



## Introduction to Tcl/ScI Surpac 6.6

**Copyright © Dassault Systèmes GEOVIA Inc.**

All rights reserved. Dassault Systèmes GEOVIA Inc. publishes this documentation for the sole use of GEOVIA product licensees.

Without written permission, you may not sell, reproduce, store in a retrieval system, or transmit any part of this documentation. For such permission, or to obtain extra copies please contact your local GEOVIA office, or visit [www.3ds.com/GEOVIA](http://www.3ds.com/GEOVIA).

This software and documentation is proprietary to Dassault Systèmes GEOVIA Inc. and, except where expressly provided otherwise, does not form part of any contract. Changes may be made in products or services at any time without notice.

While every precaution has been taken in the preparation of this manual, neither the authors nor GEOVIA assumes responsibility for errors or omissions. Neither will be held liable for any damages caused or alleged to be caused from the use of the information contained herein.

Dassault Systèmes GEOVIA Inc. offers complete 3D software tools that let you create, simulate, publish, and manage your data.

GEOVIA, the GEOVIA logo, combinations thereof, and GEMS, Surpac, Minex, MineSched, Whittle, PCBC, InSite, and Hub are either trademarks or registered trademarks of Dassault Systèmes or its subsidiaries in the US and/or other countries.

**Product**

Surpac™ 6.6

Last modified: Friday, 27 May 2013

# Table of Contents

<b>Introduction to Tcl/ScI Surpac 6.3.....</b>	<b>1</b>
<b>Introduction .....</b>	<b>7</b>
Overview.....	7
Requirements .....	7
Workflow .....	7
<b>Setup for this tutorial .....</b>	<b>8</b>
Setting the work directory.....	8
Task: Set the work directory.....	8
<b>Logicals, command aliases, and hotkeys .....</b>	<b>9</b>
Logicals .....	9
Task: Create a user logical .....	11
Command alias .....	11
Task: Create a command alias .....	13
Keymaps .....	14
Task: Create a hotkey .....	14
<b>Creating menus and toolbars .....</b>	<b>15</b>
Invoking the menu and toolbar editor tool.....	15
Task: Creating a menu from system menu items.....	16
Task: Copy and paste an existing menu .....	16
Creating your own menu items.....	16
Task: Create menu items .....	16
Creating a custom toolbar.....	18
Task: Create a custom toolbar.....	18
Task: Create a custom button to put onto a toolbar.....	18
Summary .....	19
<b>Recording tasks in a Tcl script.....</b>	<b>20</b>
Macro record.....	20
Task: Record a macro .....	20
Macro playback .....	21
Task: Playback the macro .....	21
Task: Run a script without the macro playback form.....	21
Structure of a Tcl macro .....	22
ScIFunction .....	25
Task: Record a script to rotate data .....	25
Using the _action switch .....	26
Task: Record a script to create a DTM.....	27
Task: Record a script to perform inquire point .....	29
Summary .....	29
<b>Some required Tcl basics to get working.....</b>	<b>30</b>
What is Tcl? .....	30
What is ScI? .....	30
Tcl Syntax.....	31
Commands.....	31

set .....	31
puts .....	32
Variables and substitution .....	32
Grouping constructs .....	33
Command substitution .....	33
Summary .....	34
<b>Creating user forms for scripts .....</b>	<b>35</b>
Guido .....	35
Guido command syntax .....	36
Guido command switches .....	37
GuidoForm .....	38
ScICreateGuidoForm .....	38
ScIRun .....	39
Tying Guido together with the ScI form functions .....	40
Task: Create your first form .....	40
GuidoField .....	41
Task: Create a form containing a GuidoField .....	41
GuidoComboBox .....	43
Task: Create a form containing a GuidoComboBox .....	43
GuidoCheckBox .....	45
Task: Create a form containing a GuidoCheckBox .....	45
GuidoRadioButton and GuidoButtonGroupPanel .....	47
Task: Create a form with GuidoRadioButtons and GuidoButtonGroupPanel .....	47
GuidoFileBrowserField .....	49
Task: Create a form containing a GuidoFileBrowserField .....	49
GuidoLabel .....	51
Task: Create a form containing a GuidoLabel .....	51
Dependencies in Guido .....	52
Task: Create a form with a dependency .....	52
Summary .....	53
<b>Automating a recorded script .....</b>	<b>54</b>
Task: Automate a common string maths operation .....	54
Summary .....	57
<b>A better way to select points in graphics .....</b>	<b>58</b>
The ScISelectPoint function .....	58
Task: A simple example to show usage .....	58
Piping point data back into Surpac .....	59
Task: Use ScISelectPoint and then pass xyz values back to Surpac .....	59
Summary .....	60
<b>Useful Tcl commands .....</b>	<b>61</b>
Working with numbers .....	61
The incr command .....	61
Task: Use incr .....	61
The expr command .....	62
Task: Use expr .....	63
The ScIExpr command .....	63

Working with text strings .....	65
string length <text string> .....	65
Task: Use string length .....	65
string index <text string> <charIndex> .....	65
Task: Use string index .....	66
string first <string1> <string2> [<startIndex>] .....	66
string range <text string> <first> <last> .....	67
string trim <text string> .....	67
string tolower <text string> and string toupper <text string> .....	68
Working with the File System .....	69
file copy [-force] <source file name> <target file name> .....	69
Task: Use file copy .....	69
file delete [-force] <file name> .....	69
Task: Use file delete .....	69
file rename [-force] <source file name> <target file name> .....	69
Task: Use file rename .....	70
file exists <file name> .....	70
Task: Use file exists .....	70
file mkdir <directory path> & cd <directory path> .....	70
Task: Use file mkdir .....	70
glob [-nocomplain --] <directory specification> .....	71
Task: Use the glob command .....	71
Summary .....	72
<b>Basic flow control in Tcl .....</b>	<b>73</b>
Boolean expressions .....	73
The if Command .....	75
if {expression} then {command body} .....	75
Task: Use an if then statement .....	75
if {expression} then {commands} else {commands} .....	76
Task: Use an if then else statement .....	76
if {expression} then {commands} elseif {commands} [...] .....	77
Task: Use an if elseif statement .....	77
if {exprsn} {commands} elseif {commands} [...] else {commands} .....	78
Task: Use an if elseif else statement .....	78
while {expression} {commands} .....	79
Task: Use a while statement .....	79
for {start} {expression} {next} {commands} .....	80
Task: Use a for statement .....	80
Changing loop behaviour with continue and break .....	81
Task: Use continue and break in a loop .....	81
Summary .....	81
<b>Manipulating Surpac ranges with Scl .....</b>	<b>82</b>
SclRangeExpand .....	82
SclRangeGetCount .....	82
SclRangeGet .....	83
Putting the range commands together .....	83
Summary .....	85
<b>File input and output – reading and writing files .....</b>	<b>86</b>

Opening and closing files.....	86
open <file name> <access>.....	86
close <file reference variable> .....	86
Reading from a file .....	86
gets <file reference variable> <variable for read data> .....	87
eof <file reference variable> .....	87
Writing to a file.....	87
puts <file reference variable> <text string> .....	87
A file-processing template.....	88
Task: Create a file-processing template .....	88
Summary .....	88
<b>Tcl references .....</b>	<b>89</b>
Surpac help .....	89
Text books .....	89
WWW resources.....	89

# Introduction

## Overview

GEOVIA Surpac provides tools that allow the software to be customised and many repetitive tasks automated into a single control script. These control scripts tell the software to perform a series of functions in an ordered manner. Control scripts are commonly called macros, Tcl scripts, or just scripts.

The Surpac scripting language is an extension to an independent (nonproprietary) scripting language called Tcl. Tcl has been distributed freely since 1990 and is now used in thousands of applications world-wide.

The scripting engine (macro interpreter) contained within Surpac, processes scripts that can contain standard Tcl statements as well as Surpac extension commands called Scl. Tcl is an acronym for Tool command language while Scl is an acronym for Surpac command language.

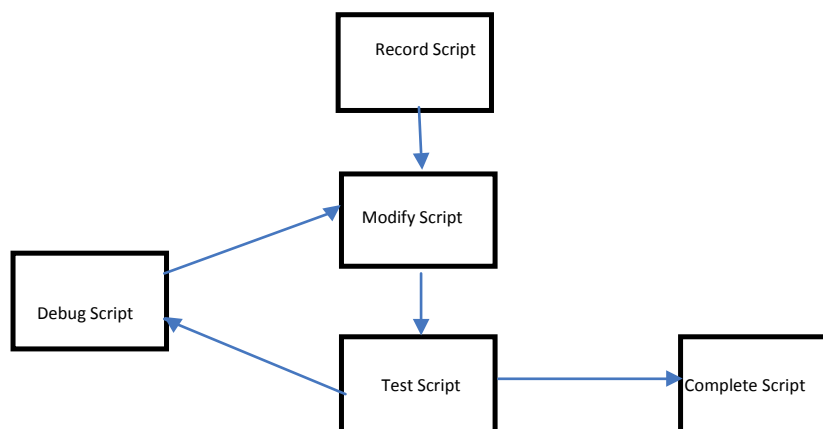
## Requirements


This tutorial assumes that you have a basic knowledge of Surpac. If you are a new Surpac user, you should complete the **Introduction to Surpac** tutorial before beginning this tutorial. This tutorial also assumes that you have basic skills in using a text editor such as ConTEXT or Notepad. The tutorial does not assume you have any previous scripting/programming background.

You will also need:

- Surpac installed on your computer
- a text editor application installed on your computer, preferably ConTEXT

## Workflow



 **Note:** This workflow demonstrates the steps in this tutorial. There are other ways to achieve a result.

# Setup for this tutorial

## Setting the work directory

A work directory is the default directory for saving Surpac files. Files used in this tutorial are stored in the folder `<shared_files> \demo_data\tutorials\tcl_scl`

Where `<shared_files>` is the directory in which the Surpac shared files were installed.

In Windows Vista, Windows 7, and Windows 8, the default path is:

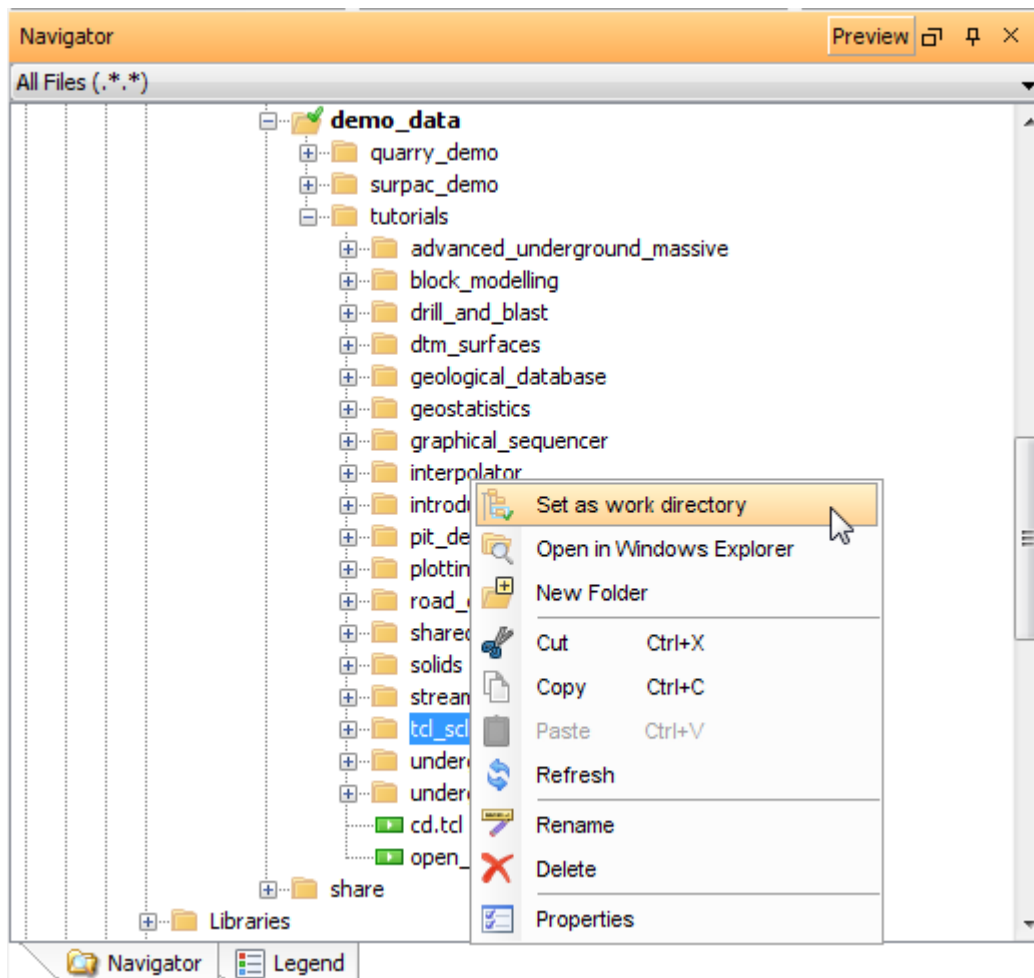
**C:\Users\Public\GEOVIA\Surpac\66\demo\_data\tutorials\tcl\_scl**

In XP, the default path is

**C:\Documents and Settings\Public\GEOVIA\Surpac\66\demo\_data\tutorials\tcl\_scl.**

### Task: Set the work directory

1. In the Navigator, right-click the **tcl\_scl** folder.
2. From the shortcut menu, select **Set as work directory**.



The name of the work directory is displayed in the title bar of the Surpac window.

## Logicals, command aliases, and hotkeys

Creating a macro script to automate some process in Surpac is an easy task to perform. However, having this macro work consistently from any directory can be difficult. Using logicals when you develop macros is crucial to achieving portability across directory structures. The use of command aliases and hotkeys then provides you with a mechanism to quickly access and run common macros.

### Logicals

'Logical' is a Surpac term that refers to the use of a known name to map to a physical directory. The known name, or 'logical', is mapped to a directory on your hard disk at runtime. Using this system your macros do not have to refer to a hard coded directory, which might be different on every installation of Surpac, to access files.

For example, the logical name **SSI\_ETC:** (the known name) can be mapped to a directory named **c:/users/Public/GEOVIA/Surpac/66/share/etc** on one system or **s:/software/Surpac/v6/share/etc** on another. Surpac uses the known name **SSI\_ETC:** to refer to the directory, this means it does not need to know where this directory actually exists because it is mapped at runtime.

The reasons logicals are used are:

- to insulate the software from the file system
- to shorten notation for accessing long directory paths
- to standardise data

Logicals come in three types:

- **System** – defined in the **translate.ssi** file
- **User** – define in the **SSI\_ETC:logicals.ssi** file
- **Personal** – defined in any file and specified to *Surpac* using the **Customise > Default Preferences** function

System logicals are defined by the software installation procedure and are stored into a file that is usually called **translate.ssi**. The system logicals are mandatory. You should never modify a system logical. Below is an example **translate.ssi** file.

```
MACHINE=WIN32
SSI_ETC:      c:\Users\Public\GEOVIA\Surpac\66\share\etc\
SSI_STYLES:   c:\Users\Public\GEOVIA\Surpac\66\share\styles\
SSI_PLOTTING: c:\Users\Public\GEOVIA\Surpac\66\share\plotting\
SSI_PROFILES: c:\Users\Public\GEOVIA\Surpac\66\share\profiles\
SSI_HMF:      c:\Program Files (x86)\GEOVIA\Surpac\66\share\hmf\
SSI_MESSAGES: c:\Program Files (x86)\GEOVIA\Surpac\66\share\msg\
SSI_REFMAN:   c:\Program Files (x86)\GEOVIA\Surpac\66\share\refman\
SSI_RESOURCE: c:\Users\Public\GEOVIA\Surpac\66\share\resource\
SSI_JAVA:     c:\Program Files (x86)\GEOVIA\Surpac\66\share\java\
SSI_BIN:      c:\Program Files (x86)\GEOVIA\Surpac\66\nt_i386\bin
SSI_LIB:      c:\Program Files (x86)\GEOVIA\Surpac\66\nt_i386\lib\
END
```



**Caution:** Do not add user logicals to the translate file. This is because at each new installation of the software this file is created and so you will lose any changes that you had made.

User logicals are optional and you can use them to make finding your macros and data easier. In the same way that system logicals make it easy for Surpac to locate files, user logicals make it easy for you to locate your data and macros. For example, you could define logicals to centralise the storage of your survey pickups, or make a repository used to store all macros so that each user on a site can access them using a consistent name. Using a logical to access macros and data using a standard naming convention, where the physical location of files on a disk does not matter, can be very helpful when you are working in a network environment, especially when users have different physical drive mappings.

User logicals are also a valuable tool when you are designing menu systems and macros. By using logicals, the menu definitions and macros can be kept insulated from the file system because there is no need to worry about physical drive mappings. This way when menus and scripts are transferred to other computers, it is only the logical reference in the **logicals.ssi** file that needs to be changed and not the actual script or menu definitions.

User logicals are defined in a file called **logicals.ssi** that is stored in the **SSI\_ETC:** directory. This is a system logical that maps to the directory where the etc files are stored. The **logicals.ssi** file is optional, if this file does not exist it does not cause an error in Surpac. However, if it does exist then Surpac will load the file and add any defined logicals to the system logical map.

Personal logicals are similar to user logicals except that they can be defined in a file with any name. For example, you can store personal logicals in a file called **mylogicals.txt**. To make Surpac load a personal logicals file you must define it in Surpac using the **alias tab** of the *Settings* form. You can access this form by choosing **Customise > Default preferences**. The format of a personal alias file is exactly the same as the **logicals.ssi** file.

When Surpac starts, it will load logical definitions in the following order:


1. System logicals from **SSI\_ETC:translate.ssi**.
2. User logicals from **SSI\_ETC:logicals.ssi**.
3. Personal logicals from your defined logical file.

If there are duplicate logical definitions then the last one read is the one that takes precedence. It is not an error to define duplicate logicals and you might have good reasons to do so, such as redefining the location of the **SSI\_PLOTTING:** logical in your personal logicals file.

Below is an example of a **logicals.ssi** file or a personal logicals file.

```
MINESOLUTIONS:  c:/minesolutions/  
MS_SPOOLER:    c:/minesolutions/spooler/  
MS_STRING:     c:/minesolutions/string/  
MS_RINGKING:   c:/minesolutions/ringking/  
MY_MACROS:     c:/macros/
```

The structure of the file is quite basic. You first define the logical name, leave some white space, this can be one or more spaces, then define the physical directory mapping followed by a trailing slash.

 **Note:** The trailing slash is mandatory, but it can be either a forward or backslash.

**Tips:**

- Use the full colon at the end of each logical. It provides a consistent end-of-logical marker and makes file paths easier to read when using logicals.
- Forward slashes are recommended over backslashes for your physical paths. Backslashes can sometimes cause obscure problems due to their use as an escaping function in Tcl scripts.
- Make sure your physical directory names are correct; Surpac does not check to see if they actually exist.

**Task: Create a user logical**


1. Open ConTEXT.
2. Choose **File > New**.
3. Determine the path name of your current working directory in Surpac.

 **Note:** The pathname of the current work directory is displayed in the title bar of the Surpac window.

4. Define a logical called MY\_WORK: that will map to the current work directory. Your definition will look similar to the following extract from the **default.ssi** file:

```
MY_WORK: c:/Users/Public/GEOVIA/Surpac/66/demo_data/tutorials/tcl_scl/
```

5. Choose **File > Save As**.
6. In the *Save As* form, navigate to the SSI\_ETC: directory, and type **logicals.ssi** for the file name, then click **Save**.
7. If you are running Surpac, exit and then restart.
8. Locate your new logical called MY\_WORK: in the Navigator.

 **Note:** All logical names are listed in the Surpac Navigator beneath the folder names in alphabetical order.



**Tip:** If you cannot find your MY\_WORK: logical in the list check that you have named the logicals file correctly as described in step six. Make sure you know what the actual pathname for SSI\_ETC is on your computer. You can identify this pathname by finding the SSI\_ETC logical in the Surpac Navigator and expanding it.

## Command alias

A command alias is a system that provides a mechanism for you to rename commands in Surpac to something that you find more suitable or easier to remember. It is often quicker for experienced Surpac users to make use of these short cut alias names than it is to type the full command name or find the function on a menu or toolbar.

Command aliases are defined in a text file with a set syntax. First, the alias name is given, enclosed in double quotes, followed by any amount of white space, and then the actual command name, enclosed in double quotes. An extract from the distributed **short.mst** alias file follows:

"2DG"	"2D GRID"
"2DT"	"2D TRANSFORM"
"3DG"	"3D GRID"
"3DT"	"3D TRANSFORMATION"
"ATP"	"ADD TO PERIOD"
"AR"	"ADD RIG"
"AB"	"APPLY BOUNDARY"
"AII"	"ARC ARC INTERSECT"
"AD"	"AUDIT DATABASE"
"BS"	"BASIC STATISTICS"
"BD"	"BEARING AND DISTANCE"
"BR"	"BENCH REPORT"




**Tip:** When defining alias names that clash, the last one Surpac reads is the one that takes precedence. There is no warning message for a duplicate name so be careful when writing your alias file.

As well as allowing you to rename existing commands, the alias system also allows you to define new command names that you can associate with your Tcl/ScI scripts. For example, if you have defined two scripts, one to import a csv file and load into a database table, and another to export a database table to a csv file, you can create keyboard commands that will run these scripts. The following alias file defines two new commands **import\_csv** and **export\_csv** to run macros stored in the **MY\_WORK:** logical.

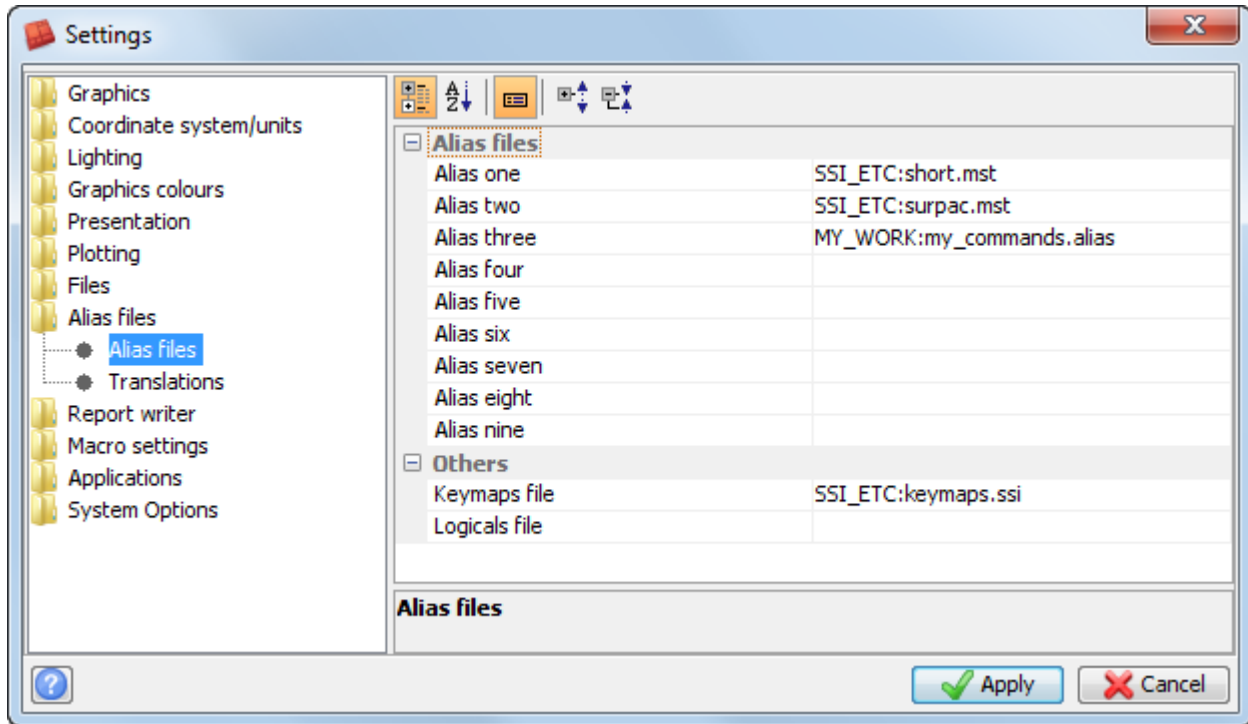
"IMPORT_CSV"	"MACRO:MY_WORK:IMPORT_CSV"
"EXPORT_CSV"	"MACRO:MY_WORK:EXPORT_CSV"

Later you will learn how to attach these two scripts to menus.

 **Note:** The keyword **MACRO:** is used in the path above. This keyword tells Surpac that it is not mapping to an internal Surpac command but instead is going to run a Tcl script that is stored on the hard disk. Notice the use of the logical **MY\_WORK:** which will map to the actual directory path where the macro is stored. This logical is assumed to be previously defined in **logicals.ssi**.

Surpac allows you to specify up to nine alias files.


To define an alias file in Surpac, choose **Customise > Default Preference**, select the **Alias files** folder and enter the name of your alias file including any logical or physical directory name.



### Task: Create a command alias


1. Open ConTEXT.
2. Choose **File > New**.
3. Type the following line.

```
puts "Hello Surpac - this is my first script"
```

 **Note:** Make sure you type exactly as shown, taking care with upper and lower case letters.

4. Choose **File > Save As**.
5. Navigate to the current work directory, and type **hello.tcl** for the file name, then click **Save**.
6. Choose **File > New**.
7. Now create a new command alias definition called HELLO that will run the script created in steps 2 and 3. Your alias definition in the editor should look as follows:

```
"HELLO" "MACRO:MY_WORK:hello.tcl"
```


 **Note:** The **MY\_WORK:** logical is the logical name created in the previous task. Using the logical name as part of the alias definition means that the command will work from any directory. If you do not use a logical the macro will work only if the current work directory is the one the macro is stored in.

8. Choose **File > Save As**.
9. Leave the location as the current work directory, and type **myCommands.txt** for the file name, then click **Save**.

10. In Surpac, choose **Customise > Default preferences**. Then enter the full pathname to the alias file you have created in steps 6 to 9.
11. Exit Surpac and then restart.
12. In the Function choose, type **HELLO**.

You should see the following output in the message window:

```
Hello Surpac - this is my first script
```


 **Tip:** When defining alias commands to run scripts it is safer to make use of logicals as in the example above.

## Keymaps

The keymaps file is a historical file used in earlier versions of Surpac to map the keyboard keys to characters. In Surpac, the only real purpose of the file is to define actions associated with the function keys. You can use the keymaps file to define your own hotkeys to run scripts that you have written.

Using the csv example from the previous discussion of the alias, the following **keymaps.ssi** file extract shows how to define hot keys on the F7 and F8 keys to run the import/export csv scripts.

```
"f7"          FUNCTION    "MACRO:MY_WORK:IMPORT_CSV"
"f8"          FUNCTION    "MACRO:MY_WORK:EXPORT_CSV"
```


 **Tip:** When defining hot keys the key name must be in lowercase characters (for example, f10 not F10).

### Task: Create a hotkey

 **Note:** This task assumes you performed the **Create a command alias** task in the previous section.

1. Open ConTEXT.
2. Choose **File > Open**, and open **SSI\_ETC:keymaps.ssi**.
3. Scroll down and locate the section that defines the F1 through F10 keys.
4. Define a hotkey F11 to run the **hello.tcl** script by inserting a new line as follows:

```
"f11"        FUNCTION    "MACRO:MY_WORK:hello.tcl"
```

 **Note:** The **MY\_WORK:** section is the logical name created in a previous task. Using the logical name as part of the hotkey definition means that the command will work from any directory. Otherwise the hotkey would work only if the macro is saved in the current work directory.

4. If you are running Surpac, exit and then restart.
5. Press **F11**.

You should see the following output in the **message window**:

```
Hello Surpac - this is my first script
```

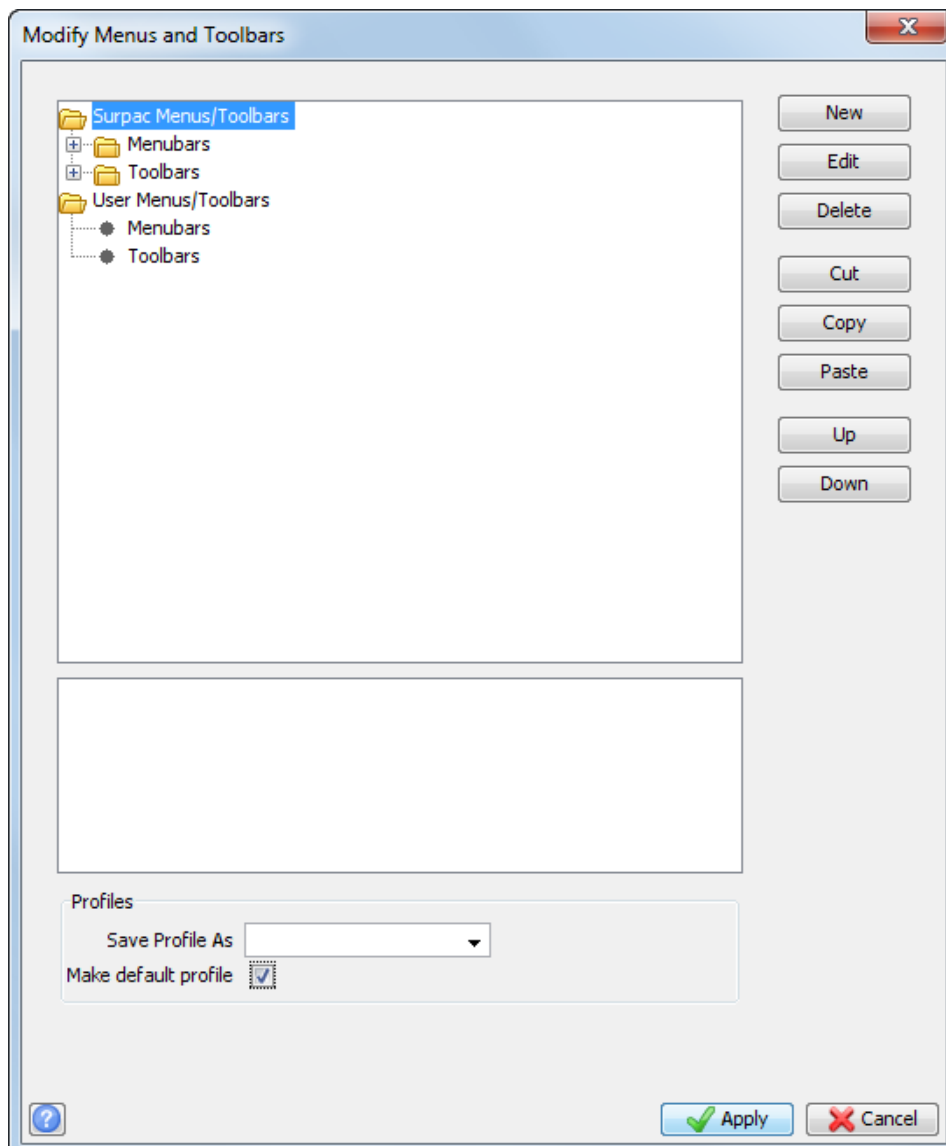
## Creating menus and toolbars

To customise Surpac you can reorganise the menu bars and toolbars to have the functions that you frequently use in the most accessible positions. As you develop macro scripts to automate your daily activities you can also make them available from menus and toolbars. After you work out and implement your preferred layout, you can save these customisations to a user profile and work within this profile.

A profile is a named set of customised menus and toolbars. Surpac distributes a number of predefined profiles and using the technique described in this chapter you can make any number of personalised profiles.

## Invoking the menu and toolbar editor tool

The process of creating menus can be done entirely within Surpac and does not require any manual editing of text files. You access the form on which you modify the menus and toolbars by choosing **Customise > Customise menus/toolbars**.



### Task: Creating a menu from system menu items

1. Choose **Customise > Customise menus/toolbars**.
2. Under **User Menus/Toolbars**, select the **Menubars** item.
3. Click **New**.  
A new menu item is created and named **Menubar\_1**.
4. Click to highlight **Menubar\_1**.
5. Click **Edit**.
6. Type the new name **MyMenubar**.
7. Press ENTER to accept your new name.

#### Notes:

- You can attach any existing menu tree to this new menu using the **copy** and **paste** buttons.
- The New, Edit, Copy, and Paste functions are also available on the shortcut menu, when you right-click in this form.

### Task: Copy and paste an existing menu

1. Find the menu tree (for example, **Display**) that you want to copy and highlight it.
2. Click **Copy**.
3. Highlight **MyMenuBar**.
4. Click **Paste**.

The **Display** menu is added to **MyMenuBar**.

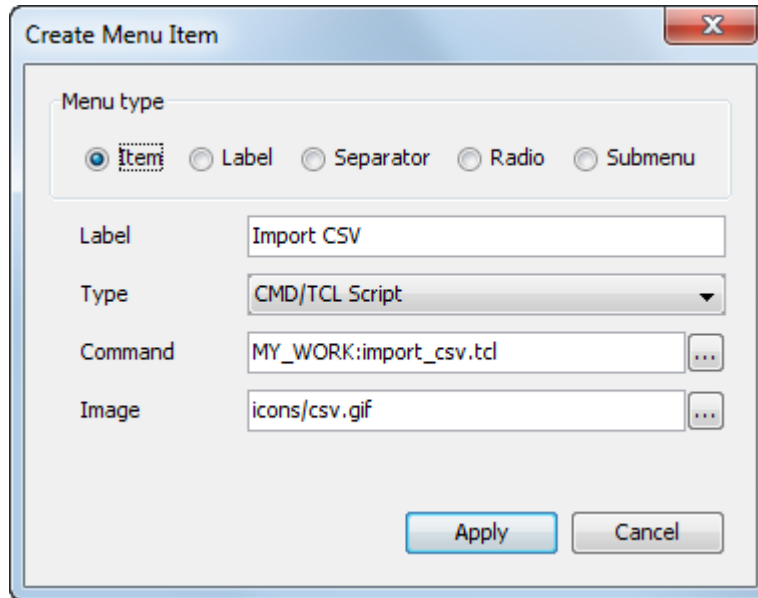
## Creating your own menu items

### Task: Create menu items

1. Click to highlight **MyMenuBar**.
2. Click **New**.  
This creates a new menu called **Menu\_1**.
3. Click to highlight **Menu\_1**.
4. Click **Edit**.
5. Type the new name **MyMenu**.
6. Press ENTER.
7. The new item **MyMenu** is now an empty drop down menu.
8. Click to highlight **MyMenu**.
9. Click **New**.
10. Click the highlight the new menu item.

11. Click **Edit**.
12. Rename the menu to something sensible for the functions you want to put in the menu.
13. Repeat steps 8 to 12 to create as many more menu items as you require.

You can then attach any number of menu items to your new menu, including other sub menus. When you click on a menu branch and click New, the **Create Menu Item** form is displayed.



**Fields on the Create Menu Item form**

Input	Description
Item Category	<p>Item – Invokes a Surpac function or specified Tcl script.</p> <p>Label – Information text to appear as a label to a group of functions on the menu.</p> <p>Separator – An etched bar to separate unrelated menu items.</p> <p>Radio – A special purpose toggle button.</p> <p>Submenu – Links a further menu to the current one.</p>
Label	The text to appear on the menu for this item.
Type	<b>Function</b> runs an internal Surpac function; <b>CMD/TCL Script</b> runs a macro script.
Command	<p>If you chose <b>Function</b> for the <b>Type</b>, the name of the internal Surpac function. If you chose <b>CMD/TCL Script</b> for the <b>Type</b>, the full path name to the Tcl script.</p> <p> <b>Note:</b> You should use logicals <i>not</i> physical paths for the location of the scripts.</p>
Image	A 24x24 pixel gif image that will appear as part of the menu label.

## Creating a custom toolbar

### Task: Create a custom toolbar

1. Under the **User Menus/Toolbars**, select the **Toolbars** item.
2. Click **New**.  
A new toolbar item is created and named **Toolbar\_1**.
3. Click to highlight **Toolbar\_1**.
4. Click **Edit**.
5. Type the new name **MyToolbar**.
6. Press ENTER.

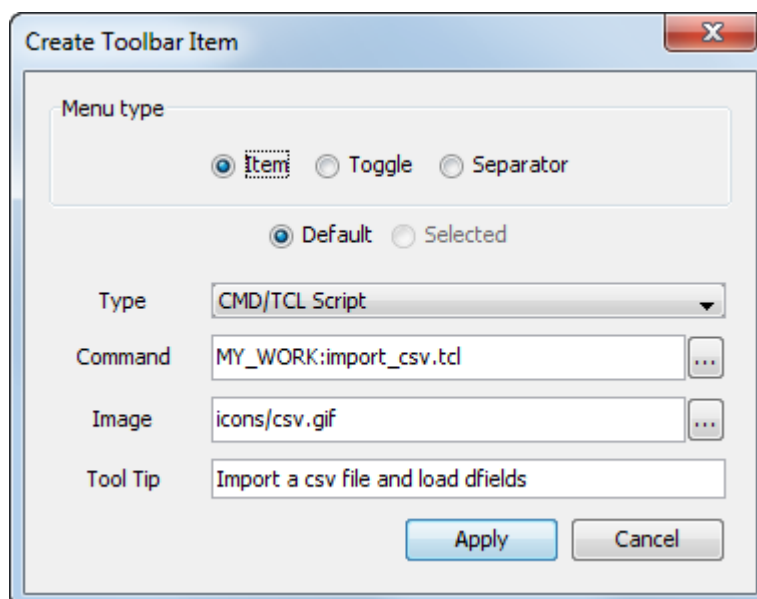
#### Notes:

- You can attach any existing toolbar button to your new toolbar using the **Copy** and **Paste** buttons.
- You can only paste toolbar buttons to the actual toolbar; toolbars are not hierarchical like menus.

### Task: Create a custom button to put onto a toolbar


1. Click to highlight **MyToolBar**.
2. Click **New**.

The **Create Toolbar Item** form is displayed.



3. Fill in the **Create Toolbar Item** form as shown, and click **Apply**.

#### Fields on the *Create Toolbar Item* form

Input	Description
Item Category	<p>Item – Invokes a software function or specified Tcl script.</p> <p>Toggle – A special purpose button that acts as a toggle.</p> <p>Separator – An etched bar to separate buttons into groups.</p>
Type	<b>Function</b> runs an internal Surpac function; <b>CMD/TCL Script</b> runs a macro script.
Command	<p>If you chose <b>Function</b> for the <b>Type</b>, the name of the internal Surpac function. If you chose <b>CMD/TCL Script</b> for the <b>Type</b>, the full path name to the Tcl script.</p> <p> You should use a logicals <b>not</b> a physical paths for the location of the scripts.</p>
Image	A 24x24 pixel gif image that will appear as part of the menu label.
Tool Tip	Some descriptive text that describes the function of the button.



#### Tips:

- Icons that you link in with your buttons using the Image item are optimal at 24X24 pixel icon. This is the standard for Surpac toolbar icons.
- Toolbar buttons are far more effective when there is an associated ToolTip that will be displayed when the mouse pointer is positioned over the button.

## Summary

In this chapter you have been introduced to the toolbar and menubar editor. You were shown how to create both menubars and toolbars. You should now know how to:

- cut and paste items between menus and toolbars
- create menu items
- create toolbar buttons

## Recording tasks in a Tcl script

You will repeat most tasks that you perform in Surpac daily, weekly, or monthly. You can record these tasks to a script so that a series of tasks are recorded for you to play back later. The scripting language is quite flexible, which means you can edit your recorded scripts to make them more suitable to the way you want to use them.

### Macro record

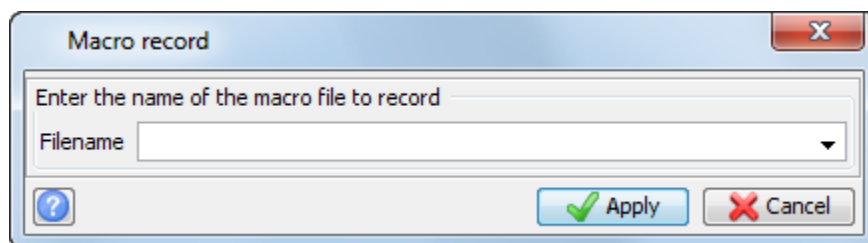
The act of recording a series of steps that you perform in Surpac is a very simple operation. You just need to click a button to start recording a macro, and then click that same button to finish recording. When you start recording a macro you will be prompted to enter a name for the **.tcl** file. You can use a new name to create a new file, or use an existing name to record over an existing script.


#### Task: Record a macro

1. Press **Start/end recording an SCL script** .

 **Note:** You can also start recording a script by pressing F4.

The **Macro Record** form is displayed.



2. Type the name that you want to use for the macro, and click **Apply**.  
A message is written to the **message window** stating that recording has started.
3. Perform the tasks you want to record.
4. Press **Start/end recording an SCL script** .
5. You can also stop recording a script by pressing F5.  
A message is written to the **message window** stating that recording has ended.

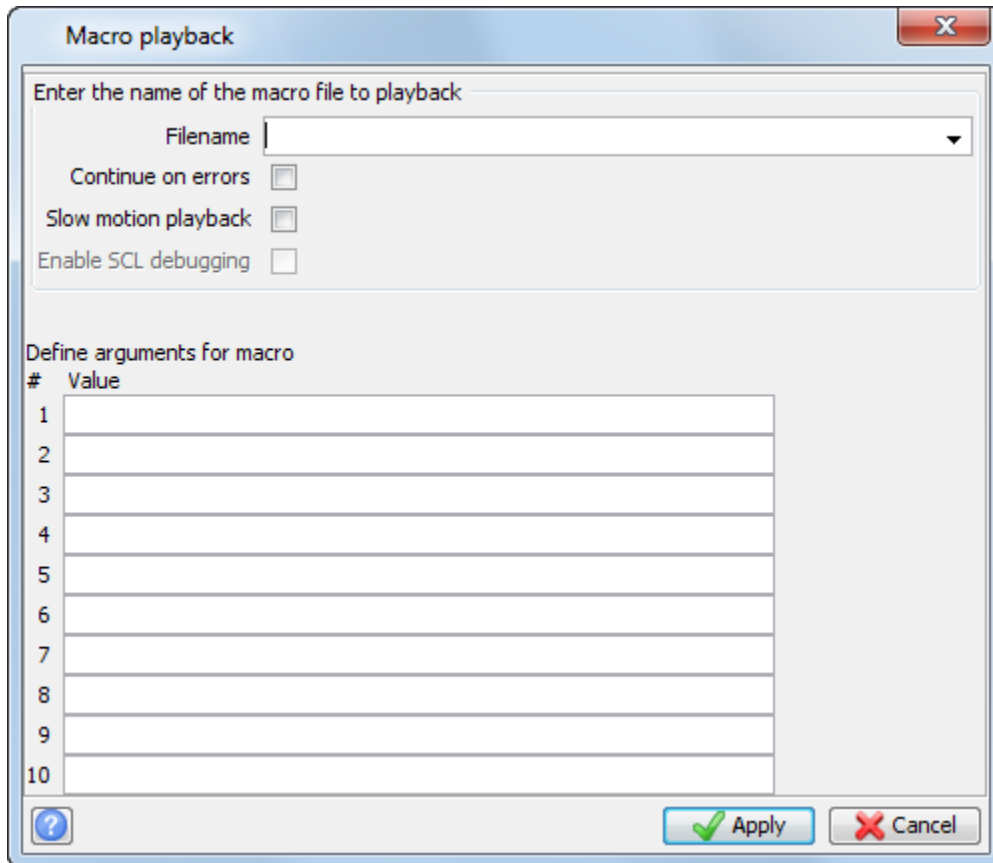
## Macro playback


### Task: Playback the macro

After you have recorded your tasks, you will want to play them back.

1. Click **Playback/abort a macro or SCL script** .

You can also playback a macro by pressing F6.



 **Note:** The ten fields to **Define arguments for macro** are provided for backward compatibility with the superseded V3.2 macro language.

2. Click the Filename drop down to select the macro you want to play, and click **Apply**.

### Task: Run a script without the macro playback form

You can also run a macro by double-clicking the **.tcl** file in the Navigator, or by dragging the **.tcl** file from the Navigator into **Graphics**.

## Structure of a Tcl macro

If you run a macro using the same data, it will always achieve the same result. Usually, you will want your macros to be able to act upon any data set that you provide. To be able to modify a macro to be able to work on any nominated data set you need to have an understanding of the structure of a recorded macro script.

Below is an example of a simple macro that recorded the **CREATE DTM** function in Surpac.

```
#####
#
# Macro Name      : c:/users/public/geovia/surpac/66/demo_data/createdtm.tcl
#
# Version        : Surpac 6.6
#
# Creation Date  :
#
# Description    :
#
#####

set status [ SclFunction "CREATE DTM" {
  frm00126={
    {
      location="DATA:pit"
      break="Y"
      bdyloc=""
      interpolate_new_points="N"
      bdystr="1"
      spotrng=""
      object_num="1"
      obj_name=""
      check_distance="0.0050"
      anyspots="N"
      inout="I"
      brktest="Y"
      apply_boundary="N"
      interpolate_new_points_distance="50"
    }
  }
}]
```

A recorded macro will always contain a header block at the top. This section of the script is non-functional but describes the script name, version of Surpac that created it and the date recorded. There is also a section for a description of the script. ***It is good practice to insert some comments*** about the purpose of, or how to use, the use of the macro. This will make it easier if you or someone else ever needs to modify the macro later.

The header block makes use of the Tcl comment character hash (#). If the first non-blank character on the line is a hash then the Tcl interpreter ignores everything afterwards. You can include comments throughout your scripts by adding lines that start with a hash. It is also good practice to include comments through your script to describe what is happening at different points, especially for large macros.

Following the header block are the actual functions that you performed while recording the macro. The previous example shows how a Surpac function is recorded making use of the Scl extension command *SclFunction*.

The *SclFunction* command takes two parameters, the Surpac function name and then any extra information that is required by that particular function grouped by curly braces. A function that includes interaction with a form will always be recorded with the form identifier (in the case above, *frm00126*) followed by a list of the fields on the form and their entered values.

You don't interact with all Surpac functions using forms. Some functions just require you to click the appropriate button. For example, **ZOOM ALL** as shown in the following.

```
#####
#
# Macro Name      : c:/users/public/geovia/surpac/66/demo_data/2_zoom_all.tcl
#
# Version        : Surpac 6.6
#
# Creation Date  :
#
# Description    :
#####
set status [ SclFunction "ZOOM ALL" {} ]
```

Other functions can be graphical in nature, and need you to input information using the mouse pointer. For example, **IDENTIFY POINT** as shown in the following example.

```
#####
#
# Macro Name      : c:/users/public/geovia/surpac/66/demo_data/3_identify_point.tcl
#
# Version        : Surpac 6.6
#
# Creation Date  :
#
# Description    :
#####
set status [ SclFunction "IDENTIFY POINT" {
  select_point={
    {
      winx="0.042"
      winy="0.205"
      objectx="1735.376"
      objecty="7425.533"
      objectz="65.564"
      snap_projection="off"
    }
  }
}]
```

When you record a graphical function that consists of more than one point selection, the format of the script is a little different to the previous example. The example below shows how the **BEARING AND DISTANCE** function uses a tabular format to record the selected in a macro.

```
#####
#
# Macro Name      : c:/users/public/geovia/surpac/66/demo_data/4_bearing_and_distance.tcl
#
# Version        : Surpac 6.6
#
# Creation Date  :
#
# Description    :
#
#####

set status [ SclFunction "BEARING AND DISTANCE" {
  digitise_point=table { winx winy objectx objecty objectz snap_projection } {
    { "-0.137" "-0.212" "1624.839" "7288.399" "45.462" "off" }
    { "0.090" "0.205" "1765.296" "7425.533" "75.518" "off" }
  }
}]
```

In Surpac 6 and later versions, a new style of function has been introduced that does not contain the traditional form structure. The way that these functions appear in a macro is shown in the following example, using the **OPEN FILE** function.

```
#####
#
# Macro Name      : c:/users/public/geovia/surpac/66/demo_data/5_open_file.tcl
#
# Version        : Surpac 6.6
#
# Creation Date  :
#
# Description    :
#
#####

set status [ SclFunction "OPEN FILE" {
  layer="main graphics layer"
  location="DATA:pit1.str"
  plugin="Surpac String Files"
  Surpac={
    IDRange=""
    range=""
    descriptions="true"
  }
  styles="ssi_styles:styles.ssi"
  replace="true"
  rescale="true"
}]
```

## ScfFunction

The **ScfFunction** command is a Surpac extension that provides the mechanics of invoking a function from within your script. A call to the **ScfFunction** command is recorded automatically by Surpac and it would be unusual that you would ever hand code one.

### Syntax

`ScfFunction FunctionName FunctionParameters`

### Where

*FunctionName* is the name of the Surpac function to execute, enclosed in double quotes

*FunctionParameters* are any further information that the Surpac function requires to run, enclosed in curly braces


### Return Values

Returns either the Scl constant **SCL\_OK**, if the function ran correctly, or **SCL\_ERROR**, if the function failed to run, or did not run correctly

An example of **ScfFunction** as recorded in a Surpac script

```
set status [ ScfFunction "ZOOM ALL" {} ]
```

The function returns a value that will be stored in a variable called **status**. The **status** variable can then be checked for success or failure. A later chapter will provide you with the skills to do check variables to assess success or failure of a script.

 **Note:** When recording macros, movements you perform with the mouse are not recorded. The graphics viewer does not interface with the macro system. This means you cannot record things like rotating data using the mouse. If you want to rotate data in your macro script, you can use the following functions that are available from **View > Rotate image** submenu:

- Orbit up
- Orbit down
- Orbit left
- Orbit right
- Roll left
- Roll right

### Task: Record a script to rotate data

1. Click **Start/end recording an SCL script**.
2. In the **Filename** field type **06\_rotate\_data**, and click **Apply**.
3. Choose **File > Open > String/DTM File**.

4. Click the **Location** drop-down and select **pit1.str**.
5. Rotate the data in **Graphics** to any view you like.
6. Click **Start/end recording an SCL script**.
7. Click **Playback/abort a macro or SCL script**.
8. Click the **Filename** drop-down, select **06\_rotate\_data.tcl**, and then click **Apply**.

Did the data return to the same view as when you recorded the script? Remember you must use the view functions on the **View > Rotate image** submenu and not the mouse to rotate, or otherwise change the view of the data.

## Using the `_action` switch

You can modify your script so that when you play it back you can view and interact with the forms. This allows you to change values on the forms, or to select points in Graphics at appropriate places when playing your macros. To achieve this you will use the `_action` switch.

In the following example the CREATE DTM function has been modified in the macro to display the form when the macro is played.

```
set status [ SclFunction "CREATE DTM" {
  frm00126={
    {
      _action="display"
      location="DATA:pit"
      break="Y"
      bdyloc=""
      interpolate_new_points="N"
      bdystr="1"
      spotrng=""
      id="1"
      check_distance="0.0050"
      anyspots="N"
      swadesc="Y"
      bdyid="0"
      inout="I"
      brktest="Y"
      apply_boundary="N"
      interpolate_new_points_distance="50"
    }
  }
}]
```

The `_action` switch has two possible settings:

- `_action="display"` – This displays the form during playback and allows you to change fields and settings on the form during macro playback.
- `_action="apply"` – This applies the form without displaying it, using the parameters recorded in the macro.

### Notes:

- You must use the double quotes surrounding the keywords to the right of the equal.
- The `_action` switch has no effect on the new style functions, such as **OPEN FILE**.

Another feature of the `SclFunction` command is that all of the parameters for fields on a form that are not defined in your macro will take on the default value for that field when the macro plays. This stops macros from crashing if there is a field that is not set for a function. So if in a later release of Surpac a new field is added to a form, your macro will continue to work, using the default value for the new field.

For example, if your macro for CREATE DTM only included then fields for the location of the string file, as shown in the following, the function would still run on playback with all of the other fields on the form taking on default values.

```
set status [ SclFunction "CREATE DTM" {  
  frm00126={  
    {  
      _action="apply"  
      location="DATA:pit1"  
    }  
  }  
}]
```

### Task: Record a script to create a DTM

1. Click **Start/end recording an SCL script**.
2. In the **Filename** field type **08\_create\_dtm\_task**, and click **Apply**.
3. Choose **Surfaces > DTM File Functions > Create DTM from String file**.
4. Enter the information as shown, and click **Apply**.

5. Click **Start/end recording an SCL script**.
6. Click **Playback/abort a macro or SCL script**.
7. Click the **Filename** drop-down, select **08\_create\_dtm\_task.tcl**, and click **Apply**.

Notice that the *Create DTM* form is not displayed.

8. Right-click on **08\_create\_dtm\_task.tcl** in the Navigator.
9. From the shortcut menu, choose **Edit**.
10. Insert an `_action="display"` switch.
11. Choose **File > Save**.
12. Drag **08\_create\_dtm.tcl** into **Graphics**.

Notice the *Create DTM* form now displays. You could choose to change the string file name that you use when you run the script.

13. Open ConTEXT.
14. Choose **File > Open**.
15. Navigate the working directory, select **08\_create\_dtm\_task.tcl**, and click **Open**.
16. Edit the script so that it appears as in the code snippet on page 26.
17. Choose **File > Save**.

18. In Surpac, click **Playback/abort a macro or SCL script**.
19. Click the **Filename** drop-down, select **08\_create\_dtm\_task.tcl**, and click **Apply**.

Does the script still work?

There are times when you will want to select points graphically when you are running a macro. For example this could be to inspect the properties of a point using the **IDENTIFY POINT** function. The ***\_action="display"*** switch can be used to achieve this as is shown in the example below.

```
set status [ SclFunction "IDENTIFY POINT" {  
  _action="display"  
}]
```

### Task: Record a script to perform inquire point

1. Open **pit1.str**.
2. Click **Start/end recording an SCL script**.
3. In the Filename field type **09\_inquire\_point**, and click **Apply**.
4. Use the menu option **Inquire > Point Properties** and then select any number of points in **Graphics**.
5. Click **Start/end recording an SCL script**.
6. Drag **09\_inquire\_point.tcl** into **Graphics**.  
Notice that it always selects the same points.
7. Open ConTEXT.
8. Choose **File > Open**.
9. Navigate to the working directory, select **09\_inquire\_point.tcl**, and click **Open**.
10. Insert the ***\_action="display"*** switch as shown in the example above.
11. Choose **File > Save**.
12. In Surpac, drag **09\_inquire\_point.tcl** into **Graphics**.

You are now prompted to select the points when the script is run.

## Summary

In this chapter you learnt how to record macro scripts and how to play them back. You were introduced to the **SclFunction** command, and you were shown how to use the ***\_action*** switch to make your macros more interactive and useful.

## Some required Tcl basics to get working

The Tcl language has a simple syntax. After you understand some basic concepts you will be able to use the language to produce good quality solutions. The concepts that you will need to master are;

- storing information in variables
- grouping
- variable and command substitution

### What is Tcl?

Tcl is both a scripting language and an interpreter. As a scripting language, Tcl provides you with a standard syntax, variables, flow control, and procedures. It is an extensible language that allows third parties to add their own functionality.

As an interpreter, Tcl can be embedded into an application such as Surpac. Many other software vendors worldwide have also adopted Tcl as an embedded scripting language. The function of the interpreter is to process scripts line by line to produce a result.

Tcl is public domain software that is freely available and can be used in commercial applications. It is available on a number of platforms including *Windows NT/98/2000/Vista/7*, *Unix varieties*, and *Macintosh*.

### What is Scl?

Scl commands are extensions to the Tcl interpreter that give you great flexibility in your scripts to manipulate both the externals and internals of *Surpac*. Using Scl commands, you can:

- load string and DTM data into graphics layers
- insert, modify, and delete point and triangle data within graphics layers
- create graphics layers and control viewport selection
- manipulate active layers
- create user forms using the Guido toolkit components

These tools allow you to create new functions within Surpac that appear as though they are intrinsic Surpac functions.

Although there are a large number of Scl commands at your disposal, you will usually use only a small subset of them. To inspect the Scl command documentation, open the Surpac help on the Table of Contents, and choose Surpac Concepts, then Macros, then SCL.

## Tcl Syntax

The language syntax used to write Tcl is quite simple. A generic statement in Tcl would be;


### Syntax

```
command parameter1 parameter2 ...
```

### Where

*Command* is the name of a Tcl internal command or defined procedure

*Parameter1*, and all subsequent parameters, are optional

 **Note:** The parameters that you define for a command will depend on the requirements of the command.

Commands and parameters are separated by white space. The end of line signifies the end of the command's arguments. If the command and arguments run over two or more lines, use a backslash character (\) to continue the command over onto the next line. You can insert comments in Tcl by using the hash character (#) as the first non-white-space character on a line.

## Commands

Everything that causes the Tcl interpreter to do something is a command. Each new line in a Tcl script begins with a command. Commands can have optional parameters; the number and type of parameters depend on the specified command.

Two commands that you will use frequently are the **set** and **put** commands.

### set

The set command is used to create variables in your script. A variable is simply a named location in memory used to store information. Think of a variable as a bucket that contains something (information) with a label on the front (its name) that you can use to identify it from other buckets. The syntax for set is as follows.

### Syntax

```
set varName value
```

### Where

*varName* is the name you want to call the variable

*value* is the value assigned to the variable

### Examples

```
set tableName assay
set dip 49.5
set sectionName temp_sec
```

## puts

The **puts** command is used to print messages to the Surpac **message window**. It takes one argument, which is the message to print.

### Syntax

```
puts message
```

### Where

*message* is the text and/or variables to print to the **message window**

### Example

```
puts "Hello Surpac user"  
puts "The DB table being reported is $tableName"
```

## Variables and substitution

Variables are named containers (buckets) that hold information that is required at some stage when the script is being processed. The action of accessing the information stored in the variable is called variable substitution or variable de-referencing.

Variables are defined by the set command as follows:

```
set sectionNumber "6800"
```

To substitute a variable for the information contained within it, prefix the name of the variable with the dollar character (\$). The following example outputs the of the variable sectionNumber to the **message window**

```
puts "The current section is $sectionNumber"
```

Executing this command in Surpac results in the following text being displayed in the **message window**:

```
The current section is 6800
```

## Grouping constructs

The previous example demonstrated the first grouping construct. Double quotation marks (" ") are used to group string tokens together and have them treated as one autonomous object.

The puts command (as used in the example) expects one argument, the text to write to the **message window**. If the double quotes were not used in the example, puts would be given six parameters instead of one, because white space is used to separate parameters.

The other grouping construct used in Tcl is curly braces ({ }). The difference between curly braces and double quotes as grouping constructs is that double quotes permit variable substitution, and further command processing, and curly braces do not.

If we change the puts example to use curly braces then

```
set sectionNumber "6800"  
puts {The current section is $sectionNumber}
```

Will result in the following being written in the **message window**.

```
The current section is $sectionNumber
```

## Command substitution


Only the first token encountered on a line is treated as a Tcl command. Sometimes you will have to execute another Tcl command as part of an original Tcl command. This will become obvious in examples later in this tutorial.

To execute a command as part of another command you enclose the secondary command in square brackets ([ ]). Whenever you want to get the result of a command and use it as a parameter to another command you use square brackets.

```
set drillholeId "rc0001"  
puts "Drill Hole ID = [string toupper $drillholeid]"
```

This will result in the message below being written to the **message window**.

```
Drill Hole ID = RC0001
```


 **Note:** The "string toupper" command used in the previous example will be explained in a later topic in this tutorial.

Another example:

```
set number 100  
set new_number [expr $number * 10]  
puts "The new number is $new_number"
```

This will result in the message below being written to the **message window**.

```
The new number is 1000
```

 **Note:** The “expr” command used in the previous example will be explained in a later topic in this tutorial.

## Summary

In this chapter you learnt some Tcl basics in the areas of:

- syntax
- variables
- variable substitution
- grouping using double quotes and curly braces
- command substitution.

You should now be able to record scripts and make useful adjustments to the scripts.

## Creating user forms for scripts

The Guido (Graphical User Interface Design Objects) toolkit provides you with a set of commands to interact with a user and get them to enter input to use in Tcl scripts. The general Guido object is the **GuidoForm** that can contain a number of other Guido widget and container objects. There are two Scl commands that you must use in order to create and display a Guido form in Surpac. These are **SclCreateGuidoForm** and **SclRun**.

### Guido

Guido commands are Surpac extensions to the Tcl interpreter.

There are two types of GUIDO commands, Guido container objects and Guido widgets. A Guido container object consists of other Guido objects. A container can contain other containers, widgets, or both other containers and widgets. A container object is used to organise the layout and appearance of your form. A Guido widget is an object that is used to retrieve information from a user.

#### Common Guido container and widget objects

Guido Containers	Guido Widgets
GuidoForm	GuidoField
GuidoPanel	GuidoComboBox
GuidoScrollPane	GuidoCheckBox
GuidoTable	GuidoRadioButton
GuidoTabbedPane	GuidoButtonGroupPanel
	GuidoFileBrowserField

## Guido command syntax

Each Guido command has the same syntax. The general command syntax for Guido is:


### **Syntax**

```
GuidoCommand name commandBody
```


### **Where**

*GuidoCommand* is the name of the actual Guido command

*name* is the name you have assigned to the Guido command (like a variable name)

 **Note:** This name will be used to reference the command later to retrieve data that is entered into it.

*commandBody* is enclosed in braces and will contain all switches used to define the look of the Guido widget or container on the form

 **Note:** If the command is a Guido container, the other Guido commands will also be contained within these braces.

## Guido command switches

Every Guido command allows one or more switches to define default behaviours for the object. Fortunately most of the switches are common to all objects so you don't have to learn a different set for each object.

### Common switches for Guido widget commands

Switch	Description
-label	Specify a label for the object
-width	Specify the width of the field in characters
-height	Specify the character height of the object
-default	Specify a default value for the widget
-format	Place a data type validation on the widget which can be: <b>none</b> – no formatting <b>double</b> – allow a number with a fraction <b>integer</b> – allow a whole number only (no fractions) <b>range</b> – a valid Surpac range (That is, 1,10,2 or 1000,2000,50) <b>string_field</b> – x y z d1 d2 d3 ... <b>decimal_angle</b> – 0 to 360 degrees <b>dms_angle</b> – ddd.mmss. For example, 45.2059. <b>colour</b> – colour name or rgb format (For example, r=1.0,g=0.5,b=0.0)
-high_bound	Specify the highest value that the widget is to allow to be entered
-low_bound	Specify the lowest value that the widget is to allow to be entered
-null	Specify if the widget is allowed to have no value. Can be <i>true</i> or <i>false</i>
-translate	Specify if any translation is to occur on character entered in the widget. Can be <i>none</i>   <i>upper</i>   <i>lower</i>   <i>mixed</i>
-input	Determines if the widget can accept input. Can be <i>true</i> or <i>false</i>
-dependency	Make the object depend on the value of another widget

 **Note:** See the Surpac Help for a full list of all the switches available and examples of how to use them.

## GuidoForm

The **GuidoForm** command is used to define a form that will contain any number of containers and widgets.

### Switches particular to GuidoForms

Switch	Description
-default_buttons	Place the standard Apply, Cancel and Help buttons on the form
-defaults_key <name>	Interface into the defaults management system to remember entries made on the form

### Example

```
GuidoForm form {
  -label "An Example Form"
  -default_buttons
  -defaults_key abc_mines
  # insert other guido objects here
}
```

## ScICreateGuidoForm

The ScICreateGuidoForm function will turn a Guido form definition into a computer representation in memory, so that it is ready to be displayed on the screen.

### Syntax

```
ScICreateGuidoForm FormHandle FormDefinition Body
```

### Where

*FormHandle* is a named variable that you use to make reference to the created form

*FormDefinition* is usually the variable name that contains the actual form definition

For example, \$myForm. It can also be the name of an external file that contains the definition prefixed with the at character (@). For example, @section\_form.tcl.

*Body* is the command body

It can contain statements to set default values for form fields. It is usually a pair of curly braces {}.

### Return Values

*SCL\_OK* if the function creates the form object with no error

*SCL\_ERROR* if something goes wrong

### Example

An example that will create a form from the definition stored in a Tcl variable named *sectionForm*.

```
ScICreateGuidoForm form_handle $section_form {}
```

An example that will create a form from the definition stored within a file called *gradecontrol.frm*.

```
SclCreateGuidoForm form_handle @gradecontrol.frm {}
```

## SclRun

The **SclRun** command will display and execute a Guido form that has previously been created with the **SclCreateGuidoFormCommand**.

### Syntax

```
FormHandle SclRun Body
```

### Where

*FormHandle* is the named variable that you use to reference the form created with the **SclCreateGuidoForm** command

*Body* is the command body can contain extra statements

It is usually enclosed by a pair of curly braces {}.

### Return Values

*SCL\_OK* if the function creates the form object with no error

*SCL\_ERROR* if something goes wrong

### Example

```
$section_form SclRun {}
```

The **SclRun** command creates a Tcl variable for each Guido widget defined on the form except if the user clicks the **Cancel** button. The variables created use the same name as was defined for each Guido widget in the form definition.

A special variable with the name *\_status* is created by the **SclRun** command with a value that represents the button that was clicked to dismiss the form. For example, if the **Apply** button is clicked, *\_status* will have the value *apply*. Whereas, if the **Cancel** button is clicked, *\_status* will have the value *cancel*.

The following example code segment allows you to test to see if the user clicked **Cancel**, and, if so, to exit the script gracefully.

```
$grade_control_form SclRun {}  
if {"$_status" == "cancel"} {  
    return  
}
```

 **Note:** The “if” statement and “return” statement are discussed in topics later in this tutorial.

## Typing Guido together with the Scl form functions


The previous section discussed the **GuidoForm**, **SclCreateGuidoForm**, and **SclRun** functions. The following task will show you how to combine these functions together to display a form on the screen.

### Task: Create your first form

1. Open ConTEXT.
2. Save a new, blank, file as **10\_create\_first\_form.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set form_def {
    GuidoForm form {
        -label "My First Form"
        -default_buttons
        -defaults_key "my_form"
    }
}

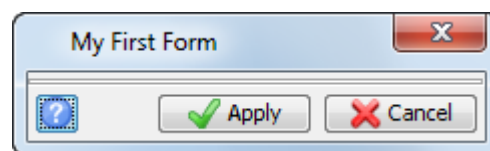
SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}
```

 **Note:** The actual form definition is stored into a Tcl variable called **form\_def**. This is the usual way to define a form, but you can also define a form in an external file, and then specify the external file to **SclCreateGuidoForm** command using the at character (@). The following code sample shows what you would write to create a form using a definition stored in a file called **my\_form.tcl**.

```
SclCreateGuidoForm form_handle @my_form.tcl {}
```

4. Choose **File > Save**.
5. In Surpac, drag **10\_create\_first\_fom.tcl** into **Graphics**.

You will see the following.



## GuidoField

A **GuidoField** is the most common Guido widget you will use. It will accept input that the user will type at the keyboard. It can contain all the various switches described previously to setup labels, field widths, and validation criteria.

### Task: Create a form containing a GuidoField

1. Open ConTEXT.
2. Save a new, blank, file as **11\_create\_form\_with\_guido\_field.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

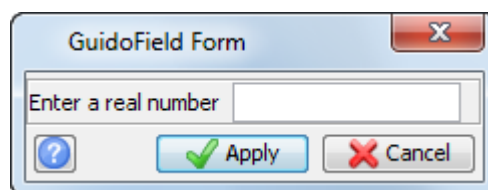
```
set form_def {
  GuidoForm form {
    -label "GuidoField Form"
    -default_buttons

    GuidoField number {
      -label "Enter a real number"
      -format double
      -width 10
      -low_bound 0
      -high_bound 100
      -null false
    }
  }
}
SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}

puts "The number is $number"
```

4. Choose **File > Save**.
5. In Surpac, drag **11\_create\_form\_with\_guido\_field.tcl** into **Graphics**.

You will see the following.



6. Type a number outside the range 0 to 100, and click **Apply**.

The following message is written to the **message window**:

```
Warning: Validation of field 'Enter a real number' failed
Warning: Value is greater than high bound 100.0
```

The form will not apply if the number does not fit the validation criteria that were set in the definition.

7. Type a number inside the range 0 and 100, and click **Apply**.

Because of the final line of the script using the 'puts' command the message, `The number is 80`, is written to the **message window**.

## GuidoComboBox

A **GuidoComboBox** is similar to a field. The difference is that you can provide a set of values that the user can choose from, as a drop-down list. GuidoComboBoxes can be exclusive or non-exclusive. Exclusive combo boxes allow only an entry from the list, non-exclusive combo boxes allow the user to enter any value, so long as it meets the validation criteria.

Any of the switches described previously can be used on a **GuidoComboBox**. There are also some special switches that can only be used with combo boxes.

### Switches that are available only for combo boxes

Switch	Description
-exclusive	If true, only values from the list can be selected. If false, any value will only allow values to be selected from the list otherwise any text can be entered
-value_in	Specify a list of space separated values. If a particular value contains spaces then enclose it in double quotes

### Task: Create a form containing a GuidoComboBox

1. Open ConTEXT.
2. Save a new, blank, file as **12\_create\_form\_with\_combo\_box.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set form_def {
  GuidoForm form {
    -label "GuidoComboBox Form"
    -default_buttons

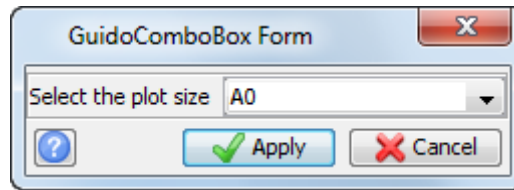
    GuidoComboBox plot_size {
      -label "Select the plot size"
      -width 10
      -null false
      -exclusive true
      -value_in A0 A1 A2 A3 A4
    }
  }
}

SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}

puts "The plot size is $plot_size"
```

4. Choose **File > Save**.
5. In Surpac, drag **12\_create\_form\_with\_combo\_box.tcl** into **Graphics**.

You will see the following.



6. Type a value that was not in the list and press **Apply**.

Because you set `exclusive` to `true`, the form cannot be applied with a value other than those available in the drop-down list.

## GuidoCheckBox

A **GuidoCheckBox** is a widget that is used to select one of two possible states, for example, yes/no, on/off up/down. When the check box is selected it will return the text true and when cleared it will return the text false. These default values can be changed using switches.

### Switches you can use with GuidoCheckBox

Switch	Description
-caption	Specify some text to appear with the checkbox component
-caption_placement	Specify where to place the caption, this can be left or right
-selected_value	Specify a value to return when selected
-unselected_value	Specify a value to return when cleared

### Task: Create a form containing a GuidoCheckBox

1. Open ConTEXT.
2. Save a new, blank, file as **13\_create\_form\_with\_check\_box.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set form_def {
  GuidoForm form {
    -label "GuidoCheckBox Form"
    -default_buttons

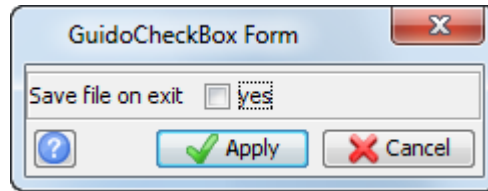
    GuidoCheckBox save_on_exit {
      -label "Save file on exit"
      -height 1
      -caption yes
      -selected_value yes
      -unselected_value no
    }
  }
}

SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}

puts "Save on exit $save_on_exit"
```


4. Choose **File > Save**.
5. In Surpac, drag **13\_create\_form\_with\_check\_box.tcl**.

You will see the following.



6. Select the check box, and click **Apply**.

`Save on exit Yes` is written to the **message window**.

 **Tip:** If you specify **-height 1** when you define your check box, the box will line up correctly with its label.

## GuidoRadioButton and GuidoButtonGroupPanel

A **GuidoRadioButton** is similar to a check box. It is also used to select one of two possible states. When the radio button is selected it will return the text true, and when cleared it will return the text false. These default values can be changed using switches.

### Switches you can use with GuidoRadioButton and GuidoButtonGroupPanel

Switch	Description
-caption	Specify text to appear with the checkbox
-caption_placement	Specify position of the caption (left or right)
-selected_value	Specify a value to return when selected
-unselected_value	Specify a value to return when cleared

The real power of radio buttons is that they can be combined together within a **GuidoButtonGroupPanel**. When they are combined only one of the buttons defined within the group maybe selected at any time, like an old fashioned car radio.

A button group panel is unusual in that it is both a container and a widget. Its value becomes that of the selected radio button within it after the form is applied.

### Task: Create a form with GuidoRadioButtons and GuidoButtonGroupPanel

1. Open ConTEXT.
2. Save a new, blank, file as **14\_create\_form\_radio\_buttons\_and\_panel.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

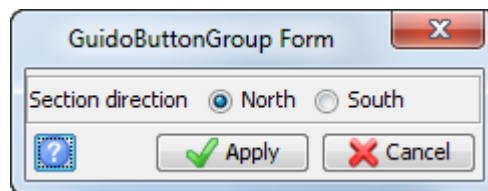
```
set form_def {
  GuidoForm form {
    -label "GuidoButtonGroup Form"
    -default_buttons


    GuidoButtonGroupPanel section_direction {
      -label "Section direction"

      GuidoRadioButton button1 {
        -caption "North"
        -height 1
        -selected_value "north"
      }
      GuidoRadioButton button2 {
        -caption "South"
        -height 1
        -selected_value "south"
      }
    }
  }
}
```

```
}  
  
SclCreateGuidoForm form_handle $form_def {}  
$form_handle SclRun {}  
  
puts "Section direction $section_direction"
```


4. Choose **File > Save**.
5. In Surpac, drag **14\_create\_form\_with\_radio\_buttons\_and\_panel.tcl** into **Graphics**.  
You will see the following.



 You can select only one button or the other, not both.

6. Select **North**, and click **Apply**.

Section direction north is written to the message window. The reference in the macro to GuidoButtonGroup means the message will return with north or south depending on which button you select.

 **Tip:** If you specify **-height 1** when you define your radio button, the button will line up correctly with its label.

## GuidoFileBrowserField

A **GuidoFileBrowserField** is a special purpose widget that you can use to ask the user to type or select a valid filename. The field looks like a combo box, and a user can type in a filename for the working directory. However, if the user clicks on the drop-down arrow, a file chooser widget is invoked, which allows the user to navigate within folders on their computer to select a file. There are a number of switches associated with file browsers.

### Switches you can use with file browsers

Switch	Description
-file_access	Specify the access of the selected file, this can be read, write, or exist
-file_mask	Specify a file mask, for example, *.* or *.tcl
-start_dir	Specify a default directory to begin the browser in
-multiple_selection	Specify that you wish to select more than one filename in the browser
-link	Link this field to another field that will contain the id part of the filename when using location and id
-extension	Specify if you want the file extension trimmed off the return name, this can be true or false

### Task: Create a form containing a GuidoFileBrowserField

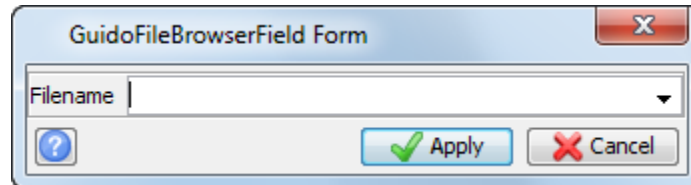
1. Open ConTEXT.
2. Save a new, blank, file as **15\_create\_form\_with\_file\_browser\_field.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set form_def {
  GuidoForm form {
    -label "GuidoFileBrowserField Form"
    -default_buttons

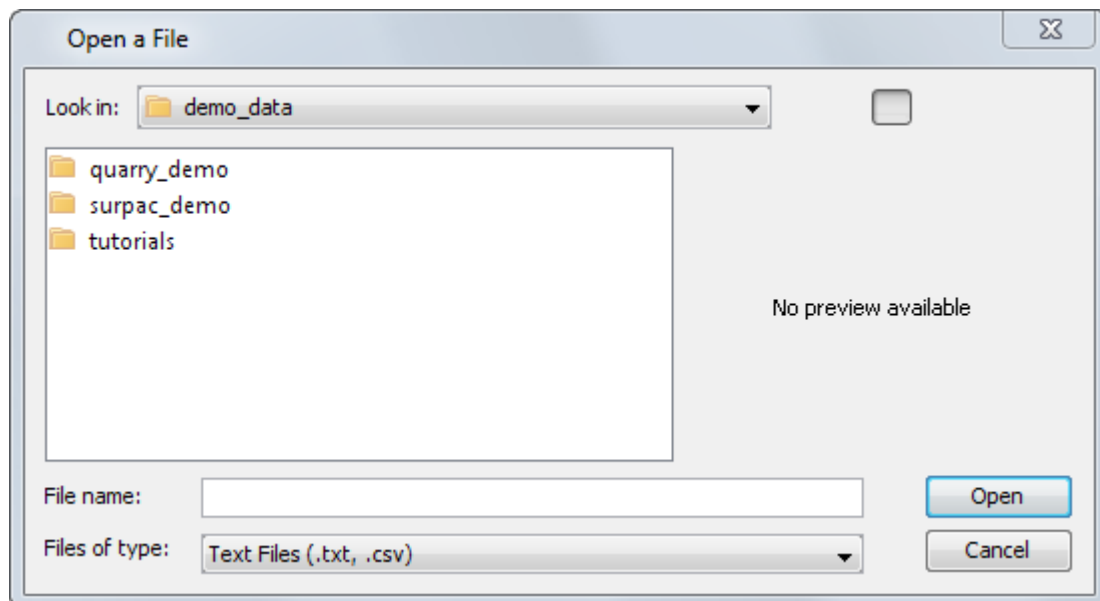
    GuidoFileBrowserField csv_filename {
      -label "Filename"
      -width 25
      -file_mask "*.csv *.txt"
      -file_access read
      -format none
      -translate lower
      -null false
    }
  }
}
SclCreateGuidoForm form_handle $form_def {}
```

```
$form_handle TclRun {}  
puts "File name is $csv_filename"
```

4. Choose **File > Save**.
5. In Surpac, drag **15\_create\_form\_with\_file\_browser\_field.tcl** into **Graphics**.
6. You will see the following.



7. Select the drop-down arrow to display the file browser as shown.



## GuidoLabel

A **GuidoLabel** is not a true widget but it does relate to the widgets, and it is appropriate to discuss how to use it along with how to use widgets. A **GuidoLabel** is used to put a text or informational messages onto your form to help users understand your macro.

### Task: Create a form containing a GuidoLabel

1. Open ConTEXT.
2. Save a new, blank, file as **16\_create\_form\_with\_guidolabel.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

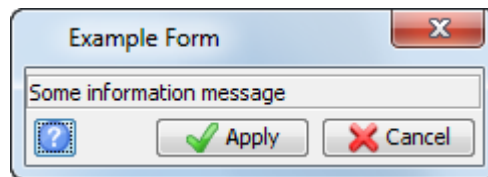
```
set form_def {
  GuidoForm form {
    -label "Example Form"
    -default_buttons

    GuidoLabel info {
      -label "Some information message"
    }
  }
}

SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}
```

4. Choose File > Save.
5. In Surpac, drag **16\_create\_form\_with\_guidolabel.tcl** into **Graphics**.

You will see the following.



## Dependencies in Guido

Any Guido object can contain a dependency switch. A dependency will make a particular widget or container dependent upon the value of another widget on the form. The effect of a dependency is to turn the dependant object on or off. When Guido Widgets are turned off (that is, their dependency condition is false) they appear greyed out. When **GuidoPanels** are turned off they become invisible as does everything that is contained within them.

### Task: Create a form with a dependency

1. Open ConTEXT.
2. Save a new, blank, file as **17\_create\_form\_with\_dependency.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set form_def {
  GuidoForm form {
    -label "Dependency Form"
    -default_buttons

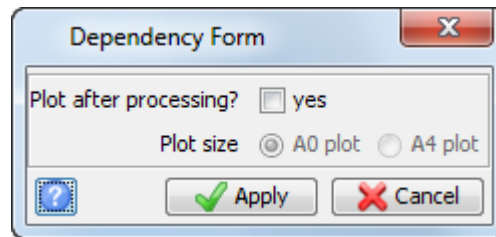
    GuidoCheckBox plotting {
      -label "Plot after processing?"
      -caption yes
      -selected_value yes
      -unselected_value no
    }
    GuidoButtonGroupPanel plotSize {
      -label "Plot size"


      GuidoRadioButton radioButton1 {
        -caption "A0 plot"
        -height 1
        -selected_value A0
        -dependency {"[${plotting getCurrentValue}]" == "yes"}
      }
      GuidoRadioButton radioButton2 {
        -caption "A4 plot"
        -height 1
        -selected_value A4
        -dependency {"[${plotting getCurrentValue}]" == "yes"}
      }
    }
  }
}

SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}
puts "Plotting $plotting ; Plot size $plotSize"
```

4. Choose **File > Save**.
5. In Surpac, drag **17\_create\_form\_with\_dependency.tcl** into **Graphics**.

You will see the following.



 **Note:** When you select the check box the radio buttons for plot size will appear as normal, when you clear the check box the radio buttons are greyed out.

## Summary

In this chapter you were introduced to some of the Guido widgets that you can use to get input from the user. You were shown how to turn your form definition into a computer representation using the `SclCreateGuidoForm` function, and how to interact with the user using the `SclRun` function. You were shown how to access the inputs made by users on a form.

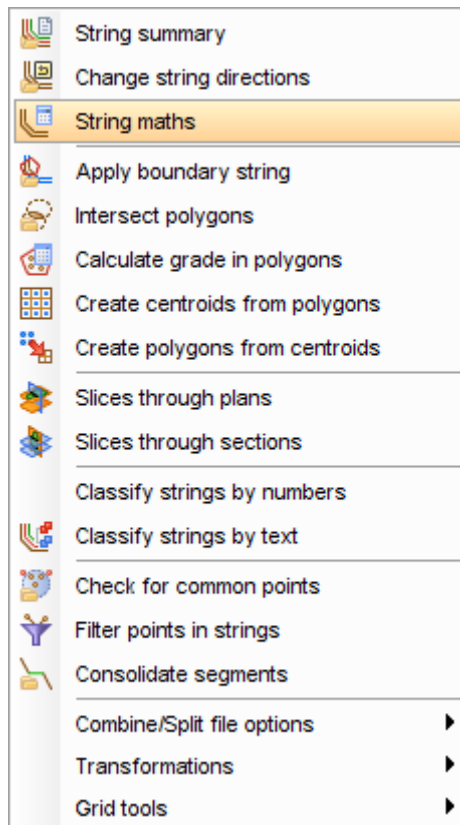
## Automating a recorded script

After you are familiar with recording scripts and creating forms for users you will start to have ideas about many areas that you could automate to make the software easier to use. By working through this chapter you will apply the skills learnt earlier in the tutorial to automate a common string maths operation.

### Task: Automate a common string maths operation

The aim of this example is to automate the common task of using the string math function to adjust the co-ordinates in a string file by a constant value. After you get the scrip to work correctly, to perform the task, it is an easy process to modify it to perform any task in string maths.

1. In Surpac, click **Start/end recording an SCL script**.
2. In **Filename**, type **18\_adjust\_coordinates**.
3. Choose **File tools > String maths**.



4. Enter the information as shown, and click **Apply**.

String maths

Define the files to be processed

Location: DATA:Fit

ID range: 1

Define the files to be created

Location:

	String range	Constraint	Field	=	Expression
1	all		x	=	x+100
2			y	=	y+100
3			z	=	z+50
4				=	
5				=	
6				=	
7				=	
8				=	
9				=	
10				=	

Apply Cancel

5. Click **Start/end recording an SCL script**.
6. Open ConTEXT.
7. Choose **File > Open**, and select **18\_adjust\_coordinates.tcl**.
8. Edit the script so that it will produce the following form.

Adjust Coordinates

Input location: [dropdown]

Id: [text]

Output location: [text]

Adjust X: 0

Adjust Y: 0

Adjust Z: 0

Apply Cancel

**Note:** You will need to use variables for the fields:

- in\_location
- in\_id
- out\_location
- adjust\_x
- adjust\_y
- adjust\_z

All fields should have appropriate validation settings for *format* and *nulls*. The input location is a file browser field linked to the id. Inserting the following form definition code at line 1 of the script, prior to the recorded section, will produce this form.

```
set adjustCoordsForm {
  GuidoForm adjustCoordsForm {
    -label "Adjust Coordinates"
    -default_buttons


    GuidoFileBrowserField in_location {
      -label "Input location"
      -width 20
      -format none
      -null false
      -file_mask "*.str"
      -link in_id
    }
    GuidoField in_id {
      -label "Id"
      -format float
      -null false
    }
    GuidoField out_location {
      -label "Output location"
      -width 20
      -format none
      -null false
    }
    GuidoField adjust_x {
      -label "Adjust X"
      -format float
      -null false
      -default 0
    }
    GuidoField adjust_y {
      -label "Adjust Y"
      -format float
      -null false
      -default 0
    }
    GuidoField adjust_z {
      -label "Adjust Z"
      -format float
      -null false
      -default 0
    }
  }
}
# create and display the form
SclCreateGuidoForm adjustCoordsFormHandle $adjustCoordsForm
{}
$adjustCoordsFormHandle SclRun {}
```

The final step will remove the hard coded values as were saved into the **ScfFunction** when you recorded the macro, and replace them with the appropriate variables from the **GuidoForm**.

9. Edit the macro again and modify the **ScfFunction** statement to look like the following.

```
# perform the conversion by calling string maths with the
form inputs
set status [ ScfFunction "STR MATHS" {
  frm00700={
    {
      file_loc="$in_location"
      file_id="$in_id"
      result_loc="$out_location"
      main=table { str_range constraint field expr } {
        { "all" "" "x" "x+$adjust_x" }
        { "" "" "y" "y+$adjust_y" }
        { "" "" "z" "z+$adjust_z" }
      }
    }
  }
}
```

10. Choose **File > Save**.
11. In Surpac, drag **19\_adjust\_coordinates.tcl** into **Graphics**, and make sure it runs as expected.

 **Note:** To test the completed macro, open **pit1.str** in **Graphics**, then drag the macro into **Graphics**, and then drag the output file into **Graphics**. You should see that the new file is 'shifted' by the values that you entered on your form.

## Summary

This application shows how easy it is to automate a mundane task that is performed regularly. It is far easier to use the script than it is to remember the syntax required by the string maths function. You could easily modify this script to perform another string maths function that you perform frequently.

## A better way to select points in graphics

Earlier in this tutorial, you recorded graphical selections in a macro. This demonstrated how cumbersome it can be to use the *select point table* of the *SclFunction* command. This topic will introduce you to the *SclSelectPoint* extension function. You can use this function to select a point in Graphics and have information about that point returned into Tcl variables.

### The SclSelectPoint function


The *SclSelectPoint* function allows you to select any data point in the graphics window and have a variety of information returned into variables for use in your script. You can retrieve the xyz values of the point, its description fields, and the numbers of the string and segment it is contained in.

#### Syntax

```
SclSelectPoint handle prompt layer stringNo segNo pointNo x y z desc
```

#### Where

*Handle* is a returned reference variable that you can use to modify the selected point using other Scl extension functions

 **Note:** How to modify a selected point using other functions is not covered in this tutorial.

*Prompt* is any text that you want to appear on the screen to ask the user to select something

*Layer* is a variable into which the layer name of the selected point is returned

*stringNo* is a variable into which the string number of the selected point is returned

*segNo* is a variable into which the segment number of the selected point is returned

*pntNo* is a variable into which the point number of the selected point is returned

*x y z* are variables into which the x, y, and z coordinates of the selected point are returned

*desc* is a variable into which the entire description field of the selected point is returned


#### Return values

Returns either the Scl constant *SCL\_OK* or *SCL\_ERROR* depending on whether the script runs successfully or fails

### Task: A simple example to show usage

1. Open ConTEXT.
2. Save a new, blank, file as **20\_simple\_select\_point.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set status [SclSelectPoint point "Select a point" \  
            layer str_no seg_no pnt_no x y z desc]  
puts "Layer=$layer, string=$str_no, segment=$seg_no, point=$pnt_no"  
puts "X=$x, Y=$y, Z=$z"
```

 **Note:** The backslash ( \ ) used on line 1 of the example is a continuation character to extend line 1 onto line 2. In your example you can combine lines 1 and 2 onto the same line.

1. Choose **File > Save**.
2. In Surpac, open **pit1.str** in **Graphics**.
3. Drag **20\_simple\_select\_point.tcl** into **Graphics**.
4. Select a point.

## Piping point data back into Surpac


You have already used a simple method that allowed you to select points in Graphics when playing back a recorded macro by using the `_action="display"` switch. While this works, it can be cumbersome because certain graphical functions sit in a fast loop until the user presses ESC. By using the **ScISelectPoint** function you can better handle macros that use functions that loop until they are cancelled.

### Task: Use ScISelectPoint and then pass xyz values back to Surpac

1. In Surpac, open **pit1.str** in **Graphics**.
2. Click **Start/end recording an SCL script**.
3. In **Filename**, type **21\_bearing\_and\_distance**, and click **Apply**.
4. Choose **Inquire > Bearing and distance between two points**.
5. Choose two points in **Graphics**.
6. Press ESC.
7. Click **Start/end recording an SCL script**.
8. Drag **21\_bearing\_and\_distance.tcl** into **Graphics**.

It performs the same bearing and distance calculation, then exits.

9. Open ConTEXT.
10. Choose **File > Open**, and select **21\_bearing\_and\_distance.tcl**.
11. Edit the script to use the `_action="display"` switch.


 **Note:** If you cannot remember where to insert this switch refer back to the earlier chapter of this tutorial.

12. In Surpac, in the Navigator, right-click the working folder and click **Refresh**.
13. Drag **21\_bearing\_and\_distance.tcl** into **Graphics**.

You can now select any points for the bearing and distance calculation, and the script returns to the selection cycle instead of exiting.

14. In ConTEXT, modify the script so that it looks like the following.

```
SclSelectPoint pnt1 "Select the setup point" \  
    layer1 str_no1 seg_no1 pnt_no1 x1 y1 z1 desc1  
  
SclSelectPoint pnt2 "Select the backsight point" \  
    layer2 str_no2 seg_no2 pnt_no2 x2 y2 z2 desc2  
  
set status [ SclFunction "BEARING AND DISTANCE" {  
    select_point=table { objectx objecty objectz } {  
        { "$x1" "$y1" "$z1" }  
        { "$x2" "$y2" "$z2" }  
    }  
}]
```

 **Note:** The backslashes ( \ ) used in the example are continuation characters to extend a line onto the next line. In your example you can combine these lines together.

15. Choose **File > Save**.
16. In Surpac, in the Navigator, right-click the working folder and click **Refresh**.
17. Drag **21\_bearing\_and\_distance.tcl** into **Graphics**.

You can now select any points for the bearing and distance calculation, and the script exits the bearing and distance function after your selection.

## Summary

In this chapter you have learnt how to use the **SclSelectPoint** function to better handle recorded Surpac graphics functions.

## Useful Tcl commands

After you have mastered the basics of recording scripts and adding user interfaces, you are able to set up scripts to process data. The commands described in this topic will help you to write scripts that process data efficiently. There are commands used to:

- work with numbers
- work with text
- interface with the file system

### Working with numbers

In Tcl everything is treated as a string or text value. When you have variables that you think of as integers or real numbers, they are still strings in Tcl. To be able to perform arithmetic in Tcl there are a number of commands that process arguments and internally handle the arguments as numbers.

#### The `incr` command

The `incr` command is used to add a number to the value stored in a memory variable. It is used extensively when coding looping structures such as the `for` loop. Use of the `for` loop is covered in the next chapter of this tutorial.

The command has two formats:

- `incr variableName`
- `incr variableName number`

The first format adds one to the value stored in `variableName` while the second format will add the specified `number` to the value stored in `variableName`.

#### Task: Use `incr`

1. Open ConTEXT.
2. Save a new, blank, file as **22\_use\_incr.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set i 0
incr i
puts $i
set section 1000
incr section 100
puts $section
set counter 100
incr counter -10
puts $counter
```

4. In Surpac, drag **22\_use\_incr.tcl** into **Graphics**.

You will see the following in the **message window**.

```
1
1100
90
```

## The expr command

If you want to perform complex calculations in your scripts, you must use the **expr** command. The expression command takes a standard arithmetic expression in computer format and returns the result. The following mathematical operands are ones you will commonly use for constructing expressions in your scripts. There are others commands available, for a full set of commands see the Tcl website, [tcl.tk](http://tcl.tk).

### Operator Meaning

* / %	Multiply, Divide, Remainder (integer division)
+ -	Add, Subtract

### Mathematical functions supported by the expr command

acos	cos	Hypot	sinh
asin	cosh	Log	sqrt
atan	exp	log10	tan
atan2	floor	Pow	tanh
ceil	fmod	Sin	

### Conversion functions supported by the expr command

Function	Description
abs(arg)	Returns the absolute value of arg. Arg may be either integer or floating-point, and the result is returned in the same form
double(arg)	If arg is a floating value, returns arg, otherwise converts arg to floating and returns the converted value.
int(arg)	If arg is an integer value, returns arg, otherwise converts arg to integer by truncation and returns the converted value.
rand()	Returns a floating point number from zero to just less than one or, in mathematical terms, the range [0,1). The seed comes from the internal clock of the machine or may be set manual with the srand function.
round(arg)	If arg is an integer value, returns arg, otherwise converts arg to integer by rounding and returns the converted value.
srand(arg)	The arg, which must be an integer, is used to reset the seed for the random number generator. Returns the first random number from that seed.

## Task: Use expr

1. Open ConTEXT.
2. Save a new, blank, file as **23\_use\_expr.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set x1 1000
set y1 1000
set x2 2000
set y2 2000

set delta_x [expr $x1 - $x2]
puts $delta_x

set delta_x [expr abs($delta_x)]
puts $delta_x

set line_len [expr sqrt(((x2 - x1) * (x2 - x1)) + \
                        ((y2 - y1) * (y2 - y1)))]
puts $line_len
```

4. Choose **File > Save**.
5. In Surpac drag **23\_use\_expr.tcl** into **Graphics**.

You will see the following in the **message window**.

```
-1000
1000
1414.21356237
```


## The SclExpr command

Sci has its own implementation of the Tcl expression command called SclExpr. The expression parser used by SclExpr has been developed by Surpac. This expression parser uses the same format that you use for expressions in String Maths, Field Maths, Block Maths, and DTM Maths.

The categories of functions you can use with SclExpr are:

- boolean functions
- bitwise functions
- algebraic functions
- string functions
- numeric comparison functions
- string comparison functions
- radian trigonometric functions
- degree trigonometric functions
- angle unit conversion functions

- exponential functions
- conversion functions
- numeric constant functions
- random number generation functions
- datetime functions
- miscellaneous functions

 **Note:** For further information about these expressions see the Surpac help.

### Task: Use SclExpr

1. Open ConTEXT.
2. Save and new, blank, file as **24\_use\_sclexpr.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set rad 3
set cos_radians [SclExpr COS($rad)]
puts $cos_radians

set deg [SclExpr RTOD($rad)]
puts $deg

set cos_deg [SclExpr COSD($deg)]
puts $cos_deg

puts [SclExpr FORMAT($cos_deg, 3)]
```

4. Choose **File > Save**.
5. In Surpac, drag **24\_use\_sclexpr.tcl** into **Graphics**.

You will see the following in the **message window**.

```
-0.9899924966004454
171.88733853924697
-0.9899924966004454
-0.990
```

## Working with text strings

You will often want to manipulate text strings to extract data from dfields or note files, or from any source of text data that could even include the Surpac **message window**. Tcl provides the **string** command that has a number of sub functions to extract text.

The common sub functions of the string command are:

- string length
- string index
- string first
- string range
- string trim
- string tolower
- string toupper

### string length <text string>

The *length* sub-function of **string** returns the number of characters present in the text string.

#### Task: Use string length

1. Open ConTEXT.
2. Save a new, blank, file as **25\_use\_string\_length.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set data_values "123, 456, 789"  
set len [string length $data_values]  
puts $len
```

4. Choose **File > Save**.
5. In Surpac, drag **25\_use\_string\_length.tcl** into **Graphics**.

You will see the following in the **message window**.

```
13
```

### string index <text string> <charIndex>

The *index* sub-function of **string** returns the char Index character of the text string argument. A *char Index* of 0 corresponds to the first character of the string. The *char Index* can be specified as follows:

- number - the character at the given number (starting from 0)
- end - the keyword "end" refers to the last character of the string
- end-number – the keyword "end" minus the specified number

### Task: Use string index

1. Open ConTEXT.
2. Save a new, blank, file as **26\_use\_string\_index.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set data_values "123, 456, 789"  
set char [string index $data_values 1]  
puts $char  
  
set char [string index $data_values end]  
puts $char
```

4. Choose **File > Save**.
5. In Surpac, drag **26\_use\_string\_index.tcl** into **Graphics**.

You will see the following in the **message window**.

```
2  
9
```

### string first <string1> <string2> [<startIndex>]

The *first* sub-function of **string** will search string2 for a sequence of characters that exactly match the characters in string1. If found it returns the index in the string of the first character that matched within string2. If string1 is not found in string2 it returns -1. If *startIndex* is specified the search begins from that index in the string.

Think of it in English terms as “find the first instance of string1 in string2 after position <start index>”

### Task: Use string first

1. Open ConTEXT.
2. Save a new, blank, file as **27\_use\_string\_first.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set data_values "123, 456, 789"  
set ndx [string first "," $data_values]  
puts $ndx  
  
set ndx [string first "," $data_values 4]  
puts $ndx
```

4. Choose **File > Save**.
5. In Surpac, drag **27\_use\_string\_first.tcl** into **Graphics**.

You will see the following in the **message window**.

```
3  
8
```

## string range <text string> <first> <last>

The *range* sub-function of **string** returns a range of consecutive characters from the text string, starting with the character whose index is first and ending with the character whose index is last. An index of 0 refers to the first character of the string and the keyword “end” refers to the last character in the text string.

### Task: Use string range

1. Open ConTEXT.
2. Save a new, blank, file as **28\_use\_string\_range.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set data_values "123, 456, 789"
set d1 [string range $data_values 0 2]
puts $d1

set d3 [string range $data_values 10 end]
puts $d3
```

4. Choose **File > Save**.
5. In Surpac, drag **28\_use\_string\_range.tcl** into **Graphics**.

You will see the following in the **message window**.

```
123
789
```

## string trim <text string>

The *trim* sub-function of **string** will remove any leading or trailing spaces from the given text string.

### Task: Use string trim

1. Open ConTEXT.
2. Save a new, blank, file as **29\_use\_string\_trim.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set data_values "123, 456, 789"
set d2 [string range $data_values 4 7]
puts $d2

set d2 [string trim $d2]
puts $d2
```

4. Choose **File > Save**.
5. In Surpac, drag **29\_use\_string\_trim.tcl** into **Graphics**.

You will see the following in the **message window**.

```
456
456
```

### **string tolower <text string> and string toupper <text string>**

The *tolower* and *toupper* sub-functions of the **string** command convert any characters of the original text string to lower or upper case respectively.

#### **Task: Use string tolower and toupper**

1. Open ConTEXT.
2. Save a new, blank, file as **30\_use\_string\_tolower\_and\_toupper.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set data_value "Ore Grade"
set lower [string tolower $data_value]
puts $lower

set upper [string toupper $data_value]
puts $upper
```

4. Choose **File > Save**.
5. In Surpac, drag **30\_use\_string\_tolower\_and\_toupper.tcl** into **Graphics**.

You will see the following in the **message window**.

```
ore grade
ORE GRADE
```

## Working with the File System

Interacting with the computers file system is something you must do when writing scripts. Tcl provides the *file* command that has a number of sub functions to perform many actions on the file system. These functions work in a similar manner to DOS commands.

### file copy [-force] <source file name> <target file name>

The *copy* sub-function of **file** allows you to copy files stored on disk. If the target file already exists, the command will fail, unless you have specified the *-force*.

#### Task: Use file copy

1. Open ConTEXT.
2. Save a new, blank, file as **31\_use\_file\_copy.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
file copy pit1.str new1.str
file copy -force pit2.str new2.str
```

4. Choose **File > Save**.
5. In Surpac, drag **31\_use\_file\_copy.tcl** into **Graphics**.

Check the current directory, you will see **new1.str** and **new2.str** in the directory.

### file delete [-force] <file name>

The *delete* sub-function of **file** allows you to delete files or directories stored on disk. If the target file is read only, the command will fail, unless you specify the *-force* switch.

#### Task: Use file delete

1. Open ConTEXT.
2. Save a new, blank, file as **32\_use\_file\_delete.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
file delete new1.str
file delete -force new2.str
```

4. Choose **File > Save**.
5. In Surpac, drag **32\_use\_file\_delete.tcl** into **Graphics**.

Check the current directory, **new1.str** and **new2.str** have been deleted from the directory.

### file rename [-force] <source file name> <target file name>

The *rename* sub-function of **file** allows you to rename files stored on disk. If the target file already exists, the command will fail, unless you specify the *-force* switch.

### Task: Use file rename

1. Open ConTEXT.
2. Save a new, blank, file as **33\_use\_file\_rename.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
file copy MY_WORK:pit1.str new1.str
file rename -force new2.str oldone.str
```

4. Choose **File > Save**.
5. In Surpac, drag **33\_use\_file\_rename.tcl** into **Graphics**.

Check the current directory, you will see a new file called **oldone.str** in the directory.

### file exists <file name>

The *exists* sub-function of **file** allows you to check to see if a file actually exists on the disk. This can be a useful function when tied together with an **if** statement.

### Task: Use file exists

1. Open ConTEXT.
2. Save a new, blank, file as **34\_use\_file\_exists.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
if {[file exists MY_WORK:pit1.str]} then {
    puts "Processing file ..."
} else {
    puts "File not found"
}
```

1. Choose **File > Save**.
2. In Surpac, drag **34\_use\_file\_exists.tcl** into **Graphics**.

You will see the following in the **message window**.

```
Processing file ...
```

### file mkdir <directory path> & cd <directory path>

The *mkdir* sub-function of **file** allows you to make new directories (folders) on the disk. If the directory path contains sub directories that don't exist then the whole path is created.

 **Note:** You can use **file delete** to remove a directory path.

The **cd** command allows you to change the current directory (folder) on the disk. **cd** is a command, not a sub function of the **file** command.

### Task: Use file mkdir

1. Open ConTEXT.

2. Save a new, blank, file as **35\_use\_file\_mkdir.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
file mkdir temp
cd temp
puts "in temp"
cd ..
file delete temp
puts "removed temp"
```

1. Choose **File > Save**.
2. In Surpac, drag **35\_use\_file\_mkdir.tcl** into **Graphics**.

You will see the following in the **message window**.


```
in temp
removed temp
```

### **glob [-nocomplain --] <directory specification>**

Unfortunately the **file** command cannot process wildcards. That is, you cannot use a command like **file delete \*.tmp**. Tcl provides the **glob** command to read a directory of files into a list that can then be processed in your script.

You can use the **-nocomplain** switch to stop an error occurring if no files match the directory specifications you have given.

If you want to use the **glob** command with the **file** command, you must use a specialised looping command called **foreach**. This command processes Tcl lists.


 **Note:** The lists data structure is not discussed further in this tutorial. However, the following example should demonstrate how to use lists with the **glob** command.

### **Task: Use the glob command**

1. Open ConTEXT.
2. Save a new, blank, file as **36\_use\_glob.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
set file_list [glob -nocomplain -- *.tmp *.cf]
foreach file_name $file_list {
    puts "deleting $file_name"
    file delete $file_name
}
```

4. Choose **File > Save**.
5. In Surpac, drag **36\_use\_glob.tcl** into **Graphics**.

 **Note:** This task will read all **\*.tmp** and **\*.cf** files in the current directory, then load them into a list variable called *fileList*. The *foreach* loop will then loop over each file in the list storing them one by one into the variable *fileName*. The **file delete** command deletes the files one by one as the loop iterates.

## Summary

In this chapter you have learnt some native Tcl commands to use when you write scripts. You can now:

- perform mathematical expressions
- manipulate strings
- interact with the file system including using wildcards

## Basic flow control in Tcl

After you are familiar with recording scripts and creating user forms, you will want to be able to process the data you have called into the script. To process the data you need to understand flow control, so that you can make choices in the way you write the script to get the desired result.

### Boolean expressions

Flow control constructs, for example, branching to a section of code or repeating a number of commands, are based on a conditional test. An example of this construction in English is:

*“If the user selected to plot **then** perform the plotting code”*

The keywords in the example are *if* and *then*; they describe some action. The *“user selected to plot”* is the condition or test criteria that determines whether the plotting code is executed. This condition can evaluate to only true or false. Another example:

*“While there is another section number **continue** processing sections”*

The keywords in this example are *while* and *continue*. The *“is another section number”* is the condition or test that determines if the software will continue extracting sections. Again, this test can only evaluate to true or false.

A Boolean expression is an expression that can result in only one of two possible outcomes that are **true** or **false**. The outcomes are based on testing something, whether it be comparing simple numbers or text strings, or some complicated arithmetic expression, or even combinations of numbers, text, and arithmetic.

When you write a Boolean expression it is best to sound out what you are trying to test for in English first. This helps you make sure you have your logic correct.

#### Common operators you can use to write Boolean expressions (grouped by decreasing precedence)


Operator	Example	Description
!	!\$a	Logical NOT; Negate the value on the left hand side That is, while not at the end of file
<	\$a < \$b	Boolean less than; Is the value on the left less than the value on the right That is, if a is less than b
>	\$a > \$b	Boolean greater than; Is the value on the left greater than the value on the right That is, if a is greater than b
<=	\$a <= \$b	Boolean less than equal to; Is the value on the left less than or equal to the value on the right

Operator	Example	Description
		That is, if a is less than or equal to b
>=	\$a >= \$b	Boolean greater than or equal to; Is the value on the left greater than or equal to the value on the right That is, if a is greater than or equal to b
==	\$a == \$b	Boolean equal; are the left and right sides equal That is, if a is equal to b
!=	\$a != \$b	Boolean not equal; are the left and right side different That is, if a is not equal to b
&&	\$a && \$b	Logical AND; If the left and right sides are both true then the expression is true That is, if expression a is true and expression b is true
	\$a    \$b	Logical OR; If either the left or right side are true then the expression is true That is, if expression a is true or expression b is true

The last two operators can be combined to form complex Boolean expressions using the && (and) and || (or) operators that perform comparisons on many arguments. The truth table below explains the results of using && (and) and || (or) in Boolean expressions.

	&& (and)	
	True	False
True	true	false
False	false	false

	(or)	
	True	False
True	true	true
False	true	false


 **Note:** The actual representation in Tcl for *false* is 0 and for *true* is anything but 0.

## The if Command

The Tcl **if** statement is used to branch to a section of code to be executed based on a Boolean expression. There are four possible forms of the **if** command.

### if {expression} then {command body}

A simple **if** statement can be read in English as “*if something is true then perform some action*”.

 **Note:** The use of the keyword **then** is optional.

### Task: Use an if then statement

1. Open ConTEXT.
2. Save a new, blank, file as **37\_use\_if\_then\_statement.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
set form_def {
    GuidoForm form {
        -label "GuidoField Form"
        -default_buttons

        GuidoField bearing {
            -label "Enter a bearing"
            -format float
            -width 10
            -low_bound 0
            -high_bound 360
            -null false
        }
    }
}

SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}


if {$bearing > 45 && $bearing <= 135} then {
    puts "The direction is East"
}
```

4. Choose **File > Save**.
5. In Surpac, drag **37\_use\_if\_then\_statement.tcl** into **Graphics**.

The result returned to the **message window** will vary according to your input value.

## if {expression} then {commands} else {commands}

A derivative of the basic **if** is the *If-then-else* construct that allows one of two sets of commands to be executed.

 **Note:** Only one branch can be executed when the script is run.

### Task: Use an if then else statement

1. Open ConTEXT.
2. Save a new, blank, file as **38\_use\_if\_then\_else\_statement.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
set form_def {
    GuidoForm form {
        -label "GuidoField Form"
        -default_buttons

        GuidoField bearing {
            -label "Enter a bearing"
            -format float
            -width 10
            -low_bound 0
            -high_bound 360
            -null false
        }
    }
}

SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}

if {$bearing > 45 && $bearing <= 135} then {
    puts "The direction is East"
} else {
    puts "The direction is not East"
}
```

4. Choose **File > Save**.
5. In Surpac, drag **38\_use\_if\_then\_else\_statement.tcl** into **Graphics**.

The result returned to the **message window** will vary according to your input value.

## if {expression} then {commands} elseif {commands} [...]

A further derivative of the basic `if` is the *if-then-else-if* construct that allows many tests in order to execute one of many command bodies. Note that only one branch is possible, that is to say that it is not possible for more than one of the code branches to be executed.

### Task: Use an if elseif statement

1. Open ConTEXT.
2. Save a new, blank, file as **39\_use\_if\_elseif\_statement.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
set form_def {
    GuidoForm form {
        -label "GuidoField Form"
        -default_buttons

        GuidoField bearing {
            -label "Enter a bearing"
            -format float
            -width 10
            -low_bound 0
            -high_bound 360
            -null false
        }
    }
}

SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}

if {$bearing > 45 && $bearing <= 135} {
    puts "The direction is East"
} elseif {$bearing > 135 && $bearing <= 225} {
    puts "The direction is South"
} elseif {$bearing > 225 && $bearing <= 315} {
    puts "The direction is West"
} elseif {$bearing > 315 && $bearing <= 360 ||
        $bearing >= 0 && $bearing <= 45} {
    puts "The direction is North"
}
```

4. Choose **File > Save**.
5. In Surpac, drag **39\_use\_if\_elseif\_statement.tcl** into **Graphics**.

The result returned to the **message window** will vary according to your input value.

## if {exprsn} {commands} elseif {commands} [...] else {commands}

The final derivative of the basic **if** is a construct for the *if-then-else-if* statement that allows a default branch to execute if none of the previous tested branches is true.

 **Note:** Only one branch can be executed.

### Task: Use an if elseif else statement

1. Open ConTEXT.
2. Save a new, blank, file as **40\_use\_if\_elseif\_default\_branch.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
set form_def {
    GuidoForm form {
        -label "GuidoField Form"
        -default_buttons

        GuidoField bearing {
            -label "Enter a bearing"
            -format float
            -width 10
            -low_bound 0
            -high_bound 360
            -null false
        }
    }
}

SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}

if {$bearing > 45 && $bearing <= 135} {
    puts "The direction is East"
} elseif {$bearing > 135 && $bearing <= 225} {
    puts "The direction is South"
} elseif {$bearing > 225 && $bearing <= 315} {
    puts "The direction is West"
} elseif {$bearing > 315 && $bearing <= 360 ||
        $bearing > 0 && $bearing <= 45} {
    puts "The direction is North"
} else {
    puts "The bearing cannot be classified"
}
```

4. Choose **File > Save**.
5. In Surpac, drag **40\_use\_if\_elseif\_default\_branch.tcl** into **Graphics**.

The result returned to the **message window** will vary according to your input value.

## while {expression} {commands}

The **while** command is used to repeat a section of code until a Boolean condition becomes false. The action of repeating a series of Tcl statements is called looping.

There are a number of looping statements available in Tcl, and each statement has a particular purpose. The **while** statement is used when you don't know in advanced how many times you need to repeat a section of code. For example if you were to read all the lines in a text file you don't know how many lines there will be until you read the last one.

### Task: Use a while statement

1. Open ConTEXT.
2. Save a new, blank, file as **41\_use\_while\_statement.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
set dfields "d1, d2, d3, d4, d5,"
set pos1 0
set pos2 [string first "," $dfields]
while {$pos2 > 0} {
    set dfield [string range $dfields $pos1 [expr $pos2 - 1]]
    set dfield [string trim $dfield]
    puts "$dfield"
    set pos1 [expr $pos2 + 1]
    set pos2 [string first "," $dfields $pos1]
}
```

4. Choose **File > Save**.
5. In Surpac, drag **41\_use\_while\_statement.tcl** into **Graphics**.

You will see the following in the **message window**.

```
d1
d2
d3
d4
d5
```

## for {start} {expression} {next} {commands}

The **for** command is a specialist loop control that you usually use when you know exactly how many times you want to loop. For example, if you were processing the range of sections "1000,2000,100" you know you have 11 sections to process.

The *start* section of the loop is executed once just prior to the first loop test. The *expression* is the test that determines whether the loop continues. The *next* section is executed at the end of each loop iteration. The *commands* are the statements to execute, or the body of the loop.

### Task: Use a for statement


1. Open ConTEXT.
2. Save a new, blank, file as **42\_use\_for\_statement.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
for {set count 1} {$count <= 10} {incr count} {  
    puts "This is the $count time through the loop"  
}
```

4. Choose **File > Save**.
5. In Surpac, drag **42\_use\_for\_statement.tcl** into **Graphics**.

You will see the following in the **message window**.

```
This is the 1 time through the loop  
This is the 2 time through the loop  
This is the 3 time through the loop  
This is the 4 time through the loop  
This is the 5 time through the loop  
This is the 6 time through the loop  
This is the 7 time through the loop  
This is the 8 time through the loop  
This is the 9 time through the loop  
This is the 10 time through the loop
```

 **Note:** The **incr** command used in the example adds 1 to the variable count. **incr** can take a second parameter such as `incr count -1` which will be added to the variable. This is useful when you need to loop backwards.

## Changing loop behaviour with continue and break

There are two commands that can alter the flow control in a loop. They are **continue** and **break**.

The **continue** command, when encountered, will force the Tcl interpreter to stop processing commands in the current loop iteration and continue onto the next loop cycle.

The **break** command, when encountered, will terminate the loop immediately and the Tcl interpreter will begin processing commands after the loop.

### Task: Use continue and break in a loop

1. Open ConTEXT.
2. Save a new, blank, file as **43\_use\_continue\_and\_break\_in\_loop.tcl**.
3. Copy and paste, or type, the text exactly as in the following

```
# loop forever
while {1} {
    # select a point from the graphics viewport
    set status \
        [SclSelectPoint point "Select a point (Esc 2 exit)" \
            layer str_No seg_no pnt_no x y z desc]

    if {$status != $SCL_OK} {
        # if user pressed escape exit the loop
        break
    }

    puts "Selected points coords are y = $y x = $x, z = $z"
}
```

4. Choose **File > Save**.
5. In Surpac, drag **43\_use\_continue\_and\_break\_in\_loop.tcl** into **Graphics**.

## Summary

In this chapter you have learnt about the basic flow control structures of Tcl. You can now create simple scripts that can perform conditional tests on variables and branch or loop depending upon these tests.

## Manipulating Surpac ranges with Scl

Use of ranges in Surpac is widespread and it is likely that you will need to manipulate ranges as part of processing in your scripts. This chapter describes Scl commands that you can use to process ranges for your scripts.

### SclRangeExpand

The *SclRangeExpand* command takes a Surpac range specification like “100, 200, 10” and then expands it out into its individual components in memory. Once a range specification has been expanded in memory you can access information on it using the other SclRange functions.

#### Syntax

```
SclRangeExpand RangeHandle RangeExpression
```

#### Where

*RangeHandle* is a reference variable that you will use to access the expanded range

For example, *secRange*, *inputRange*, and *fileId*.

*RangeExpression* is the actual range that is being expanded i.e. 1000, 2000, 100

#### Return Values

SCL\_OK if the function creates the form object with no error

SCL\_ERROR if something goes wrong

For example, the range specification 1000, 2000, 100 would expand into the following series of values in memory:

```
1000 1100 1200 1300 1400 1500 1600 1700 1800 1900 2000
```

### SclRangeGetCount

The *SclRangeGetCount* command will get a count of the number of items in a range that has been expanded in memory using the *SclRangeExpand* function.

#### Syntax

```
SclRangeGetCount RangeHandle
```

#### Where

*RangeHandle* is the reference handle (variable) that was created after performing SclRangeExpand For example, *\$secRange*, *\$inputRange*, and *\$fileIds*.

#### Return Values

SCL\_OK if the function creates the form object with no error

SCL\_ERROR if something goes wrong

In the previous example the returned range count would be 11 for the range 1000, 2000, 100.

## SclRangeGet

The *SclRangeGet* command will get a range value at the specified position in a range, which has been expanded in memory using the *SclRangeExpand* function.


### Syntax

```
SclRangeGet RangeHandle RangePosition RangeValue
```

### Where

*RangeHandle* is the reference handle (variable) that was created after performing *SclRangeExpand*. For example, *\$secRange*, *\$inputRange*, and *\$fileIds*.

*RangePosition* is the position in the range to get the value for

 **Note:** The first item in the range is at position 0.


*RangeValue* is the name of a variable to store the range value into

For example, *secNum*, *Id*, and *bench*. If the variable doesn't exist it is created.

### Return Values

SCL\_OK if the function creates the form object with no error

SCL\_ERROR if something goes wrong

 **Note:** The range position starts at zero and not one. The following example uses the range 1000, 2000, 100.

0 <sup>th</sup>	1 <sup>st</sup>	2 <sup>nd</sup>	3 <sup>rd</sup>	4 <sup>th</sup>	5 <sup>th</sup>	6 <sup>th</sup>	7 <sup>th</sup>	8 <sup>th</sup>	9 <sup>th</sup>	10 <sup>th</sup>
1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000

## Putting the range commands together

Using the range functions is quite simple. The following example demonstrates the use of these three range functions along with a Tcl **for** loop.

### Task: Use the range commands

1. Open ConTEXT.
2. Save a new, blank, file as **44\_use\_range.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
set sec_range "7000,8000,100"

SclRangeExpand sec_range_handle "$sec_range"

set sec_range_count [SclRangeGetCount $sec_range_handle]
for {set pos 0} {$pos < $sec_range_count} {incr pos} {

    SclRangeGet $sec_range_handle $pos sec_no
    set sec_file "sec$sec_no.str"
    puts "This section is $sec_file"
}
```

4. Choose **File > Save**.
5. In Surpac, drag **44\_use\_range.tcl** into **Graphics**.

You will see the following.

```
This section is sec7000.0.str
This section is sec7100.0.str
This section is sec7200.0.str
This section is sec7300.0.str
This section is sec7400.0.str
This section is sec7500.0.str
This section is sec7600.0.str
This section is sec7700.0.str
This section is sec7800.0.str
This section is sec7900.0.str
This section is sec8000.0.str
```

6. In ConTEXT, add a form that will get any value for the **sec\_range** variable to the beginning of the script.

 **Note:** You can use the following code sample to add this form.

```
set form_def {
  GuidoForm form {
    -label "Surpac Ranges"
    -default_buttons

    GuidoField sec_range {
      -label "Enter a range"
      -format range
      -width 20
      -null false
    }
  }
}
SclCreateGuidoForm form_handle $form_def {}
$form_handle SclRun {}
```

7. In Surpac, drag **44\_use\_range.tcl** into **Graphics**.
8. Enter a number of different range values in the *Surpac Ranges* form.
  - a. 100,200,25;235;256;300,500,50
  - b. 100
  - c. 1,10
  - d. 1000,1020,5;1002;1020,5

## Summary

In this chapter you learnt about Scl commands that you can use to manipulate Surpac ranges. The basic range template will handle any Surpac range the user inputs, regardless of whether the range is uniform, non-uniform, or both.

## File input and output – reading and writing files

When you start to write more complicated scripts you might need to bring in data from external sources such as text files or databases. To do this you can use third-party packages that interface between Tcl and a variety of data sources. The references list at the end of this tutorial provides the names of some of these third-party applications. In this chapter you will learn to read and write text files with Tcl.

### Opening and closing files

Tcl provides two commands to open and close files, called **open** and **close**. When the **open** command is used, a handle (reference to the file) is returned. The handle is then used to access the file in other commands.

#### **open** <file name> <access>

The **open** command allows you to access the file system so that you can read or write files. You must nominate what type of action you want to perform on the file by specifying the type of access as follows.


r - to read an existing file

w - to create or write a new file

 **Note:** This deletes the specified file name if it exists.

#### Examples

```
set read_file [open "datafile.dat" "r"]
set write_file [open "newfile.dat" "w"]
```

 **Note:** Note that the open command returns a reference value that you store into a variable. You use this variable to later refer to the opened file with other Tcl commands that you use to access the file.

#### **close** <file reference variable>

After you have opened a file and performed read or write functions on it, you must use the **close** command to close the file. This clears the input/output buffers in memory and releases any resources allocated to the file.

#### Examples


```
close $read_file
close $write_file
```

### Reading from a file

There are two Tcl commands that you can use to read from files, called **gets** and **read**. The **read** command is more advanced and allows block input/output. The **gets** command is simpler and reads one line of text from the referenced file.


## gets <file reference variable> <variable for read data>

The **gets** command will read a single line from the referenced file and place the read data into a variable that you specify. If this variable does not previously exist, it is created.

 **Note:** Lines are read sequentially one by one with each **gets** statement.

### Examples

```
gets $read_file line
gets $input_file inputData
```

 **Note:** With the examples above the actual data is placed into memory variables called **line** and **inputData**.

## eof <file reference variable>

When reading files you will normally set up a file read loop. You must be careful not to read past the end of the file because this will cause an input/output error, which will stop your script from running. Tcl provides an **eof** command to test for end of file.

The **eof** command returns true if the end of file marker has been read, or false if the end of the file has not been read. Usually you set up a **while** loop with your Boolean expression that, in English, reads as “*while not at the end of file keep looping*”.

### Example

```
gets $read_file line
while {[eof $read_file]} {
    # file processing stuff
    gets $read_file line
}
```

## Writing to a file

There are two Tcl commands that you can use to write to files, called **puts** and **write**. The **write** command is more advanced and allows block input/output. The **puts** command is simple and outputs a single line. You have already used the **puts** command for writing to the **message window**. You can also use the **puts** command to write a line to a file that has been opened for writing.

## puts <file reference variable> <text string>

When two parameters are specified to the **puts** command, the first is read as a file reference variable, and the second the information to write to the file.

### Examples

```
puts $write_file "some text to write to the file"
puts $write_file "$processedData"
```

## A file-processing template

The commands discussed previously in this tutorial are all you need to write simple file processing scripts. The following example gives you a template that you can base most of your scripts on. It opens one file for reading, one for writing, reads each line from the read file, and writes the lines to the new file for writing, and closes both files.

### Task: Create a file-processing template

1. Open ConTEXT.
2. Save a new, blank, file as **45\_create\_file\_processing\_template.tcl**.
3. Copy and paste, or type, the text exactly as in the following.

```
# open files for reading and writing
set read_file [open "pit1.str" "r"]
set write_file [open "new1.str" "w"]

# read the first line
gets $read_file line

# loop for each line in the file
while {[eof $read_file]} {

    # write the line to the backup file
    puts $write_file "$line"

    # read the next line
    gets $read_file line
}

# close the files
close $read_file
close $write_file
```

4. Choose **File > Save**.
5. In Surpac, drag **45\_create\_file\_processing\_template.tcl** into **Graphics**.

The template copies the contents from one file to the other, without altering the content. In practice a functioning script would process the data being read in before writing it to the output file.

## Summary

In this chapter you have learnt about the basic commands for performing file input/output. You now have a basic file processing template to use in your scripts.

## Tcl references

### Surpac help

Surpac includes a complete Tcl on-line reference manual that describes each of the Tcl commands in detail. It also includes a complete Scl and GUIDO reference manual, with code samples to demonstrate how you can use each of the Scl commands.

### Text books

Practical Programming in Tcl and Tk

Brent Welch. Prentice Hall, 1999. 3rd Ed

ISBN: 0-13-022028-0.

Effective Tcl/Tk Programming

Mark Harrison and Michael McLennan, Addison-Wesley, 1997.

ISBN: 0-201-63474-0.

Tcl/Tk for Dummies

Tim Webster with Alex Francis, IDG Books, 1997.

ISBN: 3-89319-793-1.

Tcl/Tk For Programmers

Adrian Zimmer. IEEE Computer Society, 1998.

ISBN 0-8186-8515-8.

### WWW resources

There are many internet sites dedicated to the Tcl language, a number of which offer Tcl solutions. The Scriptics site is a good place to begin, and it contains a link page to a number of other resources.

<http://www.tcl.tk>

<http://dev.scriptics.com/resource/community/websites/>

The mine solutions initiative is dedicated to promoting third party Surpac developments. It contains resources, and tips and tricks for using Tcl/Scl. vComp Pty Ltd manages the development and associated web site.

[www.minesolutions.com](http://www.minesolutions.com)

[info@minesolutions.com](mailto:info@minesolutions.com)

The GEOVIA software and GEOVIA support web sites provide links to a number of third party Tcl resources, as well as free scripts for download.

<http://www.3ds.com/GEOVIA>

[www.GEOVIAsupport.com](http://www.GEOVIAsupport.com)

---