

Pocket PC TestSuite 2.0 Documentation

Abstract:

This document describes using Pocket PC TestSuite 2.0. It describes installation, user interface and test script language.

Created By:	Vassili Philippov, Shamil Kerimov
Creator Email:	vassili@softspb.com , shamil@softspb.com
Creator Address:	Spb Software House Vozrozhdeniya 4, Saint-Petersburg 198904 Russia Phone: +7 812 324 49 44
Created On:	July 25, 2002
Last Changed On:	August 06, 2002

Table of Contents

Table of Contents	2
Overview.....	3
Main use cases.....	3
Installation	4
System Requirements	4
Testing is low memory conditions.....	5
Memory information.....	6
Keyboard Codes	6
Testing in low CPU conditions	7
Executing CPU stress testing	7
Scripting	8
Scripting Overview.....	8
Run test script	9
Edit test script.....	10
Record commands	11
Using grid.....	12
TCL language.....	13
Introduction.....	13
Getting started.....	13
Comments.....	14
Variables.....	14
Loops	15
Conditions	16
Procedures	16
TestSuite commands.....	17
Simulating	17
ts_click	17
ts_mousedown	17
ts_mouseup.....	17
ts_mousemove.....	17
ts_wait.....	17
ts_key	17
ts_keydown	17
ts_keyup.....	18
ts_text.....	18
ts_run.....	18
ts_cpapplet.....	18
Checking	18
ts_getpixel	18
ts_gettext	18
ts_getfreemem.....	18
ts_checkscreenrect	18
Reporting	19
ts_log	19
ts_messagebox	19
Misc.....	19
ts_savescreenrect.....	19
ts_getos.....	19
ts_getcputype	19
ts_reg_read.....	19
TCL resources	20
Main TCL commands	20
expt	20
for.....	20
if.....	21

Pocket PC TestSuite 2.0 Documentation

incr	21
proc.....	21
return	22
set	22
source	23

Overview

Spb Test Suite allows a high-level automation of testing processes. The main features include:

- **Recording** - you can start record mode, then do something and all your taps will be saved in a script file.
- **Test scripts** - you can create test scripts that simulate user activity. Test script language is based on Tcl plus 22 additional testing specific functions.
- **Simulating user activity** - there are functions for simulating taps (clicks), mouse down, mouse up, keys and delays. You can also simulate tap and hold action.
- **Checking functions** - you can not only simulate user activity but also check results. You can check: color of any point, text in any windows control, compare part of screen with a bitmap, and get free memory available.
- **Reporting** - you can create and write to any file using Tcl functions. We also provide functions for working with logs that are shown in the TestSuite program after the script is finished.
- **Edit scripts on Pocket PC** - you can edit scripts directly on your Pocket PC.
- **Memory stress** - you can test your program in limited free memory conditions.
- **CPU stress** - you can test your program in overloaded CPU conditions (different level of CPU load).

Main use cases

The most typical use of Pocket PC TestSuite is the following

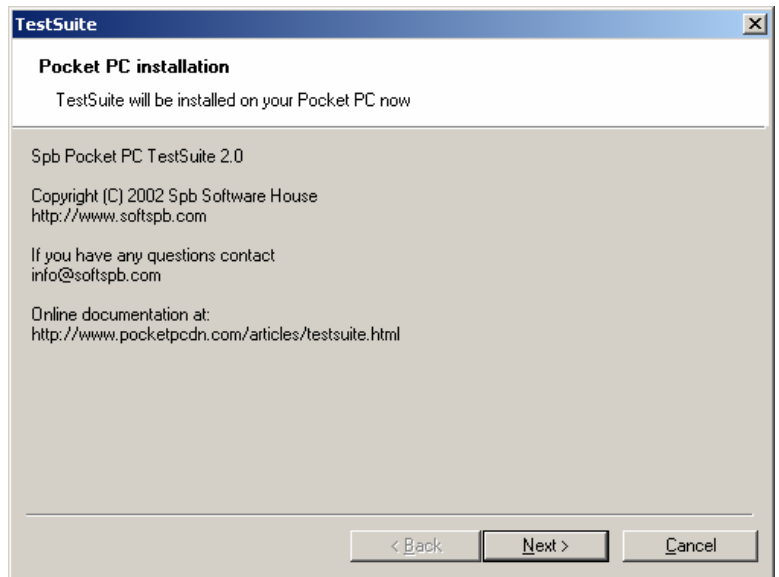
1. Create new test script
2. Work with program you test and record your actions
3. Edit the recorded script to add to checking, loops and reports
4. Create a set of test scripts
5. Run all test scripts on each new build

Installation

To install Pocket PC TestSuite 2.0 on a Pocket PC device connect the device with a desktop computer via ActiveSync and run *testsuite_setup.exe*.

The installation is strain forward an in 3 steps you will have TestSuite installed.

After being installed on Pocket PC you can start TestSuite from **Start > Programs**.



System Requirements

Pocket PC TestSuite 2.0 supports Pocket PC 2000 and Pocket PC 2002 devices with ARM and MIPS CPUs.

Pocket PC TestSuite 2.0 also works on Pocket PC 2002 Emulator that comes with Pocket PC 2002 SDK.

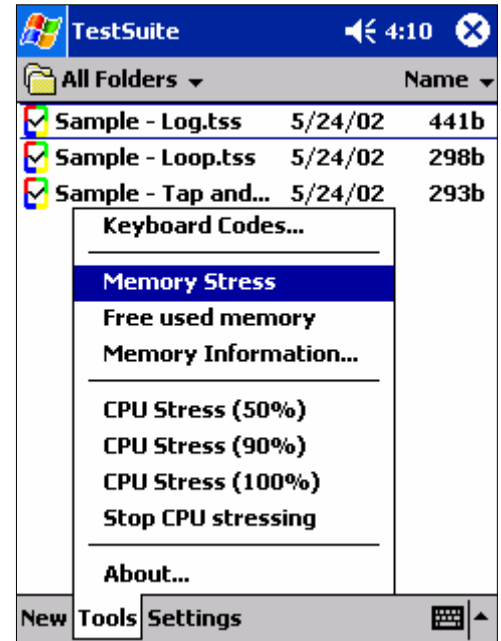
It requires about 1Mb on the device to be installed and run.

Testing is low memory conditions

Various Pocket PC devices come with different memory options. Some of older PDAs have only 16Mb of RAM, while the recently announced Pocket PCs may have up to 128 Mb and even more. But it doesn't matter how much memory the device has, since final users have almost all memory occupied with their pictures, music, documents, etc. It is important for Pocket PC programs to be "low memory friendly" and not to crash in this situation.

Memory Stress tests are executed to investigate the Pocket PC ability to operate in low memory conditions.

When started, TestSuite begins to consume free device memory. It may take a lot of time because of Pocket PC memory management. When free memory runs critical, Pocket PC device reassigns free storage memory as a system memory. TestSuite systematically polls for free system memory, and if there is some, TestSuite consumes it, waits for the system to reassign more memory and so on. When system cannot reassign more free memory, consuming process ends.



Executing memory stress testing

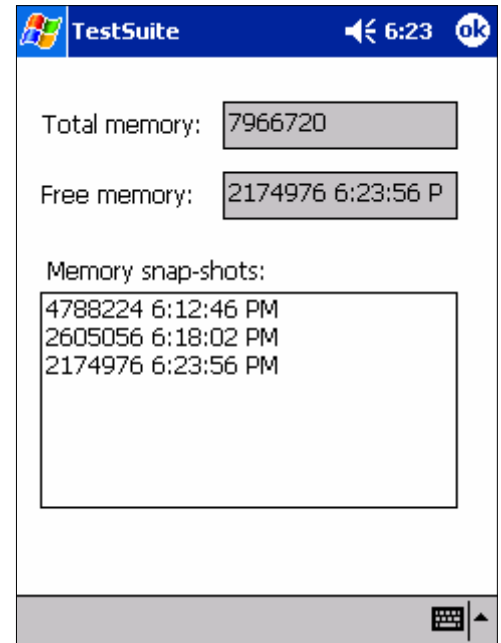
- To start executing this test: select **Tools > Memory Stress**. When memory consuming process ends, you can continue working with your Pocket PC and run programs on it so that you may notice any changes in its behavior.
- To stop the test execution: select **Tools > Free Used Memory**. Your Pocket PC will return to normal operation giving all consumed memory back to the system.

Memory information

You can gain the information about the current device memory status. To do it, select **Tools > Memory Information**. Memory snapshots are saved every time you open memory information, so you can analyze memory usage during application testing process.

A memory snap-shot includes total amount of available memory and time when the snap-shot is done.

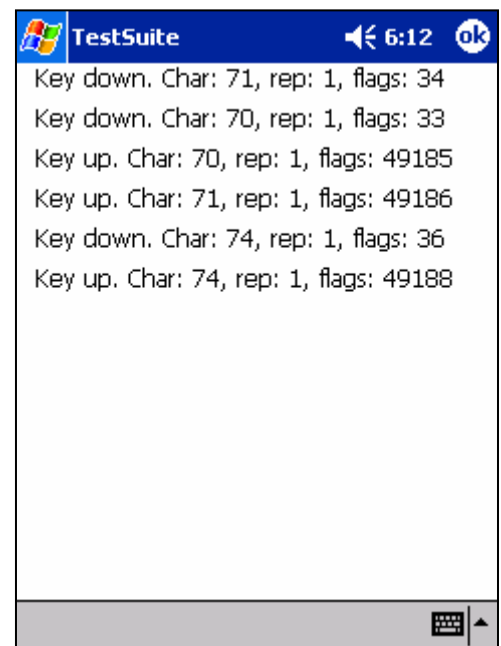
Except that you can use **ts_getfreemem** function in your TCL scripts. This function takes no parameters and returns total amount of available memory in bytes. Note that total amount of available memory does not include storage space. Also note that Pocket PC can dynamically reallocate RAM between memory and storage.



Keyboard Codes

When you use TestSuite commands to simulate key pressing (ts_key, ts_keydowns and ts_key up) you need key codes.

You can use **Tools > Keyboard Codes** menu command to open the "Keyboard Codes" dialog that shows all key commands.



Testing in low CPU conditions

Pocket PC devices cannot have many applications running in the background, as on desktop PC, but there are some additional circumstances. For example, Windows Media Player for Pocket PC can significantly decrease the amount of free processor resources. So any application with time-critical processes must ensure it can operate in overloaded CPU conditions. TestSuite can create three types of internal processes that will occupy 50%, 90% or 100% of the CPU time.

Executing CPU stress testing

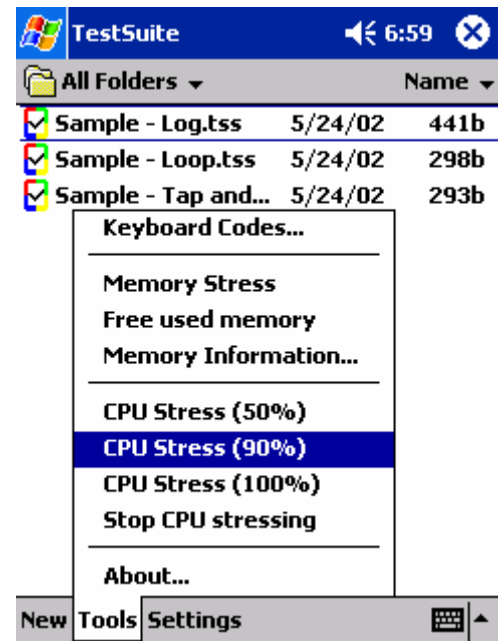
To start executing 50% CPU stress test: select **Tools > CPU Stress (50%)**. It will create an internal CPU loading process, decreasing system performance by half. This is a typical situation to test software as if some other application performing operations in a background.

To start executing 90% CPU stress test: select **Tools > CPU Stress (90%)**. It will create process, consuming almost all CPU resources. This is a typical situation to test software when some intensive operations are performing in a background, such as music or video players, etc. This stress test will ensure that your application will operate normally in those situations.

To start executing 100% CPU stress test: select **Tools > CPU Stress (100%)**. It will create process, consuming all available CPU resources. This is just a synthetic test to ensure your application doesn't crash in these conditions.

To stop the test execution: select **Tools > Stop CPU Stressing**. Your Pocket PC will return to normal operation giving all consumed CPU resources back to the system.

Note that if you have one CPU stressing process working in background (for example started by "CPU Stress (90%)") and start another one (for example started by "CPU Stress (50%)") the previous process will be terminated and only the last one will work. In other words only one CPU stressing process (or none) works at any given moment of time.



Scripting

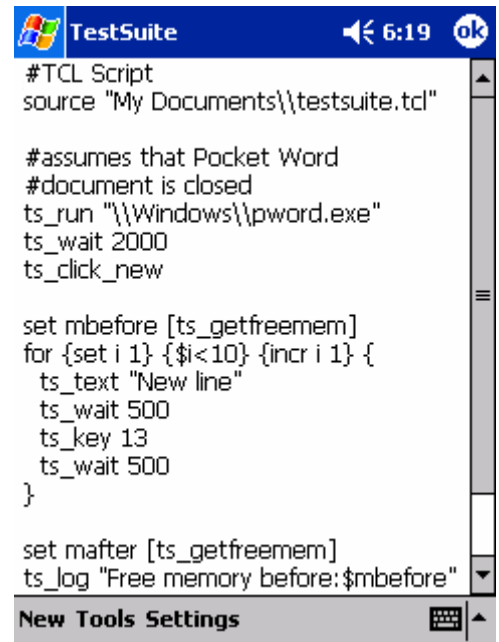
Scripting Overview

Pocket PC TestSuite 2.0 uses Tcl as a base language for test scripts. Tcl is a simple and powerful programming language, but in 95% of cases you do not need its power. Test scripts are rather simple and usually you do not need anything except variables, loops and some functions. You just know that if you need you have this power to create really advanced test scripts.

Test script files must be located in the "My Document" folder.

Test script files have .tss extension and are ASCII text files. You can create them either on your desktop computer and copy to Pocket PC or create/edit directly on Pocket PC using TestSuite editor. We recommend using Pocket PC device + full size hardware keyboard as the most convenient method to create/edit/run test scripts.

You can also record scripts. In that can TestSuite monitors you activity and writes it to the script file as a sequence of simple commands.



```
#TCL Script
source "My Documents\\testsuite.tcl"

#assumes that Pocket Word
#document is closed
ts_run "\\Windows\\pword.exe"
ts_wait 2000
ts_click_new

set mbefore [ts_getfreemem]
for {set i 1} {$i<10} {incr i 1} {
  ts_text "New line"
  ts_wait 500
  ts_key 13
  ts_wait 500
}

set mafter [ts_getfreemem]
ts_log "Free memory before:$mbefore"
```

New Tools Settings

Pocket PC TestSuite 2.0 Documentation

Run test script

When you open Pocket PC TestSuite 2.0 it shows the list of all test script files. The list support standard file operations like:

- Create copy
- Delete
- Select All
- Rename/Move
- Send via Email
- Beam file

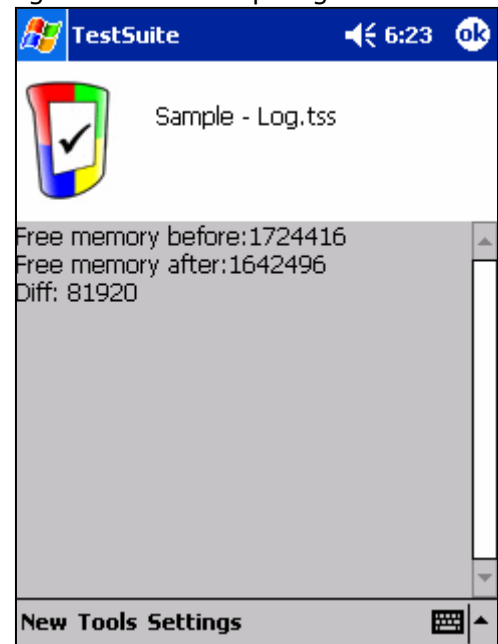
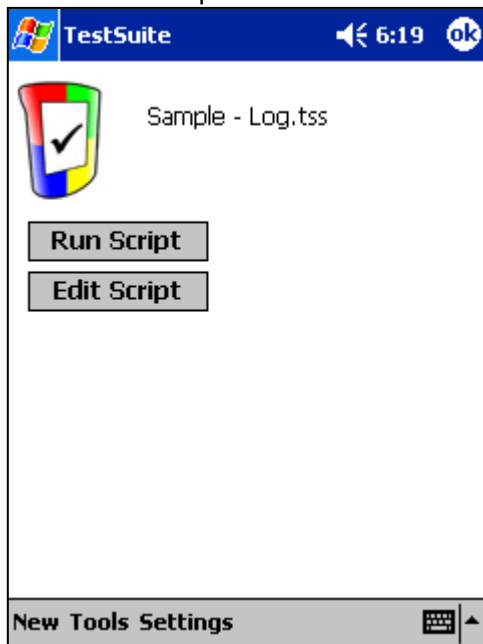
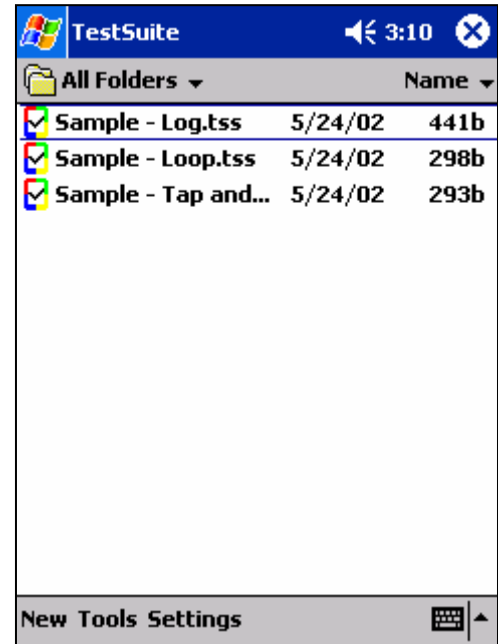
These operations are available in the context menu that can be opened using tap-and-hold.

Click on a test script file to open it.

When the test script file is opened you will see a window that offers you whether to run this script or to edit it. Press the "Run Script" button.

The script will be started. Most probably your script starts another application and does some actions (simulates user activity) in this application.

When the script is finished TestSuite is opened again and the script log is shown.



Pocket PC TestSuite 2.0 Documentation

Edit test script

You can edit test scripts either on your desktop computer or on the Pocket PC device.

If you edit test scripts on your desktop computer you have to copy them to "My Documents" using ActiveSync or some other tools.

If you edit test scripts directly on Pocket PC you can use either TestSuite or any text editor.

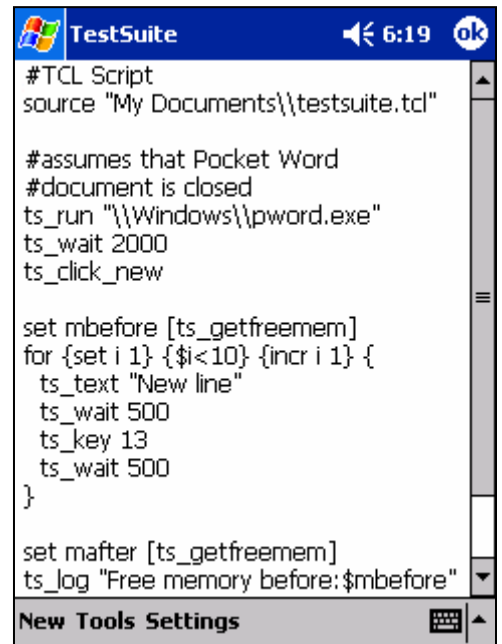
To edit a test script file in TestSuite first open it and then press "Edit Script" button in the window that appears.

Now you see a simple text editor where you can edit your test script.

Except manually editing script text you can add recorded commands (more information in the "Record commands" section).

When you edit test script text it is often helpful to know what are the coordinates of the point you need (for example you want to add a command that simulates tapping in this point). You can use a grid for that. To open the grip choose **Setting > Open Grid** menu item.

To save changes simply press the OK button in the right top corner.



```
#TCL Script
source "My Documents\\testsuite.tcl"

#assumes that Pocket Word
#document is closed
ts_run "\\Windows\\pword.exe"
ts_wait 2000
ts_click_new

set mbefore [ts_getfreemem]
for {set i 1} {$i<10} {incr i 1} {
  ts_text "New line"
  ts_wait 500
  ts_key 13
  ts_wait 500
}

set mafter [ts_getfreemem]
ts_log "Free memory before:$mbefore"
```

Pocket PC TestSuite 2.0 Documentation

Record commands

In most cases you do not need to enter test script text because you can record necessary command.

TestSuite supports recording of taps with saving delay times. Recording of mouse move, tap-and-hold, drag-and-drop operations is not supported.

To start recording choose **Tools > Record Taps** menu item. Then "Record taps" dialog will be opened. Choose record ratio (if you want the test script is played faster than you record it) and press the "Start" button.

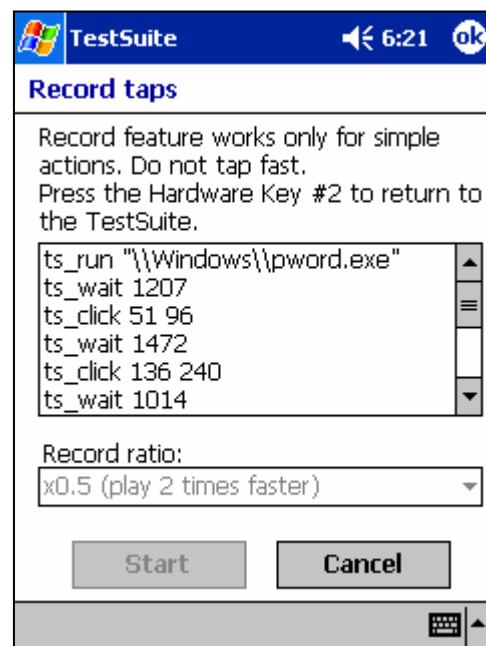
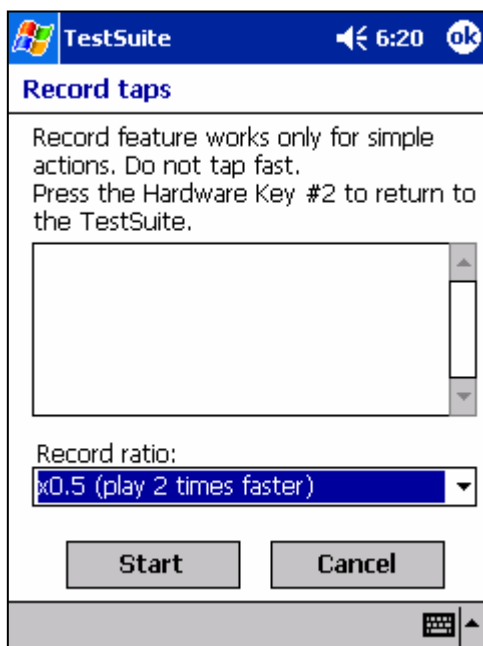
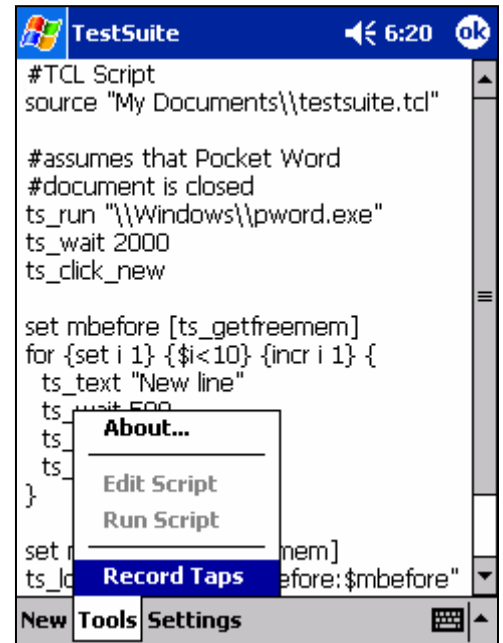
TestSuite detect the first application that is opened after "Start" button is pressed (except Pocket PC shell) and inserts a command that starts this application in the beginning of the recorded macro.

Then TestSuite records all your taps and delays between taps. It is better to avoid small delays between taps because TestSuite can record with mistakes in that situation.

To stop recording press the second hardware key (Calendar). You will return to the TestSuite and will see the "Record taps" dialog again with recorded text. Press the "OK" button (in the right top corner) and the recorded commands will be added to the test script text. Press the "Cancel" button to return to the test script editor with inserting the recorded commands.

Recorded commands always have the following structure

```
ts_run "<Program executable path>"
ts_wait <wait time in milliseconds>
ts_click <x> <y>
ts_wait <wait time in milliseconds>
ts_click <x> <y>
etc
```



Pocket PC TestSuite 2.0 Documentation

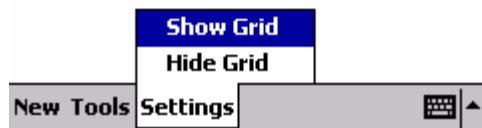
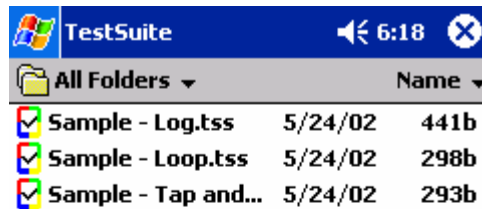
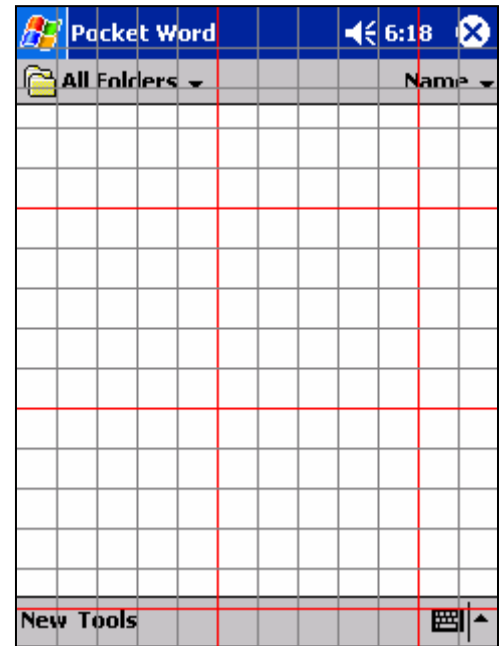
Using grid

Sometimes you edit test script without using command recording. Imagine you want to add a command that simulates user click on the menu item. How to get coordinates of the necessary point?

You can use grid feature that will help you getting coordinates of any point on the screen.

To switch the grip on use the **Settings > Show Grid** menu item. Use the **Settings > Hide Grip** menu item to switch the grid off.

The grid is shown in all Pocket PC applications. Red lines correspond to 100, 200 and 300 pixels.



TCL language

Introduction

TestSuite scripts are based on TCL language. TestSuite adds 21 new commands to standard TCL commands.

TCL language is a string-based command language. It is similar to shell languages like Perl and Csh/Sh/Bash. The only type in TCL is string. Although this language is simple and can be learned in 1-2 days it gives power to write almost any test script you can imagine.

Please keep in mind that you do not need to learn the whole TCL. All you need for creating test scripts are variables, loops, conditions, functions. That's all. This documentation describes these basic TCL commands. If you need more advanced TCL commands for some reason please check online resources that are listed in the "TCL resources" section.

Getting started

It is better to start with an example:

```
for {set i 0} {$i<10} {incr i 1} {  
  ts_click 30 [expr 100+$i]  
}
```

This script simulates 10 user clicks in points: (30, 100), (30, 101), (30, 102), etc.

What do we see here? First of all variables let's discuss each part of this code:

for

It is a TCL command. It takes 4 arguments. Here its arguments are:

- **{set i 0}**
- **{\$i<10}**
- **{incr i 1}**
- **{ts_click \$x [expr \$y+\$i]}**

Everything in TCL are strings. All 4 arguments are also strings. "for" command executes the first string, then executes the 4th and the third strings until the condition of the second string is true.

set i 0

This command assigns "0" to the variable named "i".

{\$i<10}

Note that if you want to take variable value you should use "\$" symbol before variable name.

incr i 1

This command increases content of variable named "i" on 1. Instead of that you can write **set i [expr \$i+1]**.

ts_click 30 [expr 100+\$i]

ts_click is a TestSuite specific command that takes two arguments and simulates user click in this point. Both arguments should contain numbers. Here one argument is simple - 30 and another argument is an expression - **[expr 100+\$i]**.

[expr 100+\$i]

This string executes command **expr** with the given argument and substitutes result of this command. [] brackets mean that everything inside these brackets should be considered as a command that should be executed.

Conclusion

Here are things that we learned with this sample that you should keep in mind:

- To get variable value you should use \$ symbol before variable name. So \$x is the value of variable x
- There is only one type in TCL – a string. If you have to pass a number just pass the string that contains this number
- [] brackets mean that the string inside these brackets should be considered as a TCL command and executed. The result will be substituted on the position on the brackets
- To set variable value you can use **set** command

Comments

To comment a line use "#" symbol. For example in the following code the first two lines are comments:

```
#this is a comment line because it starts with #  
#create 2 variables  
set x 40  
set y 50
```

Variables

The **set** command is used to assign a value to a variable. It takes two arguments: the first is the name of the variable and the second is the value. Variable names can be any length, and case is significant. In fact, you can use any character in a variable name except ":".

Unlike most programming languages, in TCL, when you write the name of a variable, you mean the name – not the value. That is why

```
X + 1
```

will be an error if you meant "add one to x." The reason is that "x" can only be the name of a variable, not its value. To represent the value of a variable, put a dollar sign in front of it, for example, \$x.

Before each line of TCL code is executed, something called variable substitution happens: the line is searched for '\$' signs followed by variable names. Each occurrence, for example, \$X, is replaced with the value of the named variable. Only after this substitution process is complete is the command executed.

It is not necessary to declare TCL variables before you use them.

The interpreter will create the variable when it is first assigned a value. The value of a variable is obtained later with the dollar-sign syntax.

Example

```
#this is a comment line because it starts with #  
#create 2 variables  
set x 40  
set y 50  
  
#click to point (40, 50)  
ts_click $x $y
```

Loops

There are several keywords for loops. The most important are **while** and **for**. These two commands are similar to C the corresponding command but you should use curly brackets around all parts of the command.

For is a looping command, similar in structure to the C for statement. The start, next, and body arguments must be TCL command strings, and test is an expression string. The **for** command first invokes the TCL interpreter to execute start. Then it repeatedly evaluates test as an expression; if the result is non-zero it invokes the TCL interpreter on body, then invokes the TCL interpreter on next, then repeats the loop. The command terminates when test evaluates to 0. If a continue command is invoked within body then any remaining commands in the current execution of body are skipped; processing continues by invoking the TCL interpreter on next, then evaluating test, and so on. If a break command is invoked within body or next, then the **for** command will return immediately. The operation of break and continue are similar to the corresponding statements in C. For returns an empty string.

The while command evaluates test as an expression (in the same way that **expr** evaluates its argument). The value of the expression must a proper boolean value; if it is a true value then body is executed by passing it to the TCL interpreter. Once body has been executed then test is evaluated again, and the process repeats until eventually test evaluates to a false boolean value. Continue commands may be executed inside body to terminate the current iteration of the loop, and break commands may be executed inside body to cause immediate termination of the while command. The while command always returns an empty string.

For example

```
set x 40  
set y 50  
  
#a loop, here we click 10 times increasing y coordinate by 1  
for {set i 0} {$i<10} {incr i 1} {  
  ts_click $x [expr $y+$i]  
}
```

While example

```
set x 40  
set y 50  
set i 0  
  
#a loop, here we click 10 times increasing y coordinate by 1  
while {$i<10} {  
  ts_click $x [expr $y+$i]  
  incr i 1  
}
```

Conditions

Use **if** command for conditional statements. It takes two arguments the first is condition and the second is statement body. Both arguments should be in curly brackets. You can also use **else** keyword.

if *expr1* *body1*

The **if** command evaluates *expr1* as an expression (in the same way that **expr** evaluates its argument). The value of the expression must be a boolean (a numeric value, where 0 is false and anything is true, or a string value such as true or yes for true and false or no for false); if it is true then *body1* is executed by passing it to the TCL interpreter

Example

```
set m [ts_getfreemem]
if {$m<1000000} {
  ts_log "Low memory"
}
```

Procedures

You can create your own procedures (function) with parameters.

TCL uses the **proc** command to define procedures. Once defined, a TCL procedure is used just like any of the built-in TCL commands. The basic syntax to define a procedure is:

proc *name* *arglist* *body*

The first argument is the name of the procedure being defined. The second argument is a list of parameters to the procedure. The third argument is a command body that is one or more TCL commands.

The procedure name is case sensitive, and in fact it can contain any characters. Procedure names and variable names do not conflict with each other. As a convention, this book begins procedure names with uppercase letters and it begins variable names with lowercase letters. Good programming style is important as your TCL scripts get larger.

Example

```
#this function clicks 10 times to the given point
proc click10times {x y} {
  for {set i 0} {$i<10} {incr i 1} {
    ts_click $x $y
    ts_wait 500 #wait 0.5 second
  }
}

click10times 78 45
click10times 210 200
click10times 34 100
click10times 80 80
```

TestSuite commands

Except TCL commands TestSuite adds 21 special commands that allow you to simulate user activity, check results and create logs.

Simulating

Simulating functions simulates user activity like mouse pressing, key pressing, etc.

ts_click

ts_click x y

This command simulates user tap (click) in the given point (x, y). Tap is a sequence of mouse down and mouse up events with small delay between. Tap is the most frequently used operation on Pocket PC and the only operation supported by recording.

ts_mousedown

ts_mousedown x y

This command simulates mouse down event in the given point. You can use this command if you need to simulate operations like drag-and-drop or tap-and-hold.

ts_mouseup

ts_mouseup x y

This command simulates mouse up event in the given point. You can use this command if you need to simulate operations like drag-and-drop or tap-and-hold.

ts_mousemove

ts_mousemove x y

This command simulates mouse move event to the given point. You can use this command if you need to simulate operations like drag-and-drop or tap-and-hold.

ts_wait

ts_wait mills

Delays script for mills milliseconds. Use this command when you need some delay in simulating user input. It is often when programs does not accept too fast use taps and user actions are either lost or handled in wrong way.

The command recording process automatically insert delays between recorded taps.

ts_key

ts_key keycode

This command simulates pressing of the key with the given code. You can use "Keyboard Codes" dialog to find key codes of necessary keys.

Key pressing is a sequence of key down and key up events with small delay between them.

ts_keydown

ts_keydown keycode

This command simulates key down even of the key with the given code. You can use "Keyboard Codes" dialog to find key codes of necessary keys.

ts_keyup**ts_keyup keycode**

This command simulates key up even of the key with the given code. You can use "Keyboard Codes" dialog to find key codes of necessary keys.

ts_text**ts_text yourstring**

This command simulates entering of the given string. Only simple characters like a-z, A-Z, 0-9 are supported. For other characters use either simulating tapping in SIP or **ts_key** commands.

ts_run**ts_run program_path arguments**

This command starts program with the given path and the given arguments (arguments are optional)

ts_cpapplet**ts_cpapplet applet_id page_id**

This command opens Control Panel applet with the given id (page id is optional). You can find numbers of Control Panel applets in the following article:

<http://www.pocketpcdn.com/articles/controlpanel.html>

Checking

These commands are used to check results. For example you know that in the end of your script the program must be in a certain state. You can check this state using the following commands:

ts_getpixel**ts_getpixel x y**

This command returns RGB color of the given point (for example "230 4 34"). Here is code to extract a certain color from this string:

```
set c [ts_getpixel 100 5]  
set r [lindex $c 0]  
set g [lindex $c 1]  
set b [lindex $c 2]
```

ts_gettext**ts_gettext x y**

This command returns text of the control under the given point. You can use this function to check text in edit control, comboboxes, etc.

ts_getfreemem**ts_getfreemem**

This command returns available memory in bytes

ts_checkscreenrect**ts_checkscreenrect x1 y1 x2 y2 bmp_file_name**

This command checks if the screen rectangle of [(x1,y1) - (x2,y2)] is equal to the given bitmap file.

Reporting

These commands are used to check results. For example you know that in the end of your script the program must be in a certain state. You can check this state using the following commands:

ts_log

`ts_log message_string`

This command writes the given string to the log file. When the test script is finished this log file is shown to the user. You can substitute variables as it is shown in the sample bellow:

```
set m [ts_getfreemem]  
set c [ts_getpixel 100 5]  
ts_log "Free memory: $m. Point color: $c"
```

ts_messagebox

`ts_messagebox message_string`

This command shows message box with the given string. Use this function for debug purposes only because it can affect the tested process.

Misc

Other command:

ts_savescreenrect

`ts_savescreenrect x1 y1 x2 y2 bmp_file_name`

This command saves screen rectangle to the given bitmap file. You can use this function to capture screen rectangle and than check that it is not changed after some actions.

ts_getos

`ts_getos`

This command returns device platform. Can be "pocketpc" or "pocketpc2002".

ts_getcputype

`ts_getcputype`

This command returns device CPU type. Can be "arm" or "mips".

ts_reg_read

`ts_reg_read root_key key value`

This command returns string from registry. root_key must be one of: HKEY_LOCAL_MACHINE, HKEY_CURRENT_USER, HKEY_CLASSES_ROOT.

TCL resources

- <http://www.tcl.tk/doc>
Links to many TCL related websites, manuals, tutorials, etc
- <http://www.tcl.tk/man>
Online documentation on TCL language
- <http://www.tcl.tk/doc/styleGuide.pdf>
TCL coding style guide
- <http://www.tcl.tk/scripting/primer.html>
TCL introduction. Learn how to create simple TCL programs
- <http://resource.tcl.tk/resource/doc/manual>
Online documentation on TCL keywords
- <http://www.tcl.tk/scripting/syntax.html>
Introduction to TCL syntax
- <http://resource.tcl.tk/resource/doc/books>
List of books about TCL programming
- <http://www.beedub.com/book/2nd/tclintro.doc.html>
TCL Fundamentals. The first chapter of the TCL tutorial book
- <http://www.mapfree.com/sbf/tcl/book/select/Html/Contents.html>
TCL/TK for Programmers. Basic Syntax

Main TCL commands

Here are main TCL commands. We do not expect you will use other commands. You can find documentation about all TCL commands in the online TCL resources listed in the "TCL resources" section.

expr

expr - Evaluate an expression

SYNOPSIS

expr arg ?arg arg ...?

DESCRIPTION

Concatenates arg's (adding separator spaces between them), evaluates the result as a Tcl expression, and returns the value. The operators permitted in Tcl expressions are a subset of the operators permitted in C expressions, and they have the same meaning and precedence as the corresponding C operators. Expressions almost always yield numeric results (integer or floating-point values). For example, the expression

expr 8.2 + 6 evaluates to 14.2. Tcl expressions differ from C expressions in the way that operands are specified. Also, Tcl expressions support non-numeric operands and string comparisons.

for

for - ``For" loop

SYNOPSIS

for start test next body

DESCRIPTION

For is a looping command, similar in structure to the C for statement. The start, next, and body arguments must be Tcl command strings, and test is an expression string. The for command first invokes the Tcl interpreter to execute start. Then it repeatedly

Pocket PC TestSuite 2.0 Documentation

evaluates test as an expression; if the result is non-zero it invokes the Tcl interpreter on body, then invokes the Tcl interpreter on next, then repeats the loop. The command terminates when test evaluates to 0. If a continue command is invoked within body then any remaining commands in the current execution of body are skipped; processing continues by invoking the Tcl interpreter on next, then evaluating test, and so on. If a break command is invoked within body or next, then the for command will return immediately. The operation of break and continue are similar to the corresponding statements in C. For returns an empty string.

Note: test should almost always be enclosed in braces. If not, variable substitutions will be made before the for command starts executing, which means that variable changes made by the loop body will not be considered in the expression. This is likely to result in an infinite loop. If test is enclosed in braces, variable substitutions are delayed until the expression is evaluated (before each loop iteration), so changes in the variables will be visible. For an example, try the following script with and without the braces around `$x<10`:

```
for {set x 0} {$x<10} {incr x} {  
    puts "x is $x"  
}
```

if

if - Execute scripts conditionally

SYNOPSIS

```
if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?
```

DESCRIPTION

The if command evaluates expr1 as an expression (in the same way that expr evaluates its argument). The value of the expression must be a boolean (a numeric value, where 0 is false and anything is true, or a string value such as true or yes for true and false or no for false); if it is true then body1 is executed by passing it to the Tcl interpreter. Otherwise expr2 is evaluated as an expression and if it is true then body2 is executed, and so on. If none of the expressions evaluates to true then bodyN is executed. The then and else arguments are optional "noise words" to make the command easier to read. There may be any number of elseif clauses, including zero. BodyN may also be omitted as long as else is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no bodyN.

incr

incr - Increment the value of a variable

SYNOPSIS

```
incr varName ?increment?
```

DESCRIPTION

Increments the value stored in the variable whose name is varName. The value of the variable must be an integer. If increment is supplied then its value (which must be an integer) is added to the value of variable varName; otherwise 1 is added to varName. The new value is stored as a decimal string in variable varName and also returned as result.

proc

proc - Create a Tcl procedure

SYNOPSIS

proc name args body

DESCRIPTION

The proc command creates a new Tcl procedure named name, replacing any existing command or procedure there may have been by that name. Whenever the new command is invoked, the contents of body will be executed by the Tcl interpreter. Normally, name is unqualified (does not include the names of any containing namespaces), and the new procedure is created in the current namespace. If name includes any namespace qualifiers, the procedure is created in the specified namespace. Args specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of whose elements specifies one argument. Each argument specifier is also a list with either one or two fields. If there is only a single field in the specifier then it is the name of the argument; if there are two fields, then the first is the argument name and the second is its default value.

When name is invoked a local variable will be created for each of the formal arguments to the procedure; its value will be the value of corresponding argument in the invoking command or the argument's default value. Arguments with default values need not be specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that don't have defaults, and there must not be any extra actual arguments. There is one special case to permit procedures with variable numbers of arguments. If the last formal argument has the name args, then a call to the procedure may contain more actual arguments than the procedure has formals. In this case, all of the actual arguments starting at the one that would be assigned to args are combined into a list (as if the list command had been used); this combined value is assigned to the local variable args.

When body is being executed, variable names normally refer to local variables, which are created automatically when referenced and deleted when the procedure returns. One local variable is automatically created for each of the procedure's arguments. Global variables can only be accessed by invoking the global command or the upvar command. Namespace variables can only be accessed by invoking the variable command or the upvar command.

The proc command returns an empty string. When a procedure is invoked, the procedure's return value is the value specified in a return command. If the procedure doesn't execute an explicit return, then its return value is the value of the last command executed in the procedure's body. If an error occurs while executing the procedure body, then the procedure-as-a-whole will return that same error.

return

return - Return from a procedure

SYNOPSIS

return ?-code code? ?-errorinfo info? ?-errorcode code? ?string?

DESCRIPTION

Return immediately from the current procedure (or top-level command or source command), with string as the return value. If string is not specified then an empty string will be returned as result.

set

set - Read and write variables

SYNOPSIS

set varName ?value?

DESCRIPTION

Pocket PC TestSuite 2.0 Documentation

Returns the value of variable `varName`. If `value` is specified, then set the value of `varName` to `value`, creating a new variable if one doesn't already exist, and return its value. If `varName` contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise `varName` refers to a scalar variable. Normally, `varName` is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If `varName` includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

If no procedure is active, then `varName` refers to a namespace variable (global variable if the current namespace is the global namespace). If a procedure is active, then `varName` refers to a parameter or local variable of the procedure unless the global command was invoked to declare `varName` to be global, or unless a variable command was invoked to declare `varName` to be a namespace variable.

source

`source` - Evaluate a file or resource as a Tcl script

SYNOPSIS

`source fileName`

`source -rsrc resourceName ?fileName?`

`source -rsrcid resourceId ?fileName?`

DESCRIPTION

This command takes the contents of the specified file or resource and passes it to the Tcl interpreter as a text script. The return value from `source` is the return value of the last command executed in the script. If an error occurs in evaluating the contents of the script then the `source` command will return that error. If a `return` command is invoked from within the script then the remainder of the file will be skipped and the `source` command will return normally with the result from the `return` command. The `-rsrc` and `-rsrcid` forms of this command are only available on Macintosh computers. These versions of the command allow you to source a script from a TEXT resource. You may specify what TEXT resource to source by either name or id. By default Tcl searches all open resource files, which include the current application and any loaded C extensions. Alternatively, you may specify the `fileName` where the TEXT resource can be found.