

***screenshot(name)***

produces a full screen snapshot.

***test(func)***

runs test function.

***wait(milliseconds)***

suspends script execution for a specified interval.

## 5.2 Assertion Class Methods

***.valueOf()***

returns the object value
called automatically when the object is used in 'value' context

***.toString()***

returns the object value as string
called automatically when the object is used in 'string' context

***.is(value)***

strict equality (x === y), chainable (returns original assertion object or remote element)

***.not(value)***

strict equality, negative (x !== y), chainable

***.equals(value)***

equality (x == y), chainable

***.notEquals(value)***

equality, negative (x != y), chainable

***.greaterThan(value)***

greater than (x > y), chainable

*.lessThan(value)*

less than (x < y), chainable

*.greaterOrEquals(value)*

greater or equals (x >= y), chainable

*.lessOrEquals(value)*

less or equals (x <= y)}, chainable

*.between(min, max)*

between min and max (x >= y && x <= z), chainable

*.contains(text)*

contains text (String(x).indexOf(y) >= 0), chainable

*.matches(regexp)*

matches regular expression (y.test(x)), chainable

## 5.3 Page Control Methods (global)

*open(url)*

opens a page.

*type(keys)*

sends keystrokes to the active page. For sending special characters (key codes) please consult Appendix 1.

*call(func, arg1, arg2, …)*

executes function remotely in the active browser/page

*$(selector, description)*

selects element(s) in the active browser/page.

returns Element object

## 5.4 Page element methods (general)

*.screenshot(name)*

produces a full screen snapshot, chainable.

*.echo(text)*

outputs text to console window and log file, chainable.

*.wait(milliseconds)*

suspends script execution for a specified interval, chainable.

*.type(keys)*

sends keystrokes to the active page, chainable.

## 5.5 Page element methods (mouse)

*.click()*

Executed by the OS agent; chainable.
With or without arguments, as follows:

.click() is the same as .click(“center”)

.click(dx, dy) offsets dx,dy from top left corner of the element

.click(‘reference’) positions on the reference point, where ‘reference’ can be one of the following:

'center', 'left', 'top', 'right', 'bottom', 'top left', 'top right', 'bottom right', 'bottom left'; respective reference point being either a corner, or a middle point of the respective edge of the (rectangular) element.

.click(‘reference’, dx, dy) offsets dx,dy from the respective reference point.

### *.dblclick()*

Executed by the OS agent; chainable.
With or without arguments, same as above.

### *.mousemove()*

Executed by the OS agent; chainable.
With or without arguments, same as above.

### *.mousedown()*

Executed by the OS agent; chainable.
With or without arguments, same as above.

### *.mouseup()*

Executed by the OS agent; chainable.
With or without arguments, same as above.

## 5.6 Page element methods (jQuery)

### *.width()*

gets the current computed width for the first element in the set of matched elements,
returns remote Assertion object.

### *.height()*

gets the current computed height for the first element in the set of matched elements,
returns remote Assertion object.

### *.text()*

gets the combined text contents of each element in the set of matched elements, including their descendants,
returns remote Assertion object.

### *.val()*

gets the current value of the first element in the set of matched elements,
returns remote Assertion object.

### *.html()*

gets the HTML contents of the first element in the set of matched elements,

returns remote Assertion object.

### *.css(name)*

gets the value of a style property for the first element in the set of matched elements,
returns remote Assertion object.

### *.attr(name)*

gets the value of an attribute for the first element in the set of matched elements,
returns remote Assertion object.

### *.size()*

gets the number of DOM elements matched by the jQuery object.
returns remote Assertion object.

### *.index(selector)*

search for a given element from among the matched elements.
returns remote Assertion object.

### *.innerHeight()*

gets the current computed height for the first element in the set of matched elements, including padding but not border.
returns remote Assertion object.

### *.innerWidth()*

gets the current computed width for the first element in the set of matched elements, including padding but not border.
returns remote Assertion object.

### *.outerHeight()*

gets the current computed height for the first element in the set of matched elements, including padding, border, and optionally margin.
returns remote Assertion object.

### *.outerWidth()*

gets the current computed width for the first element in the set of matched elements, including padding and border.
returns remote Assertion object.

### *.scrollLeft()*

gets the current horizontal position of the scroll bar for the first element in the set of matched elements.
returns remote Assertion object.

### *.scrollTop()*

gets the current vertical position of the scroll bar for the first element in the set of matched elements.
returns remote Assertion object.

### 5.7 Page element methods (jQuery traversal)

first, last, next, nextAll, nextUntil, prev, prevAll, prevUntil,
siblings, children, contents, parent, parents, parentsUntil,
not, has, filter, find, closest, add, andSelf, end

Please see jQuery reference at http://docs.jquery.com/

### 5.8 Page element methods (jQuery manipulation)

addClass, removeClass, toggleClass, removeAttr,
after, before, append, appendTo, prepend, prependTo,
insertAfter, insertBefore, replaceAll, replaceWith,
empty, detach, remove,
wrap, wrapAll, wrapInner, unwrap,
data, removeData

Please see jQuery reference at http://docs.jquery.com/

### 5.9 Page element methods (jQuery effects)

hide, show, animate, stop,
fadeIn, fadeOut, fadeTo,
slideDown, slideToggle, slideUp

Please see jQuery reference at http://docs.jquery.com/

### 5.10 Page element methods (jQuery events)

error, load, unload,
keydown, keypress, keyup,
mouseenter, mouseleave, mouseout, mouseover,
hover, toggle, resize, scroll,
focus, blur, focusin, focusout,
change, select, submit,
bind, unbind, one, delegate, undelegate,
live, die, trigger, triggerHandler

Please see jQuery reference at http://docs.jquery.com/

## 6. Logging

### 6.1 Logging Streams

We made an attempt to foresee essential logging features which would enable you to easily integrate Tester Bench into your normal workflow. We created four streams of logging, as follows:

- Console

- Planar log
- Structured XML log
- System log

The purpose of these logging streams is briefly explained below.

### 6.2 Demo Preview version limitations

Demo preview version of the Tester Bench package available for free download has only one logging stream: console.

### 6.3 System logs

System logs are the usual ones. Software error messages and info messages are written respectively in the system log files, which are always located in the user directory

```
%APPDATA%\Mouseclick Technologies\Tester Bench\SystemLogs
```

### 6.4 Planar logs

Planar logs constitute the journal of the testing process. Every action, every assertion, and every test result is written in a planar log file. Names of the log files are generated using the test name and the timestamp corresponding to the moment when the test started. For example, test file `jquery.jsq` will generate log files named similar to "`jquery_20100911T092603.087219.log`". Planar log contains all necessary information, however, it is not the best suited for human or even machine reading. The next two formats are covering this gap.

Planar logs are written in the day folder generated automatically, e.g. "2010-Sep-11".

Containing folder can be configured using the "test_results_path" key.

```
<parameter id="test_results_path" type="string">

<value></value>
</parameter>
```

If not defined, the default path is

```
%APPDATA%\Mouseclick Technologies\Tester Bench\TestResults
```

### 6.5 Structured XML logs

We chose structured xml as the most common intermediate format for most of the enterprise users. Provided with structured XML, you can easily find necessary data either manually, or programmatically. Structured XML can be easily parsed, and uploaded into the database. You can use CSS or other standard tools to automate your data analysis, as soon as your test results are presented in XML form. Naming convention is the same as the convention for the planar log files, and they are always saved in the same day folder.

XML tree structure hierarchically represents the organisation of the test procedures. The test hierarchy is constructed as follows:

suite/testfile/test/page/element/assert/result

Each Suite may contain several test files. Test file may contain several tests, etc.

Some other nodes are non-hierarchical, such as: “info”, “message”, “error”.

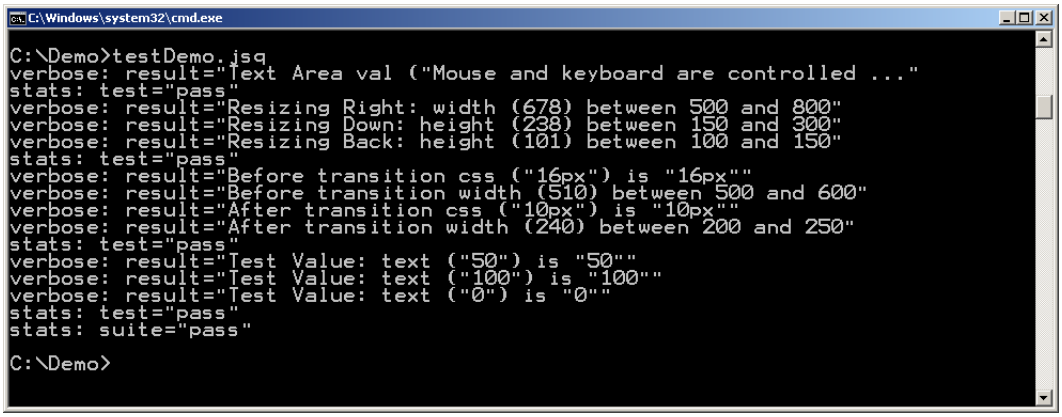
Planar log is produced in such a way that the consistency of the hierarchical logging is guaranteed.

### 6.6 Console logs

Console logging offers a quick monitoring tool to watch the test in progress. Console presents a filteres subset of the planar log with minimum information, or as much information as you choose. You can filter your messages using the configuration key “`console_log_level`”, for example, as follows:

```
<parameter id="console_log_level" type="string">
  <value>error|stats|verbose</value>
</parameter>
```

This choice of the console filter will produce console output similar to the following:




So, the console will display only filtered types of log messages. In the example above, “verbose” will print the results of the assertions in readable form; “stats” will print test results and suite result summary .

While printing on the console, the data will be at the same time written in the console.log file. This file “console.log” is an extended replica of the console logging stream: console.log file is written in utf8 encoding, which allows to handle strings that are not suitable for your console locale.

### 6.7 Logging typology

Below is a full list of logging types, with brief explanations.

| Type | Comment |
|---|---|
| action | User action such as mouse move, keyboard type, etc. |
| assert | Verification method designed to validate page elements and attributes. |
| element | HTML element on the page, usually obtained by a selector $("expression"); |
| error | Script error. Duplicated in the system log as "error" |
| info | Any relevant info, e.g. return vaue of a script call |
| message | User message. Duplicated in the system log file as "info" |
| page | Single HTML page (single URL) |
| param | Parameter of an action or a script call |
| result | Result of an assert method. Single assertion constitutes an atomic test which can "pass" or "fail". |
| script | Script action, such as jquery call, etc. |

| | |
|---|---|
| screenshot | Screen snapshot |
| stats | Summary of the assertion "result" type for objects that may contain multiple assertions. |
| suite | Test suite file |
| test | Test function defined in the test file. One test file may contain several tests. |
| testfile | Test file |
| verbose | Human readable summary of the results, stats, or other significant test conclusions. |

The typology is defined in a way which enables us o represent the test procedure in a form of a structured XML document. In order to achieve this, we define strict hierarchy combined with non-hierarchical leaf nodes.

Hierarchy is strict for the following types:

suite/testfile/test/page/element/

Non-hierarchical types are: “error”, “info”, and “message”. These types of messages may relate to any level of the hierarchy, depending on the test execution.

Semi-hierarchical types are: “action”, “assert”, “script”, “screenshot”. They will appear under “page” or under “element”, depending whether they were chained or non-chained at the execution time. “param” type is used to log function arguments for these types.

Finally, two types are specific: “result” can only belong to “assert”; and “stats” can belong to “test” or “suite”. “verbose” is a human readable form of “assert/result” data.

Finally, why does it look so complicated? We tried to suggest data structure which would be natural, human readable, machine readable, and ready to be integrated with customized data processing. We hope that starting from here it will become only easy.

### 6.8 Screenshots

Screenshots are saved in the daily images folder, e.g. “2010-Sep-11\images”, under “test_results_path”. Default screenshot format is **png**. If, for whatever reason, png format fails, you can use bmp format. Bmp format should never fail. Obviously, we recommend using png due do disk space considerations. PNG offers space saving by factor of approximately 20 times. Actual size of the screen shots will depend on your screen configuration. Log files incorporate links to the screenshots. The names of the screenshots are generated by the screenshot procedure, using a custom name, with uuid as a suffix. For example, the following command:

screenshot(“thispage”);

will generate a screen image with a name similar to:

thispage_f8a079dd-5061-4bb0-b26d-7bee28f26f1b.png

This name will be logged in the planar log and in the XML log as a parameter, to enable linking of the test actions and test results to the respective screenshots. If you would like to print screen shot names in the console, you just add “snapshot” type to your console filter.

### 6.9 Break on fail

We allow multiple assertions to be verified within the same test function. If any of the assertions fails, we account this test as failed. Starting from this point, there are two testing philosophies. Some people think that if any assertion has failed, then there is no need to test further. Those people think that further testing will only collect more garbage information. Also, usually, a failed assertion only shows that something else has fundamentally failed elsewhere.

Other people think that every failed assertion gives us a clue about what went wrong and where, and even if we already know that the test has failed, we should collect all assertion results for further analysis.

In order to give some comfort to users of both confessions, we provide a setting in the configuration file

```
<parameter id="break_on_fail" type="boolean">
<value>0</value>
</parameter>
```

This setting allows to switch between these two philosophies. If value="0" (=false) then further assertions will be executed even after a failure. If value="1" (=true) then test function will terminate after first failed assertion.

### 6.10 Counting results

In our hierarchy we always validate assertions, i.e. we compare various values with expected pre-defined values. Assertions may pass, or fail. One or more assertions may be validated within one single test function. One or more test functions may be defined within a single test file. One or more test files may be executed in a sequence, therefore becoming a test suite. If any assertion in a test has failed, we consider that the test has failed. If any test in a suite has failed, we consider that the suite failed. Furthermore, we can foresee that number of executed assertions within a single test may depend on the runtime conditions, and therefore we think that it would be only confusing to count assertions pass/fail in the results. On the contrary, number of test functions should not change (realistically, it could, but that is not our concern at this point). Therefore we count tests pass/fail and log them respectively.

The following example of console logging illustrates our approach:

```
verbose: result="Before transition css ("16px") is "16px""
verbose: result="Before transition width (510) between 500 and 600"
verbose: result="After transition css ("10px") is "10px""
verbose: result="After transition width (240) between 200 and 250"
stats: test="pass"
verbose: result="Test Value: text ("50") is "50""
verbose: result="Test Value: text ("100") is "100""
verbose: result="Test Value: text ("0") is "0""
stats: test="pass"
stats: suite="pass"
```

The assertions are logged using "verbose result=…" ; the test and the suite summaries are logged using "stats" type. In the XML log structure stats are expanded by respective pass/fail counters:

```
<stats suite="pass" date="2010-Oct-03 12:18:11.470222">
  <info name="pass">4</info>
  <info name="fail">0</info>
  <info name="total">4</info>
</stats>
```

# 7. Known deficiencies in this release

### 7.1 Opera is not supported

### 7.3 No handling for the browser raised dialogues

### 7.5 Automatic proxy configuration may fail under Windows XP

# 8. FAQ

This chapter is intentionally left blank.

## Appendix 1

OS Agent Keys reference.

| Send Command | Resulting Keypress |
|---|---|
| {!} | ! |
| {#} | # |
| {+} | + |
| {^} | ^ |
| {{} | { |
| {}} | } |
| {SPACE} | SPACE |
| {ENTER} | ENTER key on the main keyboard |
| {ALT} | ALT |
| {BACKSPACE} or {BS} | BACKSPACE |
| {DELETE} or {DEL} | DELETE |
| {UP} | Up arrow |
| {DOWN} | Down arrow |
| {LEFT} | Left arrow |
| {RIGHT} | Right arrow |
| {HOME} | HOME |
| {END} | END |
| {ESCAPE} or {ESC} | ESCAPE |
| {INSERT} or {INS} | INS |
| {PGUP} | PGUP |
| {PGDN} | PGDN |
| {F1} - {F12} | Function keys |
| {TAB} | TAB |
| {PRINTSCREEN} | PRINTSCR |
| {LWIN} | Left Windows key |
| {RWIN} | Right Windows key |
| {NUMLOCK} | NUMLOCK |
| {BREAK} | for Ctrl+Break processing |
| {PAUSE} | PAUSE |
| {CAPSLOCK} | CAPSLOCK |
| {NUMPAD0} - {NUMPAD9} | Numpad digits |
| {NUMPADMULT} | Numpad Multiply |
| {NUMPADADD} | Numpad Add |
| {NUMPADSUB} | Numpad Subtract |
| {NUMPADDIV} | Numpad Divide |
| {NUMPADDOT} | Numpad period |
| {NUMPADENTER} | Enter key on the numpad |

| | |
|---|---|
| {APPSKEY} | Windows App key |
| {LALT} | Left ALT key |
| {RALT} | Right ALT key |
| {LCTRL} | Left CTRL key |
| {RCTRL} | Right CTRL key |
| {LSHIFT} | Left Shift key |
| {RSHIFT} | Right Shift key |
| {SLEEP} | Computer SLEEP key |
| {ALTDOWN} | Holds the ALT key down until {ALTUP} is sent |
| {SHIFTDOWN} | Holds the SHIFT key down until {SHIFTUP} is sent |
| {CTRLDOWN} | Holds the CTRL key down until {CTRLUP} is sent |
| {LWINDOWN} | Holds the left Windows key down until {LWINUP} is sent |
| {RWINDOWN} | Holds the right Windows key down until {RWINUP} is sent |
| {ASC nnnn} | Send the ALT+nnnn key combination |