Semper 6 *Plus*

USER INTERFACE GUIDE

Synoptics

Trademarks:

Semper is a registered trademark of *Synoptics Limited.*

No part of this manual may be copied or reproduced in any form, or by any means, without prior written permission of *Synoptics Limited.*

Manual number: SEM003
Update number: 0
Printed: June 1990

Copyright © 1990: Synoptics Ltd, All Rights Reserved

Table OF CONTENTS

Chapter 1: Overview

Chapter 2: User Interface Objects

Chapter 3: A Simple Example

Chapter 4: A More Complicated Example

Chapter 5: Useful Objects to Create

Chapter 6: Points to Watch

Chapter 7: Where to go from here

Appendix A: Troubleshooting

Chapter 1

OVERVIEW

Introduction

This guide explains how to program the user interface system which is built into Semper 6 *Plus*. The user interface system allows you to build and customize a menu interface to interact with the features of the Semper image processing language. You create user interfaces by combining various components, which are known as *objects*. These objects provide the basic building blocks of a user interface and several commands are available to create and manipulate them. These commands are listed below:

- **cell**
- **device**
- **execute**
- **justification**
- **menu**
- **mouse**
- **panel**
- **textfield**
- **uif**

For a complete description of these commands, refer to the on-line help or to the following manual:

Semper 6 Command Reference

This manual provides examples of using the above commands to create a number of user interfaces. It provides simple examples of user interface programming, which give an idea of the general principles involved.

This guide assumes that you are already familiar with Semper and the user interface system, as described in the following manuals:

Tutor User Guide and *Beginners' User Guide*

both of these manuals are contained in the *Semper 6 Guide*. It also assumes that you have some familiarity with programming in Semper, which is described in the following manual:

Advanced User Guide

also contained in the *Semper 6 Guide*.

How this manual is organised

This manual contains seven chapters and one appendix and is structured in the following way:

- Chapter 2 gives an overview of the objects provided by Semper 6 *Plus* and their functionality.
- Chapter 3 gives a simple example of programming the user interface system.
- Chapter 4 describes how you might make a pop-up calculator using the user interface toolkit.
- Chapter 5 gives some ideas about useful objects which you can create to help you in programming your own user interface.
- Chapter 6 explains how the user interface system affects Semper programming.
- Chapter 7 gives advice on what to do next.
- Appendix A describes some common error situations that arise when user interface programming. It details the action to take in these situations and includes some useful programming tips.

Conventions used in this manual

Semper commands are shown in the following font:

```
semper
```

except when they are embedded in the text, in which case they are highlighted using a **bold** font.

Chapter 2

USER INTERFACE OBJECTS

Introduction

Semper 6 *Plus* provides the following objects:

- panels
- menus
- cells
- textfields

These objects provide the basic building blocks to create a particular user interface. Figure 2-1 shows an example of these user interface objects:

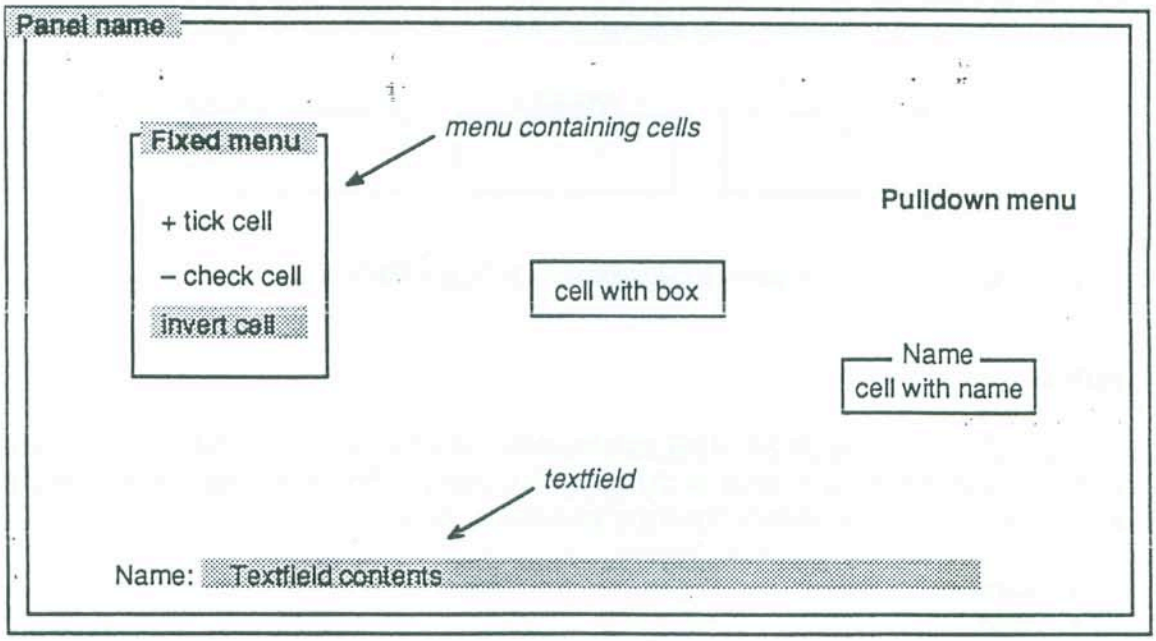

Figure 2-1. An Example User Interface

You can assemble the user interface objects in a variety of ways to provide more complicated objects, for example:

- status panels
- dialogue panels
- menu bars
- calculators

Part of the flexibility of the objects arises out of the actions that you can define for them. An *action* is a command, or series of commands, which is executed when something happens to an object, for example, when the cursor moves across the object or when the object is displayed. For example, a cell object can have three actions defined for it: one for when the cursor enters the cell, one for when the cell is selected by clicking with the cursor on it and one for when the cursor leaves the cell. You can use these actions when making a menu interface to create several different styles of interaction, for example:

- slide on, where menus pop up as the cursor moves onto their title.
- click on, where menus pop up when the cursor is clicked on the menu title.

All objects have a unique numeric identifier which is given to them when they are created: you use this when referring to the object subsequently. There is a limit to the number of objects that can exist at any one time. This is not, however, a serious limitation since you can create or destroy objects at any time. All objects can be named; the name is shown when the object is displayed and is normally positioned according to the horizontal justification in force at the time (see Figure 2-2).

Figure 2-2. Justification of Object Names

Panels

Panel objects are the foundation of the user interface. All other objects are built on panels and inherit some of the characteristics of the panel (for example the foreground and background colours) unless this is overridden. There are two types of panel:

- transient
- fixed

A transient panel can be hidden and redisplayed at will. A fixed panel remains visible after it has been shown for the first time, until you destroy it or leave the user interface. The default panel type is fixed. A panel can also be defined as mandatory, in which case the user can only interact with objects which are built on the panel. Mandatory panels gather information without which processing cannot continue, for example, a mandatory panel can be defined to ask for confirmation from the user before beginning to delete data.

Elements

There is a general class of objects called *elements* which may be placed onto panels, these are the *menu*, *cell* and *textfield* elements. The elements have a number of attributes in common including colour, actions, size and position. These elements are described in the following sub-sections.

Menus

The *menu* element comprises a matrix of cell elements, presenting the user with a choice of actions when they are displayed. A menu element can be created in one of the following styles:

- pulldown
- popup
- fixed

The pulldown menu element normally displays just its name but, after being clicked on, the actual menu drops down from the name as its name suggests (see Figure 2-3). The popup menu element is only activated by program request and may appear at any position on the screen. One application of this might be to pop up a menu at the cursor position. The fixed menu element is similar to the fixed panel object, in that once shown it remains visible until it is destroyed. The default menu style is pulldown.

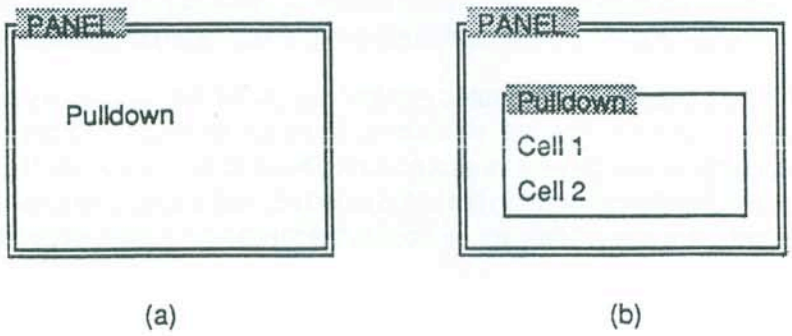

Figure 2-3. Pulldown Menu Operation (a) Inactive (b) Active

As well as having a style, menus also have a type which can be either *choice* or *toggle*. The difference between the two types of menu is only apparent if different sorts of cells are put onto a menu. In choice type menus, only one of the cells may be active at once (some types of cell may have a number of states). In toggle type menus, cells may take any state without regard to the other cells on the menu, as shown in Figure 2-4 overleaf. (For a description of cells, see the following subsection).

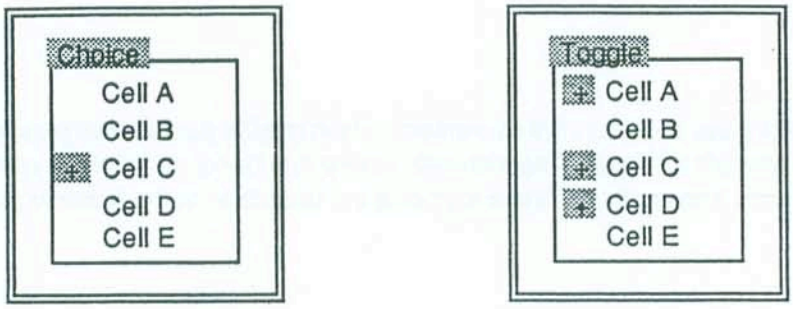

Figure 2-4. Choice and Toggle Menu Types (All Cells Are Tick Cells)

Cells

The *cell* element is like a labelled box. The box contains a text string. A cell may be placed onto a panel or onto a menu. The cell element may be displayed in one of four different styles which allow it be marked (or highlighted) in different ways. The four styles are as follows:

- check
- flash
- invert
- tick

The check style cell may be marked with a plus, '+', minus, '–', or nothing at all – three different states. The flash style cell only has one state but flashes when it is selected with the pointing device. The invert and tick style cells have two states: normal or inverted for the invert style, and '+' or no mark for the tick style. Figure 2-5 overleaf illustrates the four different styles of cell.

Note that all of the cells that are illustrated overleaf are shown with a centrally justified name; the naming of cells is optional. The cell element is, however, more sophisticated than just a box containing text because you can define actions for it. These actions are performed when the cursor is moved in and out of the cell or when the cell is selected, that is when the cursor is clicked on the cell. Cell elements, therefore, allow you to access the functions of the Semper image processing language.

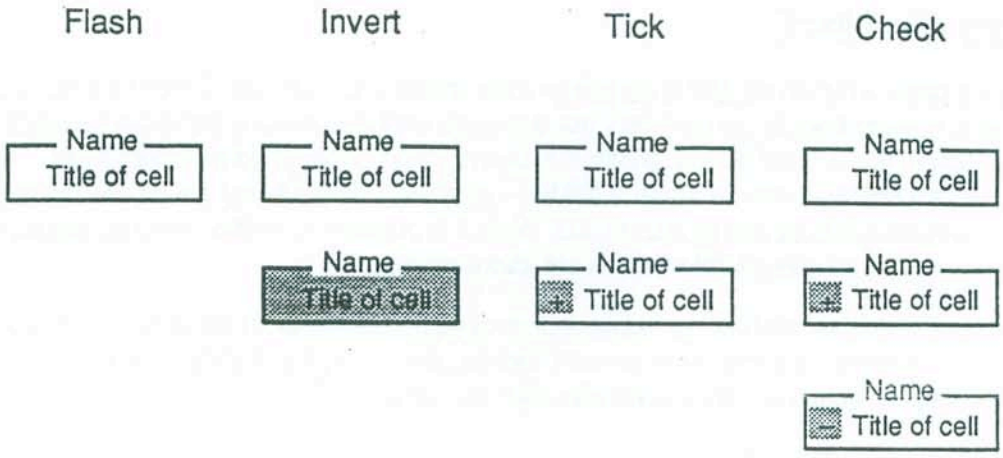

Figure 2-5. Styles of Cell Highlighting

Textfields

The final element type is the *textfield* element. This type of element is used for the input of alphanumeric data. The data may come from the keyboard or may be supplied by a program. In addition to this the contents of a textfield can be executed as a command, or evaluated as a numeric expression and stored in a variable, provided that the expression is valid.

Like cells, textfields can have a name; this is displayed to the left of the active area of the textfield. When displayed, the name and active area are separated from each other by a colon (:), as shown in Figure 2-6. A textfield can also be write-protected so that any input from the keyboard cannot overwrite its contents.

Figure 2-6. A Textfield

Using the objects

When a panel is displayed, all the objects on it are shown too. However, if one of the objects on the panel is a menu then its contents are not displayed until the menu is activated specifically. The exception to this is fixed menus, which are always shown when a panel is displayed. The same mechanism operates when a panel is hidden – all objects on the panel are hidden along with the panel, except pulldown and popup menus. While it is possible to destroy objects individually, if a panel is destroyed then all objects on it are also destroyed.

The same effects when showing, hiding or destroying panels are also true of menus. This removes the need to destroy objects, such as cells, individually. When the 'higher level' object is finished with, it may be hidden and destroyed with one command.

A note about colour

By default, the foreground colour of newly created panels is white and the background colour is black (foreground colour is used for lines and text). Menu, cell and textfield elements default to the colours of the panel to which they belong. However, you can change the colours of any of the user interface objects by specifying the colour keys, **foreground** *<number>* and **background** *<number>* with the relevant user interface command.

The following table shows the colour that the user interface system associates with a particular number. These are default settings.

Number	Colour
0	Black
1	White
2	Red
3	Green
4	Blue
5	Cyan
6	Magenta
7	Yellow

Note that some devices may not support colour or only support a restricted range, or use different defaults (for example, 0=white, 1=black).

Chapter 3

A SIMPLE EXAMPLE

Introduction

This chapter provides you with a simple example that you can follow to create a new user interface. It gives step-by-step instructions on how to use the user interface objects to create an image viewer. For example, suppose that you would like to be able to view an image in two modes:

- as a 3-D wireframe model
- as an image

To achieve this you need to create a user interface consisting of a menu with options to display the picture in the required style and, of course, an option to return to the operating system. You also need to create a new run file instead of *semper.run*. (Note that it is better to develop larger programs as a series of library programs that can be run from within Semper).

This example is made up of two stages. The first stage consists of writing a set up procedure to define the framework for the new user interface. The second stage involves creating the user interface system itself. These stages are described in the following sections.

Set up procedure

The first step in this example program consists of writing a number of set up procedures to perform the following tasks:

1. Initialise the display and display partitions.
2. Initialise the work disc to store the captured picture.
3. Initialise the user interface system and set the justification of objects.
4. Check that a mouse is available.
5. Determine the size of the terminal.

First of all you need to write a series of commands to initialise the display and the regions of the screen where the output will be shown (the display partitions). Write the following commands to a text file and name the file *test.run*.

```
trap = -1 assign display
if (rc = 0) jump asd
type 'Sorry: unable to initialise the display'
stop
asd:
```

```
partition 1                ;! One full screen partition
partition 2                ;! A second full screen partition
lut 1 create monochrome    ;! Select a monochrome look-up table
view lut 1 frame           ;! Load the table into the framestore
```

Note the use of the **trap** command to prevent Semper itself reporting an error if the framestore does not initialise correctly. Next, you need to add commands to the text file to initialise a work disc to store the calculated picture. This is 512kb in size so that you can calculate a large picture.

```
trap = -1 assign scratch size 512
if (rc = 0) jump ass
type 'Sorry: not enough space for scratch disc'
stop
ass:
cd = n
```

Again, the **trap** command ensures that the program catches any errors and not Semper. Next, you add commands to initialise the user interface system and set the justification of the user interface objects.

```
uif enable
justification top
```

Note that it is easy to forget the fact that the default justification of user interface objects is central. If you do forget this you will find, if you position your panels too near the edge of the display, that Semper will report an error when you try to display the panel for the first time. The next stage in the program is to check that the system has a mouse. While the user interface system does work without a mouse, it is not possible to select any objects without one:

```
mouse query
if (nbuttons > 0) jump mpresent
type 'Sorry: there does not appear to be a mouse on your system'
stop
mpresent:
```

The **mouse query** command returns the number of buttons on the mouse in the variable *nbu(ttons)*, and a reasonable assumption to make is that no buttons means no mouse. The mouse may, in fact, be any form of pointing device (a tracker ball, digitising tablet); this depends on the type of hardware that you have attached to your system. Finally, as part of the set up, you determine the size of the terminal so that you can locate the panel containing the menu:

```
device query 1
sx = uix
sy = uiy
```

The **device** command sets the Semper variables *uix*, *uiy* (device size) and *ncolours* (number of colours on the display) according to the characteristics of the specified device. The type of device is specified by the **query** key; 1 is the terminal and 2 is the framestore (if the user interface system supports it). The Semper environment is now suitably defined to allow you to create a user interface.

Creating a user interface

This stage consists of creating a user interface which will give the user a choice of viewing the picture as a wireframe model or as an image. To do this, you need to write a program to perform the following tasks:

1. Create a panel.
2. Create a menu to contain the required functions.
3. Create the cells that comprise the menu.
4. Display the panel that contains the menu.
5. Create an image to be displayed.
6. Run the user interface.

The first step is to create a panel, as follows:

```
sp = 25
if (sp > sx) sp = sx
panel create size sp,9 auto transient position (sx / 2), 2
$pd = pno
```

This particular user interface has defined the justification, at this stage, to be central and top so the panel has to be positioned about its central *x* coordinate. In Semper 6 *Plus* all object coordinates are specified in terms of characters, with *x* coordinates increasing across the screen and *y* coordinates increasing down the screen. The **panel** command returns the identifier of the panel created in the variable *pno* so this is stored in the variable *$pd* for future reference. (If you create any panels subsequently, their identifiers would overwrite the value of the variable *pno*). Next, you create the menu that will contain the required functions:

```
menu create pulldown name 'Test'
mid = eno
```

Notice that this time you do not give a position because the menu is positioned relative to the panel, not to the display. The **menu** command returns the identifier of the created menu in the variable *eno* (short for element number) which is then stored in the variable *mid* for future reference . The next step is to create the cells that comprise the menu:

```
drop = yes; add = mid; invert = yes; create = yes; column = 1
cell text 'Erase image          ' row 1 changes 'erase image frame'
cell text 'Erase overlay        ' row 2 changes 'erase overlay frame'
cell text 'Display image        ' row 3 changes 'display cd:1 dis:1'
cell text 'Display wireframe    ' row 4 changes 'ymod cd:1 dis:2'
cell text 'EXIT                 ' row 5 changes 'uif stop'
unset drop, add, invert, create, column
```

Here, rather than specifying all the keys and options with each command, the constant ones are set before the commands are executed. They are unset at the end to prevent any interactions with any commands executed later and also to reduce the number of variables that the application uses (since the number of variables allowed in Semper is fixed). The option **drop** causes the menu to be

de-activated automatically whenever one of the cells is selected, removing the need to program this directly.

There are two main parts to the **cell** command as shown: the part which labels the cell, appearing on the menu, given by the **text** key and the action performed when the cell is selected, specified by the **changes** key. The spaces have been put in the cell titles, padding all cells on this menu to the same length, so that they are justified correctly. The **row** key gives the cell's vertical position on the menu; you would use the **offset** keys if you wanted to specify a character position on the menu, rather than positioning the cells in terms of the objects on the menu with the **row** and **column** keys.

Next, you display the panel which contains the menu:

```
panel show
```

and create the image which you wish to display:

```
create cd:1 size 256,256 fp
calculate sin(x / 16) * cos(y / 8) to cd:1
```

After creating the picture, the **calculate** command generates a sine/cosine surface. This statement could be replaced by the **live** command to capture a live image (if your Semper system supports it) and a **copy** command to store the resultant image on disc. The final command in the program runs the user interface:

```
uif go
```

Summary

The complete example program is given below:

```
trap = -1 assign display
if (rc = 0) jump asd
type 'Sorry: unable to initialise the display'
stop
asd:
partition 1                ;! One full screen partition
partition 2                ;! A second full screen partition
lut 1 create monochrome    ;! Select a monochrome look-up table
view lut 1 frame           ;! Load the table into the framestore
trap = -1 assign scratch size 512
if (rc = 0) jump ass
type 'Sorry: not enough space for scratch disc'
stop
ass:
cd = n
uif enable
justification top
mouse query
```

```
if (nbuttons > 0) jump mpresent
type 'Sorry: there does not appear to be a mouse on your system'
stop
mpresent:
device query 1
sx = uix
sy = uiy
sp = 20
if (sp > sx) sp = sx
panel create size sp,3 auto transient position (sx / 2), 2
$pd = pno
menu create pulldown name 'Test'
mid = eno
drop = yes; add = mid; invert = yes; create = yes; column = 1
cell text 'Erase image       ' row 1 changes 'erase image frame'
cell text 'Erase overlay     ' row 2 changes 'erase overlay frame'
cell text 'Display image     ' row 3 changes 'display cd:1 dis:1'
cell text 'Display wireframe' row 4 changes 'ymod cd:1 dis:2'
cell text 'EXIT                    ' row 5 changes 'uif stop'
unset drop, add, invert, create, column
panel show
create cd:1 size 256,256 fp
calculate sin(x / 16) * cos(y / 8) to cd:1
uif go
```

One point which is not obvious in this program is that Semper labels have only three significant characters. The labels in the program (*asd*, *ass*, *mpresent*) have been carefully chosen so that they are distinct. Note also that the command **uif stop** is defined as a cell action to exit the program. This command leaves the user interface and returns to the operating system. The command **uif exit** can also be used – this command leaves the user interface and returns not to the operating system but to the Semper command-line environment. For this reason, it is often more convenient to use **uif exit**, in place of **uif stop**, when you are developing a program.

All that remains now is to run the interface. Ensure that your text file is called *test.run* and type the following command at the operating system prompt:

```
semper /run=test
```

This command creates and runs the interface that you have just defined. Your interface should look like those illustrated in Figures 3-1 and 3-2 overleaf.

Figure 3-1. Simple Interface – Inactive State

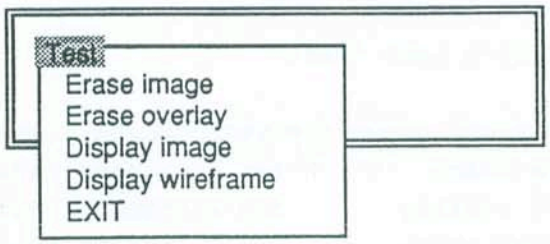

Figure 3-2. Simple Interface – Active State

A note about the Semper scrolling area

When you create a user interface, Semper uses some of the space on the screen for its messages and for displaying output from its commands – this is called the Semper scrolling area. The user interface system calculates the size of the scrolling area when you use the **ulf go** command. It calculates how far down the screen the *displayed* panels extend and reserves the remaining area for the scrolling area. This is illustrated in Figure 3-3 which uses the simple interface, described in this chapter, as an example.

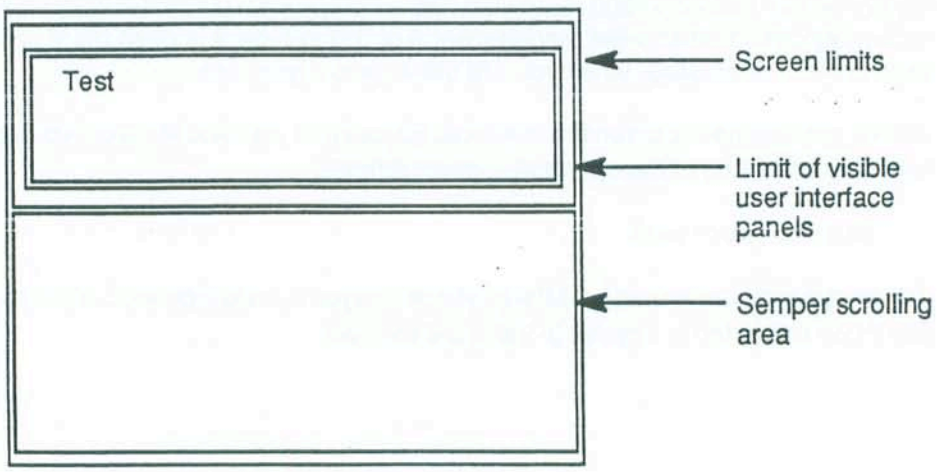

Figure 3-3. Scrolling Area Definition

Chapter 4

A MORE COMPLICATED EXAMPLE

Introduction

The example program described in this chapter produces a calculator. It is written so that it can be incorporated easily into any user interface. This introduces more of the user interface commands and some more of the techniques which may be of use in programming a user interface. In this example, the calculator is created and controlled through a series of library programs. If you are in any doubt about how to write a program and add it to a Semper program library, refer to *Chapter 6, Programming with Semper* in the following manual:

Beginners' User Guide

The example consists of writing a program to perform the following tasks:

1. Create a panel to contain the calculator elements.
2. Create the cells and textfields that comprise the calculator.
3. Reset the default justification.
4. Return from the library program.
5. Write a function to evaluate the expression entered into the calculator.
6. Write a function to hide the panel when the OK button is clicked.

A calculator

As a first step, you need to write a program to create a panel on which you will place the other elements. Create a text file called *calcfuns.spl* to contain the following calculator definition:

```
make$calculator()

justification top left
! Create the 'Calculator' panel
panel create name 'Calculator' size 17,8 position 30,15 auto +
      transient foreground $c0 background $c1
$pc=pno
```

The justification of all elements is set since the button positions are specified (later) relative to the top left hand corner of the panel. The calculator is created on a transient panel so that it may be popped up at will. The command also introduces the use of foreground and background colours, specified in the variables *$c0* and *$c1*: these have to be set up when you initialise the user interface system (which is described later). The panel identifier is stored in the variable *$pc* since the

calculator will be used later and other programs may use the **panel** command, destroying the returned value of *pno*. The buttons of the calculator will be represented by cells and the calculator display will be represented by a textfield. You use a textfield because it allows characters to be added to its contents interactively, without needing to be hidden and re-displayed like other user interface objects.

Next, the cells and textfields comprising the different components of the calculator are created:

```
create=yes invert=yes
! Program the calculator buttons
cell position  2,6  text ' 0 ' changes 'tex id $cd app ''0'''
cell position  1,5  text '1'   changes 'tex id $cd app ''1'''
cell position  3,5  text '2'   changes 'tex id $cd app ''2'''
cell position  5,5  text '3'   changes 'tex id $cd app ''3'''
cell position  1,4  text '4'   changes 'tex id $cd app ''4'''
cell position  3,4  text '5'   changes 'tex id $cd app ''5'''
cell position  5,4  text '6'   changes 'tex id $cd app ''6'''
cell position  1,3  text '7'   changes 'tex id $cd app ''7'''
cell position  3,3  text '8'   changes 'tex id $cd app ''8'''
cell position  5,3  text '9'   changes 'tex id $cd app ''9'''
cell position  6,6  text '.'   changes 'tex id $cd app ''.'''

cell position  8,3  text '*'   changes 'tex id $cd app ''*'''
cell position 10,3  text '/'   changes 'tex id $cd app ''/'''
cell position  8,4  text '+'   changes 'tex id $cd app ''+'''
cell position 10,4  text '-'   changes 'tex id $cd app ''-'''
cell position  8,5  text '('   changes 'tex id $cd app ''('''
cell position 10,5  text ')'   changes 'tex id $cd app '')'''

cell position 12,3  text ' C  ' changes 'tex id $cd cle;uns h'
cell position 12,4  text 'Exp ' changes 'tex id $cd app ''E'''
cell position 12,5  text 'Root' changes 'tex id $cd app ''Root('''

! Both the '=' and '[OK]' buttons evaluate the display but the OK
! button hides the calculator panel too

cell position  8,6  text ' =    ' changes 'lib c$eval'
cell position 12,6  text '[OK]' changes 'lib c$eval;lib c$ok'

! Main 'display' textfield
textfield position 1,2 length 15 name '' end '' wp
$cd = eno

unset create, add, invert
```

Note that in this program, all the Semper commands have been truncated to three characters in the text of the **changes** keywords. This is quite legitimate and has two advantages: the program will run slightly faster and it conserves 'string' space (where the names of objects and their actions are stored). (In the user interface system there is only a limited amount of 'string' space: the **uif status** command gives details of what is left.)

To see how the calculator works look at the action defined for one of the calculator's buttons, for example, for the '8' button the **changes** action is defined as:

```
tex id $cd app ''8''
```

or, in full:

```
textfield id $cd append ''8''
```

from this you can see that the **textfield** command adds an '8' to the contents of the display textfield. This does mean that, unlike a normal calculator, it is possible to enter an illegal numerical expression but this is handled when the expression is evaluated: this is described later. The most important two buttons on the calculator are the **equals** button and the **OK** button. Selecting either of these two buttons calls the library program *c$eval*. The **OK** button also calls the *c$ok* library program which turns off the calculator (hides the panel). After evaluating the expression and providing that the expression is correct, the result of the calculator will be returned in the variable *h*. (Notice that the clear button unsets this variable to indicate that there is no result.)

The remaining steps are to add the commands to reset the default justification and return from the library program:

```
justification
return
end
```

You have now defined the calculator panel, the next step is to write the functions called when the expression in the display textfield is to be evaluated:

```
c$eval()

! Evaluate the expression in the display textfield
trap = -1 textfield id $cd assign 'h'
if (rc = 0) jump done
beep
type 'Error in expression'
unset h
textfield id $cd clear
jump qce
done:
textfield id $cd contents h
qce:
return
end
```

This program shows how powerful textfields are; their contents may be evaluated as a numeric expression, the result being put into a Semper variable. Notice that the variable is in quotes so that Semper does not attempt to evaluate it and the user interface system can determine the name of the variable rather than its value. If there is an error in the expression, the textfield is cleared and a simple form of error reporting is used – a better method might be to display an information panel. (An example of how to make information panels is given in *Chapter 5, Useful Objects To Create).* If the expression is evaluated correctly the textfield displays the result as well as putting it in a variable. The only remaining part of the program is concerned with hiding the panel when the **OK** button is clicked:

```
c$ok()

! Hide the calculator panel
panel id $pc hide
return
end
```

Summary

The complete program to create a calculator is given below. We have called this program *calcfuns.spl.*

```
make$calculator()

justification top left
! Create the 'Calculator' panel
panel create name 'Calculator' size 17,8 position 30,15 +
auto transient foreground $c0 background $c1
$pc=pno

create=yes invert=yes
! Program the calculator buttons
cell position  2,6  text ' 0 ' changes 'tex id $cd app ''0'''
cell position  1,5  text '1'   changes 'tex id $cd app ''1'''
cell position  3,5  text '2'   changes 'tex id $cd app ''2'''
cell position  5,5  text '3'   changes 'tex id $cd app ''3'''
cell position  1,4  text '4'   changes 'tex id $cd app ''4'''
cell position  3,4  text '5'   changes 'tex id $cd app ''5'''
cell position  5,4  text '6'   changes 'tex id $cd app ''6'''
cell position  1,3  text '7'   changes 'tex id $cd app ''7'''
cell position  3,3  text '8'   changes 'tex id $cd app ''8'''
cell position  5,3  text '9'   changes 'tex id $cd app ''9'''

cell position  6,6  text '.'   changes 'tex id $cd app ''.'''

cell position  8,3  text '*'   changes 'tex id $cd app ''*'''
```

```
cell position 10,3  text '/'   changes 'tex id $cd app ''/'''
cell position  8,4  text '+'   changes 'tex id $cd app ''+'''
cell position 10,4  text '-'   changes 'tex id $cd app ''-'''
cell position  8,5  text '('   changes 'tex id $cd app ''('''
cell position 10,5  text ')'   changes 'tex id $cd app '')'''

cell position 12,3  text ' C  ' changes 'tex id $cd cle;uns h'
cell position 12,4  text 'Exp ' changes 'tex id $cd app ''E'''
cell position 12,5  text 'Root' changes 'tex id $cd app ''Root('''
! Both the '=' and '[OK]' buttons evaluate the display but the OK
! button hides the calculator panel
cell position  8,6  text ' = '  changes 'lib c$eval'
cell position 12,6  text '[OK]' changes 'lib c$eval;lib c$ok'

! Main 'display' textfield
textfield position 1,2 length 15 name '' end '' wp
$cd = eno

unset create, add, invert
justification
return
end

c$eval()

! Evaluate the expression in the display textfield
trap = -1 textfield id $cd assign 'h'
if (rc = 0) jump done
beep
type 'Error in expression'
unset h
textfield id $cd clear
jump qce
done:
textfield id $cd contents h
qce:
return
end

c$ok()

! Hide the calculator panel
panel id $pc hide
```

```
return
end
```

The program for the calculator has been written so that you can incorporate it into your own user interfaces. To test the design, create the following run file, called *calc.run*, to display the calculator:

```
uif enable
$c0 = 1 $c1 = 0 ! Set calculator colours
trap = -1 assign name 'library' program
if (rc = 0) jump lok
type 'Cannot assign library disc'
stop
lok:
add name 'calcfuns'          ! Text file containing the
                             ! calculator definition
mouse right 'uif stop'       ! Exit button
lib make$calculator          ! Build calculator
panel id $pc show            ! Display it
uif go                       ! Start executing
end
```

To display the calculator, type the following command at the operating system prompt:

```
semper /run=calc
```

When you pop up the calculator, it should look something like the one shown in Figure 4-1.

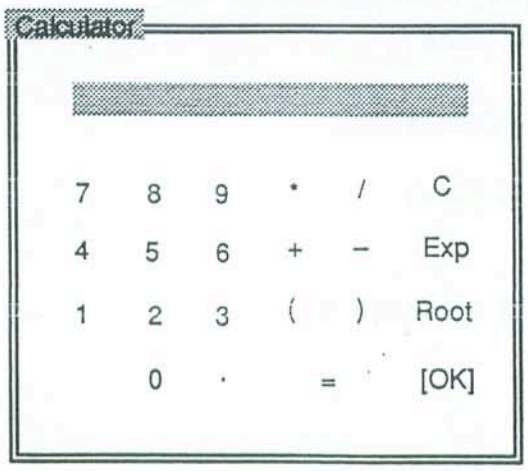

Figure 4-1. The Calculator

The run file is obviously only sufficient to test and verify the functionality of the calculator. You can find out the number of colours that your system supports using the **device query** command, which returns the number of colours on your display in the variable *ncolours*, so that you can change the *$c0* and *$c1* variables. See *A note about colours* in *Chapter 2, User Interface Objects* for further detail.

The program described in this chapter will pop up the calculator, allow expressions to be evaluated and let control be returned to the Semper command line. Some suggested enhancements to the calculator might include:

- Using colours to indicate the different functions of the calculator buttons rather than just the foreground and background colours of all objects. This would depend on the ability of the display to show them.
- Make the calculator pop up at the cursor position, rather than at a fixed position. (Using the **mouse query** command.)
- Adding a memory function to the calculator.

Chapter 5

USEFUL OBJECTS TO CREATE

Introduction

When programming with the user interface system, it is quite common to find that a number of similar operations are required. For example, it may be necessary to display an information message, pausing until the message has been read. This chapter describes some commonly used objects which you can use to implement these operations. The following objects are discussed:

- A wait panel
- An information panel
- A dialogue panel
- Context sensitive help for menus

A wait panel

Some of the more complex image processing functions provided by Semper may take a considerable time to execute on large pictures. Rather than displaying a blank screen while Semper is calculating, it may be appropriate to display a message. Two functions provide this facility, one creates the message panel and the other destroys it when the operation is complete.

```
pwc()                          ! Create please wait panel

justification                  ! Central
panel create auto transient position $cx, $cy foreground $c1 +
   background $c0;$pw = pno
cell create text 'Please wait'
panel show                     ! pno still has panel number just created
return
end

pwd()                          ! Destroy please wait panel

unless (set($pw)) return
panel destroy id $pw
unset $pw
return
end
```

The library program *pwc* expects you to define the following variables so that it can display the wait panel in the centre of the screen:

$cx	central *x* (column) position of screen
$cy	central *y* (row) position of screen
$c1	colour of text on panel
$c0	colour of panel background

The justification of the panel is central so that the message does not appear in a lop-sided manner. The destroy routine, *pwd*, only destroys the panel if it exists. Notice that a transient panel does not need to be hidden in order for it to be destroyed. The variable *$pw*, which contains the wait panel identifier, is unset after the panel is destroyed to indicate that the panel does not exist any more. Figure 5-1 illustrates a wait panel.

Figure 5-1. A Wait Panel

An information panel

Many applications may require that one of a number of messages is displayed on the screen. An information panel is one way of achieving this. A panel is displayed, with the appropriate message and an **OK** button on it. When the user clicks on the button, the panel disappears. The first step to create an information panel is given below:

```
inform() ! Display information message

local $ip, $ic

justification
panel create auto transient mandatory size 40,8 position +
   $cx, $cy name 'Information' foreground $c1 background $c0
$ip = pno ! Record panel id
```

The variables *$ip* and *$ic* are only used in this routine, so they are made local to it (they will be thrown away when the *inform* program returns). Notice that the panel is defined as mandatory, so that no interactions with other (visible) menus are allowed while it is showing. The panel itself is given a name, which will appear in the middle of the top of the panel (because of the central justification).

The next step is to define the variable *op* in a cell:

```
if (op > 10) jump i10
if (op = 1) cell create text 'Bad picture' position 20,2
if (op = 2) cell create text 'Area of particle',a position 20,2
if (op = 3) cell create text 'Not enough space on disk' position 20,2
jump idone
i10:
idone:
```

The variable *op* is used, effectively, as a function argument. The text relating to *op* equals 1 is only shown for the sake of example. The reason for jumping when *op* is greater than 10 to is reduce the number of **If** statements that have to be checked in the program. This mechanism can obviously be extended for cases where *op* might be greater than 20, etc. Next, you create the **OK** button:

```
cell create text ' OK ' box position 20,5 +
changes 'pan des id ',$ip,';uns $mn;lib op;lib mr'
$ic = eno
```

A box is drawn around the cell to indicate the button-like qualities that it offers. The **changes** text, that is the Semper commands which will be executed when the button is clicked on, is more understandable when written out in full:

```
panel destroy id <$ip>;unset $mn;library op;library mr
```

where *<$ip>* is replaced by the number of the information panel (for example, **panel destroy id 12**). This is a feature of the way in which Semper handles text strings and is useful as it removes the need to keep the variable *$ip* after the *inform* program finishes executing. This is especially useful in large Semper applications where you may use many variables. The library programs *op* and *mr* restore the old mouse position and functionality respectively, as they are about to be altered. The next stage displays the information panel:

```
$mn = yes
panel id $ip show
device save 1
mouse left '' centre '' right '' id $ic
return
end
```

The variable *$mn* is set to indicate that a mandatory panel is showing. (Any attempt to display other menus or panels will cause Semper to generate an error). After showing the panel, the current cursor position is saved using the **device** command and then the mouse button actions are re-defined. The **mouse id** command positions the cursor over the **OK** cell – this is quite reasonable since the user can only interact with this panel. Using the **id** key saves having to calculate the desired cursor position – this is especially convenient if the program is to run on a number of different terminal screens. At this stage the information panel is showing, with the cursor positioned over the **OK** button. Clicking on the button destroys the panel and re-positions the mouse in its old position.

The library program *op* (old position) looks like this:

```
op()

device restore 1
return
end
```

Alternatively, the string '*device restore 1*' could be substituted for the '*library op*' string in the **changes** action:

```
cell create text ' OK ' box position 20,5 +
changes 'pan des id ',$ip,';uns $mn;dev res 1;lib mr'
```

The mouse action redefinition program, *mr*, depends on the actual user interface implemented and is not shown here.

In order to call the information function, you use a line similar to the following:

```
local op;op = 3 lib inform
```

Figure 5-2 shows an example of an information panel.

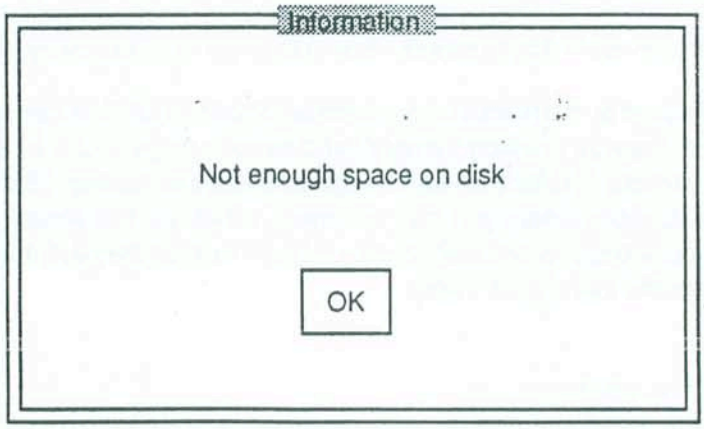

Figure 5-2. The Information Panel

Another information panel

The last section showed you how to create an information panel. When the panel was required, it was created and displayed. If you are going to use information panels frequently, then it is more efficient to create the panel when you start your program and change the message before you show the panel. This method requires the use of two library programs, the first of which is given below:

```
init$inform() ! Create information message panel

justification
```

```
panel create auto transient mandatory size 40,8 position +
   $cx, $cy name 'Information' foreground $c1 background $c0
$ip = pno ! Record panel id
cell create text ' OK ' box position 20,5 +
   changes 'pan hid id ',$ip,';uns $mn;lib op;lib mr'
$ic = eno
cell create text ' ' position 20,2
$im = eno
return
end
```

Note that this time the variables describing the parts of the information panel are not local, since you will need to refer to them later, and that the id of the message cell is also stored. The other difference is that the changes action of the **OK** cell is to hide the panel rather than destroy it. The changes action still requires the use of the library programs *$op* and *$mr* to restore the old mouse position and functionality. The remaining program shows the information panel and positions the mouse correctly:

```
show$inform()
$mn = yes
panel id $ip show
device save 1
mouse left '' centre '' right '' id $ic
return
end
```

As before, the fact that the panel is mandatory is noted, as is the mouse position. The new mouse position and functionality are specified before returning. In this version of the information panel, the message is set up by the routine which calls the *show$inform* program. Again, this is faster, because the routine does not have to perform a large number of comparisons. A typical use of the panel is shown below:

```
.
.
trap = 95 calculate :a / :b to c
unless (rc = 95) jump nozero
cell id $im text 'Divide by zero'
lib show$inform
jump done:
nozero:
.
.
done:
return
end
```

The library program attempts to form the ratio of two pictures, *a* and *b*, storing the result in another picture, *c*. The program uses the **trap** command to stop Semper itself reporting any attempt to divide by zero (which causes error 95). Another advantage of this type of information panel is that it helps anyone reading the program to understand it, as it is clear that error 95 means that an attempt to divide by zero has occurred.

A dialogue panel

This is similar to the information panel described above, except that two buttons are provided: one for 'YES' and the other for 'NO'. The dialogue panel is useful in situations where some drastic action is to be confirmed, for example deleting a picture. The commands are similar to those of the information panel:

```
dialogue() ! Display dialogue message

local $dp, $dc

justification
panel create auto transient mandatory size 40,8 +
   position $cx,$cy name 'Are you sure?' +
   foreground $c1 background $c0
$dp = pno ! Record panel id
if (op > 10) jump i10
if (op = 1) cell create text 'Delete picture' position 20,2
if (op = 2) cell create text 'Proceed with copy' position 20,2
jump idone
i10:
idone:
cell create text ' YES ' box position 10,5 +
   changes 'p1=',$dp,' p2=yes lib dcb'
cell create text ' NO ' box position 30,5 +
   changes 'p1=',$dp,' p2=no lib dcb'
$dc = eno
$mn = yes
panel id $dp show
device save 1
mouse left '' centre '' right '' id $dc
return
end
```

As with the information panel, the cursor is positioned over a button (the **NO** button) when the dialogue panel is first shown. Notice that, rather than have several lines to redefine the **changes** action for each different value of the variable *op*, the button actions call another library program which looks like this:

```
dcb()     ! Dialogue Call Back

panel destroy id p1          ! Remove dialogue panel
device restore 1             ! Old cursor position
lib mr                       ! Old mouse functionality
if (op = 1) lib dpcb
if (op = 2) lib cpcb
return
end
```

As before, the panel is destroyed and the cursor is moved back to its old position. In *dcb*, after restoring the previous state, a routine is invoked with the variable *p2* indicating whether a YES or a NO response was given. This routine (*dpcb* – delete picture call back) can take the appropriate action according to the response. Other routines can be added to this 'call back' routine as new opcodes are added to the *dialogue* program. An example of a dialogue panel is shown in Figure 5-3.

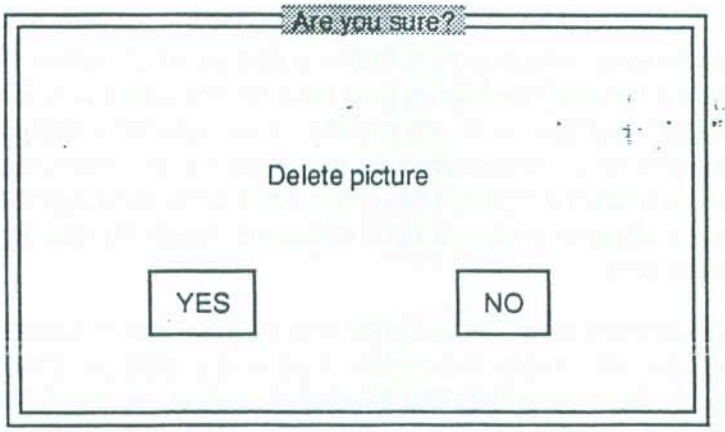

Figure 5-3. A Dialogue Panel

The *dialogue* program illustrates one important point when programming using the user interface commands: your application must wait for the user to initiate the actions defined by the user interface. For example, after showing the dialogue panel, the application returns to waiting for command input; effectively it is at an internal command line. When the appropriate button, **YES** or **NO**, is selected there has to be a way for the program executed by the action of the button to discover that it was during the *Delete Picture* request that the dialogue panel was made. Hence the need for a routine which passes a value, in the variable *op*, to say which function originally made the request. This aspect of programming user interfaces arises very frequently and is easily overlooked – see *Chapter 6, Points To Watch* for further detail.

Context sensitive help

In many applications it is quite useful to be able to provide context sensitive help, that is help relevant to the menu that is currently displayed. One method of achieving this is given below. The first step is to define a 'hidden' panel – this is never displayed and contains a textfield for executing the help commands. After enabling the user interface system, you add the following commands to the start-up program:

```
panel create auto transient name 'Hidden Panel'
textfield create length 50
$ht = eno
```

Notice that the panel identifier is not stored because it is only ever necessary to use the textfield. When help is required the following library program is called (for example as the **begins** action of a cell).

```
hfn() ! Context sensitive help

textfield id $ht contents 'lib h',p1 execute
cls
return
end
```

The library program expects one argument, in the variable *p1*, which defines which help function to call. For example, if *p1* equals 7 then library program *h7* will be called. Notice that the relevant help program will be called ***after*** *hfn* returns; which is why there is a **cls** command (clear screen) after the **textfield execute** command. The user interface system cannot cause commands to be executed at random points within a library program; it can only execute commands at points where the program would normally be waiting for keyboard input. *Chapter 6, Points To Watch* provides more detail about this important point.

The actual library programs called, as part of the help, might consist of a series of **type** commands giving a brief description of the help. You could create a more sophisticated program by specifying a flag which indicates the level of help that is required, calling the actual Semper help library if necessary.

Chapter 6

POINTS TO WATCH

How actions are translated into Semper commands

In *Chapter 5, Useful Objects To Create,* the section which described creating a dialogue panel mentioned that your application must wait for the user to initiate the actions defined by the user interface. An application consists of a (large) number of individual Semper programs which are bound together with the actions generated by the user interface system. This can be quite difficult to understand.

In the case of the dialogue panel, the difficulty arises because it is as if the program has to break off suddenly after displaying the panel. When either of the buttons **YES** or **NO** is clicked, the program must continue with the subsequent commands as if it had not returned in order to wait for command input. What you want to achieve, effectively, is something like this:

```
panel id $dp show
device save 1
mouse left '' centre '' right '' id $dc
<get action here>
panel destroy id p1          ! Remove dialogue panel
device restore 1             ! Old cursor position
lib mr                       ! Old mouse functionality
if (op = 1) lib dpcb
if (op = 2) lib cpcb
```

In a conventional, non menu-driven environment the Semper program might look like this:

```
if (op = 1) type 'Delete picture'
if (op = 2) type 'Proceed with copy'
ask answer
cls
if ((op = 1) & (answer = yes)) jump deleteit
if ((op = 2) & (answer = yes)) jump copyit
```

The way that the user interface system operates means that the original program must be split into two parts. One program displays the panel with its message and then Semper returns to waiting for command input (that is waiting for an action from the user interface system). The other program decodes the response on the basis of the message that was displayed and the answer that was given.

In the conventional, command-line orientated environment the program might look like the one given below:

```
fun1()
if (op = 1) type 'Delete picture'
if (op = 2) type 'Proceed with copy'
return
end

fun2()
cls
if ((op = 1) & (answer = yes)) jump deleteit
if ((op = 2) & (answer = yes)) jump copyit
.
.
return
end
```

You would execute the two programs as follows:

```
op = 1;lib fun1
ask answer
lib fun2
```

When you use the user interface system, the **changes** action of the **YES** (or **NO**) button effectively replaces the **ask answer** command. So, if the **YES** button was clicked, it is as if the following commands were executed:

```
op = 1;lib dialogue
p1=1;p2=yes;lib dcb
```

One way to make this process easier to understand is to re-define the **changes** actions of the **YES** and **NO** cells in the *dialogue* program. These cells are now defined as follows:

```
cell create text ' YES ' box position 10,5 +
   changes 'p1=',$dp,' p2=yes lib cb',op
cell create text ' NO ' box position 30,5 +
   changes 'p1=',$dp,' p2=no lib cb',op
```

This uses a similar technique to the one described in the section of *Chapter 5* called *An information panel* where the way in which Semper allows text strings to be defined is used to alter the **changes** action. In this case, if the *dialogue* program is called with *op = 1*, meaning show a message asking 'Delete picture' then, when one of the **YES** or **NO** buttons is selected a program called *cb1* is invoked. The full **changes** action for the **YES** button would look like this, after Semper evaluated it:

```
p1=2;p2=yes;library cb1
```

assuming that the panel identifier (*$dp*) is 2.

The program *cb1* would replace the *dpcb* and *dcb* programs with the following lines of Semper commands:

```
cb1()                          ! Call back routine for delete picture
lib dcb                        ! Tidy screen and restore mouse
.                              ! continue with deletion process
.
.
return
end

dcb()                          ! Dialogue call back, tidy screen and mouse
panel destroy id p1            ! Destroy the dialogue panel
device restore 1
lib mr
return
end
```

The routine *dcb* is called by any of the programs that use the dialogue panel so that each routine does not have to perform these operations. The execution of this style of programming will be faster because the *action* program does not have to perform several comparisons of the form:

```
if (op = 1) lib dpcb
```

in order to continue its execution.

Execution of a textfield's contents

Earlier, in *Chapter 5, Useful Objects to Create*, the section called *Context sensitive help* made the point that the user interface system cannot cause commands to be executed at random points within a library program. Figure 6-1 shows how the context sensitive help actually executes.

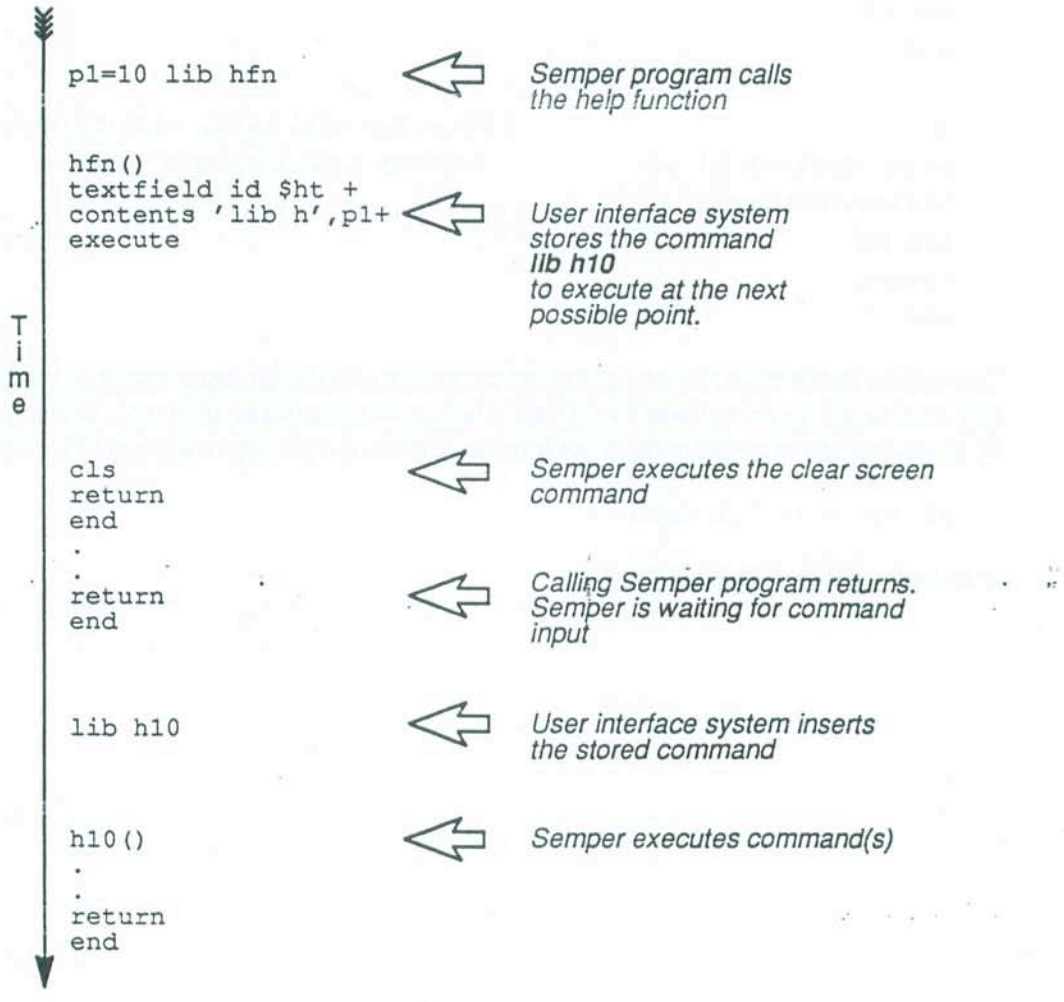

Figure 6-1. Execution of a Textfield's Contents

This is one of the hardest things to grasp when using the contents of a textfield as a Semper command. It is perhaps easiest to see this principle in operation by experimenting with simpler textfields.

Consider the following Semper program:

```
test1()

local $tf
panel create auto transient
textfield create length 40
$tf = eno
a = 7
textfield contents 'type a' execute
a = -918
type a
panel destroy
return
end
```

A textfield is created on a standard panel, notice that it is unnecessary to display the textfield. In fact, this is quite a useful feature because you can make textfields longer than the display width if you would like to use them to execute Semper commands without displaying a textfield. Executing this program causes the following output to appear on the terminal:

```
-918
-918
```

This is because the first assignment to the variable *a* does nothing – the **textfield execute** command just queues the **type a** command for execution later. You could write the program as follows:

```
a = 7;a = -918;type a
type a
```

It is important to realise that the commands given as the contents of the textfield are only executed when *all* library programs have returned so the following program would cause *exactly* the same output to occur as it did in the previous example when you executed the program *test1*:

```
test2()

local $tf
panel create auto transient
textfield create length 40
$tf = eno
a = 7
textfield contents 'type a' execute
panel destroy
return
end
```

```
test1()

lib test2
a = -918
type a
return
end
```

If you are going to use textfields to execute Semper commands it is well worth spending some time experimenting with them. There is, of course, no necessity for you to write Semper programs using textfields in this way but, occasionally, it makes programming some aspects of a user interface simpler.

Chapter 7

WHERE TO GO FROM HERE

What next?

This guide has introduced some of the ideas involved in using the user interface system. The easiest way to become familiar with the user interface commands is to experiment with them: since Semper is an interactive language you can do this quickly and easily. For further details of the Semper language it is a good idea to read the:

Advanced User Guide

You may also find that the following manuals are helpful:

- the *Quick Reference List* provides a list of all the available Semper commands.
- the *Semper 6 Command Reference* gives a full description of these commands (including the user interface commands).

You can also refer to the Semper on-line help for a comprehensive description of each Semper command. Note that the demonstration user interface, *Tutor*, uses many of the user interface commands in combination with conventional image processing commands. The library programs which make up *Tutor* may be useful to give further ideas as to how to program with the user interface commands – but they are quite complicated. Therefore, it is best to examine them only after you have experimented with the user interface commands for a while and are familiar with the way in which they operate. You may find that the manual:

Tutor User Guide

is useful for providing ideas about how to create your own user interfaces.

Note that the *Advanced User Guide*, *Quick Reference List* and *Tutor User Guide* are all contained in the *Semper 6 Guide*.

Appendix A

TROUBLESHOOTING

Overview

This appendix describes some common error situations that may arise when user interface programming and suggests the action to take in these circumstances.

Programming tips

A number of points are set out below which you may find useful when user interface programming:

- When you are developing a user interface, it is advisable to define a textfield that allows you access to the Semper command line. This textfield can be used to debug your program and can be removed when you are satisfied with the user interface.
- It is quicker to display and hide a *transient* panel than to create and destroy a *fixed* panel.
- It is necessary to surround *quotation marks (")* or *apostrophes (')* with double quotation marks in the text of a **cell** command, otherwise Semper cannot detect the correct end of a **cell** statement. An example of using punctuation in a **cell** command is given below:

```
cell create text 'This program illustrates Semper"'"s unique'+
   position 1,1
```

Known problems

The following recognised problems occur when using the user interface system:

- Use of the **pcb** command. The **pcb** command sets the variable *ncolumns* (number of columns) which clashes with the uif variable *ncolours* (number of colours) – as Semper only reads the first four characters (*ncol*). Therefore it is advisable to reset the *ncolours* variable if you use the **pcb** command within a user interface program. The **pcb** command also sets the variable *for* (picture form) which clashes with the uif variable *for* (foreground colour). To prevent Semper reporting an error, you need to unset the *for* variable each time you use the **pcb** command within a user interface program, using the following command:

```
unset for
```

- Use of the **ask** command. Under certain circumstances use of the **ask** command within a user interface program can provoke a problem with the Semper input queues and cause your program to 'lock' the machine. It is advisable not to use **ask** unless absolutely necessary.

UIF Error Messages

The following error messages are specific to the user interface system.

250 ***No UIF error – status set TRUE with no error code***

You should not normally get this error but you may see it when performing operations on elements with the variable *eno* set incorrectly. Have you omitted the **create** option from a **cell, menu, panel** or **textfield** command?

251 *UIF is not initialised*

Error 251 means that you have issued a user interface command before specifying the **uif enable** command. The **uif enable** command is necessary to initialise internal variables.

252 ***Maximum number of windows exceeded***

Error 252 means that you have specified too many windows. A window is another name for a *panel* – so you are using too many panels. The **uif status** command reports the maximum number of panels that are allowed on your system.

253 ***Maximum number of panels exceeded***

Error 253 means that you have specified too many panels. The **uif status** command reports the available number of panels. Note that pull-down and pop-up menus both use up (hidden) panels.

254 ***Maximum number of scrolling areas exceeded***

Reserved for future use.

255 ***Maximum number of cells exceeded***

Error 255 means that you have specified too many cells. The **uif status** command reports the maximum number of cells that are allowed on your system.

256 ***Maximum number of menus exceeded***

Error 256 means that you have specified too many menus. The **uif status** command reports the available number of menus on your system.

257 ***Maximum number of textfields exceeded***

Error 257 means that that there are too many textfields. Use the **uif status** command to list the available number of textfields.

258 ***Operation cannot be performed while panel is showing***

Error 258 means that you are trying to perform an operation when a panel is visible. This may be caused by:

- an attempt to add a new element to the panel
- an attempt to alter any of the characteristics of the panel

259 ***Operation cannot be performed while scrolling area is showing***

Reserved for future use.

260 ***Operation cannot be performed while panel is not showing***

Error 260 means that the operation cannot be performed while a panel is hidden – use the **panel show** command to show the panel.

261 ***Error setting variable value***

Error 261 occurs when the user interface system is setting a value to a variable. This is most likely if you have used too many variables (try **unset**ting some of them). If not, this indicates an internal error. Contact *Synoptics* for advice.

262 ***Error getting variable value***

An internal error, please report this to *Synoptics*.

263 ***Error creating window***

An internal error, please report this to *Synoptics*.

264 ***Error clearing window***

An internal error, please report this to *Synoptics*.

265 ***Error obtaining size of window device***

An internal error, please report this to *Synoptics*.

266 ***Error showing window***

An internal error, please report this to *Synoptics*.

267 ***Error destroying window***

An internal error, please report this to *Synoptics*.

268 ***Error hiding window***

An internal error, please report this to *Synoptics*.

269 ***Error moving window***

An internal error, please report this to *Synoptics*.

270 ***Error creating panel***

An internal error, please report this to *Synoptics*.

271 ***Error destroying panel***

An internal error, please report this to *Synoptics*.

272 ***Error hiding panel***

An internal error, please report this to *Synoptics*.

273 ***Error showing panel***

Error 273 means that the panel is too big to display on the device (framestore/host screen) or the panel is located outside the display limits. Very often, this is because **justification** is incorrectly set – try changing the justification to **top left** instead of central (the default).

274 ***Error moving panel***

Error 274 means that an attempt was made to move the panel outside the device limits.

275 ***Error naming panel***

An internal error, please report this to *Synoptics*.

276 ***Error setting panel/element colours***

You are trying to set a colour that your hardware cannot support. The **device query** command reports the number of supported colours on your device. See also the section called *A note about colour* in *Chapter 2, User Interface Objects*.

277 ***Error clearing device***

An internal error, please report this to *Synoptics*.

278 ***Attempt to perform operation when mandatory panel showing***

Error 278 means that a *mandatory* panel is showing but you are trying to perform some other user interface related action, for example showing another panel or popping up a menu. Remember that when a mandatory panel is showing interactions are only allowed with the panel or with elements on it.

279 ***Invalid panel identifier***

Error 279 means that you have used an identifier for a panel which the user interface system knows nothing about. Ensure that you specify the **id** key with the **panel** command.

280 ***Invalid scrolling area identifier***

Reserved for future use.

281 ***Error showing scrolling area***

Reserved for future use.

282 ***Invalid position***

Error 282 means that an attempt has been made to position the panel outside the limits of the device (framestore/host display).

283 ***String too short for copy/concatenate operation***

An internal error, please report this to *Synoptics*.

284 ***UIF initialisation failed***

Error 284 means that the user interface system is unable to initialise. Have you already initialised the system using the **uif enable** command? If not, it is advisable to exit Semper and then re-start the user interface system.

285 ***Invalid device requested***

Error 285 means that an attempt was made to use an unknown device for a panel. Is the variable *cdi* set correctly? Permitted values for *cdi* are 0, 1 and 2.

286 ***Invalid element type***

An internal error, please report this to *Synoptics*.

287 ***Invalid action***

An internal error, please report this to *Synoptics*.

288 ***Invalid element position/size***

Error 288 means that the specified element (menu, cell, textfield) was too big or was badly positioned on its panel. If it is badly positioned, it may be advisable to change the justification to **top left** instead of central (the default).

289 ***Invalid cell identifier***

Error 289 means that you have used an identifier for a cell which the user interface system knows nothing about. Ensure that you have specified a correct value with the **Id** key and check that the variable *eno* is set correctly. Also ensure that you have specified the **create** option with the **cell** command.

290 ***Invalid cell highlighting type***

An internal error, please report this to *Synoptics*.

291 ***Invalid textfield identifier***

Error 291 means that you have used an identifier for a textfield which the user interface system knows nothing about. Ensure that you have specified a correct value with the **Id** key and check that the variable *eno* is set correctly. Also ensure that you have specified the **create** option with the **textfield** command.

292 ***Invalid textfield length***

Error 292 means that you have specified an incorrect length for a textfield. Textfield lengths should be greater than zero and less than the amount of (dynamic) memory available. Type **uif status** for a list of the available memory.

293 ***Invalid numeric textfield range specifier***

Reserved for future use.

294 ***Numeric textfield contents out of range***

Reserved for future use.

295 ***Unable to convert numeric textfield contents to a number***

Reserved for future use.

296 ***Unable to convert number to a string***

Error 296 means that Semper was unable to convert the specified string into a number.

297 ***Invalid menu identifier***

Error 297 means that you have used an identifier for a menu which the user interface system knows nothing about. Ensure that you have specified a correct value with the **Id** key and check that the variable *eno* is set correctly. Also ensure that you have specified the **create** option with the **menu** command.

298 ***Invalid menu type***

An internal error, please report this to *Synoptics*.

299 ***Invalid menu style***

An internal error, please report this to *Synoptics*.

300 ***Menu panel has not been created***

An internal error, please report this to *Synoptics*.

301 ***Action string is too long to fit application input buffer***

Error 301 means that the string that you specified for an action is too long to fit into Semper's command line buffer. An action might be using the **begins** key action on a cell, or the click of a **left** mouse button, or the execution of a textfield. Try breaking the command down into smaller parts.

302 ***Framestore access error***

Error 302 means that the framestore reported an error when the user interface system was accessing the framestore. Check for framestore hardware failure.

303 ***UIF is not running***

An internal error, please report this to *Synoptics*.

304 ***Window of requested size/position will not fit device***

Error 273 means that the panel is too big to display on the device (framestore/host screen) or that the panel is located outside the display limits. Very often, this is because the **justification** is incorrectly set. Try setting the justification to **top left** instead of central (the default).

305 ***UIF termination failed***

An internal error, please report this to *Synoptics*.

306 ***Cursor position stack is empty***

Error 306 means that you have specified **device restore** before **device save** (or that you have used **restore** more times than the **save** option).

307 ***Cursor position stack is full***

Error 307 means that you have specified too many **device save** commands. The cursor position stack is of a fixed size, typically about four positions. Have you omitted to restore a position somewhere?

308 ***Dynamic memory system not initialised***

An internal error, please report this to *Synoptics*.

309 ***Dynamic memory system already initialised***

An internal error, please report this to *Synoptics*.

310 ***Invalid block size of dynamic memory requested***

An internal error, please report this to *Synoptics*.

311 ***Invalid logical index to dynamic memory used***

An internal error, please report this to *Synoptics*.

312 ***Dynamic memory logical index table full***

An internal error, please report this to *Synoptics*.

313 ***Dynamic memory exhausted***

Error 313 means that all the memory reserved for storing names of objects (panels, elements), actions, etc. has been used. **uif status** will show how much you have left. Note that the amount of memory available is fixed and installation dependent.

314 ***Invalid positioning point***

An internal error, please report this to *Synoptics*.

315 ***Invalid mouse button number***

An internal error, please report this to *Synoptics*.

316 ***Display is not assigned for windows on framestore***

Error 316 means that you have tried to show a panel on a display that you have not **assigned**.

317 ***Invalid element identifier***

An internal error, please report this to *Synoptics*.

318 ***Invalid panel/scrolling area/element identifier***

An internal error, please report this to *Synoptics*.

Index

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W