Finesse User's Guide

Finesse User's Guide

Version 4

June 1996

science + computing gmbh
Hagellocherweg 71
D-72070 Tübingen, Germany
Phone: (49) 0 70 71 / 94 57-0
Fax: (49) 0 70 71 / 94 57 27

Finesse User's Guide

Printed in Germany

Contents

Figures

Tables

How to use this manual

Purpose and audience

This guide describes how to install and use Finesse. This information is required by the system manager when setting up Finesse and by the user when developing Finesse scripts.

Using this guide

Chapter 1 introduces users to Finesse. It will show you who would want to use the product.

Chapter 2 covers the installation of the system as well as describing system requirements.

Chapter 3 contains the release notice which would be of value to both the system administration and the user.

Chapter 4 contains the Finesse user guide. It describes the commands and how they work.

Chapter 5 contains the reference pages for the Finesse commands.

Chapter 6 gives two complete examples of Finesse shell scripts.

Appendix A lists error messages and their meanings.

The glossary contains definitions for Finesse and X Windows System terms.

Notational conventions

This section discusses notational conventions used in this book.

`Bold monospace`	In command examples, text shown in **`bold monospace`** identifies user input that must be typed exactly as shown.
`Monospace`	In paragraph text, `monospace` identifies command names, system calls, and data structures and types. In command examples, `monospace` identifies command output, including error messages. In command syntax diagrams, text shown in `monospace` must be typed exactly as shown.
Italic	In paragraph text, *italic* identifies new and important terms and titles of documents. In command syntax diagrams, *italic* identifies variables that must be supplied by the user.
{ }	In command syntax diagrams, text surrounded by curly brackets indicates a choice. The choices available are shown inside the curly brackets and separated by the pipe (|) sign. The following command example indicates that you can enter either `a` or `b`: `command { a | b }`
$\{ \ldots \}_+$	In command syntax, text surrounded by curly brackets with subscript + indicates a choice that is repeated once or more.
[ ]	In command syntax diagrams, square brackets indicate optional data. The following command example indicates that definition of the variable *output_file* is optional: `command` *input_file* [*output_file*]
...	In command syntax, horizontal ellipsis shows repetition of the preceding item(s). The following command example indicates you can optionally specify more than one *input_file* on the command line. `command` *`input_file`* `[`*`input_file`* `...]`

key	In paragraph text, text shown in **key** indicates keyboard keys you must press to execute the command. For example, **return** refers to the carriage return key. Two **key** terms separated by a hyphen indicate two keys that you must press simultaneously. For example, **ctrl-d** indicates that you must press the **d** key while holding down the **ctrl** key.

Special characters in window declarations

The following characters have a special meaning within window declarations:

; (semicolon)

Terminates widget declaration.

= (equal sign)

Is used for setting variables to a value.

# (sharp sign)

Comment character.

tab

Is a word separator and a default separator for `-items` entries.

space

Is a word separator and a default separator for `-items` entries.

newline

Is a word separator and a default separator for `-items` entries.

' ' (single quotes)

Eliminate special meaning of other special characters. The values of the following keywords can be surrounded by single quotes:

`-label`, `-title`, `-name`, `-items`, `-fsbutton`, `-inputsep`, `-outputsep`, and assigning a value to a variable: `var=`*value*. Single quotes inside of single quotes have to be escaped by a \ (backslash).

Notes and cautions

This document presents notes and cautions in the following formats.

Note

A Note highlights supplemental information.

Caution

A Caution highlights information necessary to avoid damage to the system.

Associated documents

Using this software may require information not specific to the tasks described in this document. The following list of documentation may assist you with X Windows programming questions.

X Window System User's Guide

OSF/Motif User's Guide

For more detailed information on the X window system, you can find the following books from O'Reilly and Associates at your local computer book store.

X Protocol Reference Manual

Xlib Progammer's Manual

Xlib Reference Manual

X Window User's Guide

XToolkit Intrinsics Progammer Manual

XToolkit Intrinsics Reference Manual

Motif Programming Manual

The X Window System in a Nutshell

Overview 1

This chapter contains information for users of Finesse. Basic concepts of Finesse will be presented.

OSF/MOTIF interface for shell scripts

Implementing point and click dialogs in a shell script is usually very time consuming and needs advance experience in window programming. With Finesse you can generate these dialogs quickly without any knowledge of X window or C programming.

With simple shell commands you create OSF/Motif windows where users can select or fill in the appropriate values. On confirmation, these values are returned to the shell script as shell variables and are available for further processing.

Variable window construction

You can build dialog boxes out of an extensive set of window elements. Windows can be customized to meet various needs for different input with the use of the following elements:

- labels
- push buttons
- text fields
- file selection boxes
- separators
- option menus
- check buttons
- radio buttons
- lists
- forms

Customized resources may be set up to accommodate your personal preferences. These resources include items such as:

- fonts
- colors
- widget sizes
- window positions

Easy handling

Since Finesse is shell based, applications do not have to be compiled and linked. The syntax of the shell you are using will be used to generate windows so you do not have to learn a "new language." Existing shell scripts can be modified to incorporate Finesse, and modifying window layouts is done quickly.

A comfortable resource mechanism manages parameter settings that are specific for a certain user or application. Repeated access to the same window automatically activates the previous parameter settings.

Range of use

Finesse is not only an excellent tool for creating new applications, it is also useful for updating existing shell procedures. By choosing appropriate window elements you can handle input far more efficiently and minimize input errors.

Generating graphical user interfaces for Finite Element Analysis, simplified handling of complex UNIX commands and of administration scripts are but a few samples out of the wide range of uses that Finesse supplies.

Availability

Finesse is available on the following platforms:

Table 1 Supported platforms

Platform	**Operating System**
HP PA-RISC	HP-UX 9.0.X, 10.10
IBM RS/6000	AIX 3.2.5, 4.1.4
DEC Alpha	OSF1 V3.2
Silicon Graphics	IRIX 5.x, 6.2
Convex C-Series	ConvexOS 11.x
SUN SPARC	SunOS 4.1.3
SUN SPARC	Solaris 2.4, 2.5
LINUX	Linux 1.2.13
SCO Open Server	SCO_SV 3.2

Installation

2

The Installation Chapter contains information on installing Finesse.

Getting started

To install Finesse you will need:

- The Finesse software
- A Finesse evaluation license or Finesse development license
- This book, *Finesse User's Guide*

Installation

Your Finesse software is distributed as a compressed tar file. In order to install the Finesse software, do the following:

Step 1 Make a directory to install the Finesse files into.

Example: `# mkdir /usr/local`

Step 2 cd into the installation directory.

Example: `# cd /usr/local`

Note

The Finesse files will be extracted starting with the finesse directory.

Step 3 Install the Finesse software on this directory, either by copying it to the directory or by extracting it from tape.

Example: `# tar xvf devicename`

where *`devicename`* is your tape drive name, such as `/dev/rmt/0mn` for DAT at an HP.

Step 4 Uncompress the tar file:

Example: `uncompress finesse-`*version*`.tar.Z`

Step 5 Restore the finesse files:

Example: `tar xvf finesse-`*`version`*`.tar`

Step 6 Refer to the "After the installation" section on page 6 for configuration information.

After the installation

If you have not restoredFinesse in `the /usr/local` subdirectory, set the `FINESSEPATH` environment variable to the Finesse installation directory, including the `finesse` subdirectory from the tar file.

Example: `setenv FINESSEPATH /inst/dir/finesse`

Note

The environment variable FINESSEPATH only needs to be set if the installation did NOT take place in /usr /local/finesse. If FINESSEPATH is not set, the program searches in /usr/local/finesse.

Refer to "User guide," on page 17 of this book for information on using Finesse.

Licensing

There are three types of license keys for Finesse.

1. Finesse Evaluation key

   The evaluation key is good for a time limited evaluation of Finesse. The evaluation key is contained in the `$FINESSEPATH/.finesselicence` file.
2. Finesse Development License key

   The development key is a node locked key that allows unrestricted use of Finesse on that workstation. The development key is contained in the `$FINESSEPATH/.finesselicence` file.
3. Finesse Distribution License Key

   Any finesse application may be run-time licensed by generating a specific license key for that application. Run-time licensed applications will run on any workstation and platform supported by finesse without further licensing. Your finesse distribution contains the shell script `fskeygen` that automatically generates and inserts run-time licenses into applications. `fskeygen` requires a finesse distribution license key to be able to generate runtime licenses. This license key has to be inserted into the file `$FINESSEPATH/.applkeygenlicence`.

Refer to "Licensing of runtime applications," on page 18 for more information on licensing.

Getting a license

Your Finesse software is distributed as a compressed tar file. License keys are either contained in this file or provided separately, depending on your distribution. Please see your distribution notes for details of how to get a valid license.

Release notes version 4

3

Finesse version 4 offers a bunch of features that have been enhanced since version 3. Among the most important are the possibility to convert resource settings during runtime, a perl interface, "shell-callbacks" for radio, check, and option menus, the possibility to define default fields and default actions, as well as standard output being redirected to the Finesse echo window automatically. The look-and-feel of Finesse applications has been made even pleasanter by reducing the modality of Finesse windows. This chapter is organized into the following categories:

- Enhancements—these are new features in V4 of Finesse compared to V3.
- Fixed problems—these problems have been solved in Version 4 of Finesse
- Known Problems—these problems have been identified as of Juni 15, 1996. Problems reported after this date are not included in this document.

Enhancements

The following new features have been added to Finesse for V4.

Modifying resource settings

The `Fsdisplay` command has a new `-r` option to modify X resource settings in Finesse elements during runtime. This comes in useful, for example, if on recalling a window you want to change the background color for text fields with wrong input in order to give users an optical help for correcting their input. Calling

`Fsdisplay -r` *name*`:`*resource*`:`*value*

converts the X resource *resource* in widget *name* to the value *value*. For simple Finesse elements such as `FsPushButton`, *name* means the name given in the window declaration by the `-name` option.

For compound elements such as `FsText`, consisting of label field and text field, a number enclosed in square brackets can be added if the desired X resource is not to be set for the main element. The `FsText` main element is the text field, the label field has number 2. Calling

```
Fsdisplay -r mytext:background:red \
          -r mytext[2]:background:blue
```

will set the background colors for text element `mytext` to different values for label field and text field. The chapter on **Setting X Resources** contains a table that tells you which element is the main element (`[0]`) and what are the numbers of other elements. Fields in menus (`FsRadio`, `FsCheck`, and `FsOptionMenu`) are also referenced by the name of their Finesse element and receive successive numbers. An element with no name set in the window declaration cannot convert settings of X resources. Elements designed to convert resource settings must not contain colons in their names.

Perl interface

Now Finesse commands can be embedded in perl scripts. To achieve this, Finesse has got an additional shell type `perl` that returns window information according to perl syntax. The user interface has been realized with an initialization script `fsperlinit` similar to those of the shell types `sh` and `csh`, and being called at the beginning of the perl script dialog part.

Shell callbacks for menus

So far the widget `FsPushButton` was the only one to allow returning to the shell script by using the `-winstat touch` option and so to link certain widgets with certain shell script actions (shell callbacks). Now this is possible as well in connection with the menu fields `FsRadio`, `FsCheck`, and `FsOptionMenu`. These three elements now possess additional options

`-winstat {` **`input`** `| touch }`

`-data {` **`yes`** `| no }`

`-fsbutton` *value*

to realize multiple connections between widgets ("When radio element `'Canis maior'` is pressed then select item `sirius` from list `'main_stars'`").

Menus containing `-winstat touch` return to the shell script when any field is pressed. Window values are returned or not according to the `-data` option. If the `-fsbutton` option has been set, the variable `fsbutton` contains the corresponding value when returning, otherwise the label of the menu field concerned is returned in the variable `fsbutton`. So in the script you can check flexibly either the menu or a single menu field.

Changes in modality of windows

Typically, a Finesse dialog is modal, i.e. usually one and only one of the windows shown possesses the input focus, and only there user input is possible. This quality reflects the sequential course of scripts in the window dialog and makes possible programming the user dialog in the script in a simple and structured way. This modality has been reduced by several modifications from the previous version while consistency to existing programs is completely guaranteed:

- Generally windows are not any more insensitive; in particular, there is no window shadowing any more. All the same, input is still not possible in an inactive window, i.e. in a window with status `touch`. You still cannot return to the shell script from an inactive window. Windows are signaled inactive by a one-way cursor.
- Echo window and save window for saving variables are always accessible, i.e. input sensitive, independent of other windows: they can be closed anytime, for example.
- External list windows and file selection windows are not modal any more, i.e., those windows can be called simultaneously and more than once from the main input window. Input in those windows can be made at the same time as in the main input window. When the main window is inactive or closed, the corresponding list and file selection windows are inactive or closed as well.

These modifications accomplish a user interaction that is more comfortable and can be used more variably in the window dialogs concerned.

Redirecting standard output

Now the Fsopen command can be called with the option `-o`:

```
Fsopen -o { 1 | 2 | 3 }
```

Thereby it is possible to redirect standard output and standard error to the echo window. In `sh` type scripts afterwards all the standard output / error is redirected to the echo window automatically up to the `Fsclose` command. In particular: by `-o 1` standard output only, by `-o 2` standard error only, by `-o 3` standard output as well as standard error. In `csh` type scripts options 2 and 3 are not different from each other as this shell type does not support different routing of standard output and standard error. Option `1` is different from option `2` and `3` in that with `1`, Finesse server error output is not redirected to the echo window. This can be important especially in cases where there are errors in the settings of X resources by means of the `-r Fsdisplay` option (see above). Furthermore, in csh scripts you have to make sure with every command that standard output and standard error are redirected to echo windows. This can be done by using the Finesse environment variable `$FSSTDOUTFILE`. For example:

```
echo hallo > $FSSTDOUTFILE
```

Comments in window declarations

Now window declarations may contain comments. The comment character is # (sharp sign). All characters following a # are ignored up to the next **newline** character.

Default buttons

The default button of a window can be chosen by the `FsWindow` option

`-bdefault` *name*

name designs the name of the `FsPushButton` activated by pressing return inside the window. If the `-bdefault` option is missing, the leftmost `FsPushButton` field of the `FsPushButton` row that was the last to be declared in the window definition is set as default. Setting a default only has an effect when the window is opened for the first time as the default field changes according to user input.

Double clicking in lists

You can link double clicking in a list to any push button in the window concerned. The `FsList` option

`-expert` *pushbutton_name*

on double clicking executes the action that is defined by the `FsPushButton` referenced by *pushbutton_name*. If the `-expert` option is omitted, double clicking activates the default button of the window, if this exists.

Default action in text fields

Usually, pressing **return** in a text field of `FsText` and `FsSelectionText` widgets executes the default action of the window. The option

`-bdefault` *pushbutton_name*

may link a text field to any push button. Pressing **return** in the text field then activates the corresponding push button.

Error handling of Fsopen and Fsdisplay

Error handling of `Fsopen` and `Fsdisplay` commands now is placed in the initialization script. Therefore, an explicit status check of these commands in the script is not necessary any more. If the corresponding fsopen and fsdisplay commands return on error the Finesse application terminates with an exit value of 1. In particular, infinite loops created by missing status checks can thus be avoided in the script.

List selection

Now you can browse lists in single mode while holding the mouse button down in order to select an item. Your selection becomes effective when you release the mouse button. This offers more comfort for selecting items.

Width of FsPushButton fields

The `FsPushButton` option

`-packing {` **`equal`** `| tight }`

for the first push button within a row allows you to set individually the width for every one button. Default is giving equal width to all buttons in a row. The `-packing tight` option prevents the width from being adapted automatically so that you can set every single width by setting X resources.

Initializing empty info fields

The info field in the upper part of a Finesse window is generated only if it is specified by the `-m` argument in the `Fsdisplay` command. An initially empty info field, which shall receive its contents when calling the window later on, at least has to be initialized with a blank by calling `Fsdisplay -m ' '`.

Multi-line labels

Newline characters are interpreted in `FsLabel`, too, so now you can create labels with more than one line.

Local path extension

Starting with version 4, elementary Finesse commands used for initialization scripts are accessible by local path extensions. In this way installation and are completely independent from system paths, and path extensions containing the application are unnecessary. Only the specification of the environment variable FINESSEPATH is required, if any other than the Finesse default path /usr/local/finesse is used for installation.

Documentation

Version 4 documentation has been completely revised and supplemented.

Examples

Starting with version 4, the directory `examples` takes the place of the subdirectory `demo`. It contains several subdirectories `csh`, `perl`, `sh`, `app-defaults`, `app-defaults/bitmaps`. Version 4 has been supplemented with some new examples. The `examples` directory contains the master demos `examples_sh` and `examples_csh`, from where the individual examples in their corresponding subdirectories can easily be called. Apart from this, the examples can still be called directly.

Bug Fixes

The following bugs have been corrected from version 3.2.3:

- Window titles in file selection windows and external list windows were set differently, depending on the architecture used. If the elements `FsSelectionText` and `FsList` don't set a `-title` option the corresponding push button label is now set as a window title.
- In certain circumstances an inactive window could be closed by means of the window decoration, if at the same time there was an active window.
- The `Fsecho` command without an argument on some architectures did not create a blank line in the echo window.
- The `fsopen -x` command returned `1` as its value.
- The `cancel` button in external lists did not restore the status previous to opening the window.
- In an insensitive external list, the text field remained sensitive.

Known problems

The following are the known problems for Finesse V4.

Standard command option

DESCRIPTION: Standard command options corresponding to application resources with tight bindings, for example, the `.borderWidth` resource set by `-bw`, are ignored except the `-name` option.

Default resource settings

DESCRIPTION: Because of varying X resource settings the layout of our examples in the examples directories may differ slightly between different computers.

Display_primes example

DESCRIPTION: The `factor` command is missing for Digital UNIX. Therefore, the `display_primes` examples in subdirectories `sh` and `csh` does not work correctly and should not be started.

For `SunOS-4.1.x` the factor command is located in the subdirectory `/usr/games`. The same is true for Linux if the slackware distribution is used.

`csh` examples on SCO Open Server

DESCRIPTION: The C-Shell on SCO Open Server is not supported by Finesse, since this shell does not know the `eval` command. Therefore all examples in subdirectory `examples/csh` will not work.

perl Installation

DESCRIPTION: The scripts in subdirectory `examples/perl` expect the perl interpreter to be installed on path `/usr/local/bin/perl`. If perl is installed elsewhere, the path component at the beginning of the perl scripts has to be changed appropriately.

Argument too long

DESCRIPTION: Arguments to shell commands are of limited size, which varies from system to system. This may lead to error messages like “Argument too long” when the length of window declarations or window update arguments exceeds this limit. On many systems kernel parameters may be changed to raise this limit.

User guide

4

Introduction

Many UNIX applications are started with a shell script. In many cases, you have a shell dialog at the beginning of the script to gather information such as input and output files that will be passed on to the application. Convenient utilities of X terminals such as selecting items or setting options with a simple mouse click, saving and reading of resource files, etc. are usually ignored since it would require extensive programming to create X based programs to collect such user data.

Finesse is a tool to generate and evaluate windows in shell scripts. With the help of Finesse, you can easily convert dialog parts of existing or future scripts to a mouse-oriented OSF/Motif user interface. You don't need any X programming knowledge since the windows are generated by several commands that are called in the script. These window generation calls are easy-to-use. In a short time, a graphical shell script user interface is available.

How Finesse works

By calling simple Finesse commands you create OSF/Motif windows where any user can enter the necessary input comfortably and quickly. When the input is confirmed, the window entries are returned as shell variables to the shell script. Management of window entries is handled through the usual X application resource mechanism that saves the current values in a file and uses them as default settings for subsequent calls.

One essential Finesse part is the application server that is set up at the beginning of the shell script. It is in charge of the window setup and the associated window and variable management. The shell script communicates with the server by calling several commands as Finesse clients. In this way, Finesse also enables you to execute a complete shell dialog with X windows. The shell script passes to the client parameters that contain the information necessary to create and manage the windows. Like window entries, window descriptions can be saved in a separate file. At the end of the Finesse shell dialog, the application server has to be terminated.

As Finesse is called within shell scripts, you can make use of all the utilities of your shell for window programming. For example, command substitutions such as `` `hostname` `` may be used as values for window entries.

Licensing of runtime applications

Runtime applications can be licensed inside the shell script. You need no longer generate or modify the licensing file `.finesselicence` used so far. Your software contains the shell script `fskeygen`. The command

```
fskeygen scriptname
```

generates a runtime license for application scriptname and writes

```
# FINESSEAPPLICATIONKEY licensestring
```

to the shell script. The script can be run without any license file. Calling `fskeygen` repeatedly for a certain shell script only replaces the license key in the licensing line. The licensing line can be placed anywhere in the script. Existing applications need not be licensed as licensing through `.finesselicence` is possible as well.

The examples directory

Your software release contains a directory named `examples`, which is located either in the /usr/local/finesse directory or in a directory specified by the `FINESSEPATH` environment variable. In the subdirectories `csh`, `perl` and `sh` you will find examples of Finesse based shell scripts you might use as models for your own shell scripts. The examples outlined in the text are partially taken from these shell scripts. In the `examples` directory you will also find to example scripts `examples_sh` and `examples_csh`. They may be used for conveniently calling the examples in the corresponding subdirectories. The subdirectory `app-defaults` contains resource files to the examples. In order to make these resource files available when calling the demos, you may set the X environment variables `XUSERFILESEARCHPATH` or `XAPPLRESDIR`, e.g.

```
setenv XAPPLRESDIR \
/usr/local/finesse/examples/app-defaults
```

Note

Finesse applications are not limited to a certain UNIX shell. At the very start of an application you have to call a shell dependent script to perform the necessary initialization according to shell type.

Finesse differentiates between the following types of scripts:

- `sh` for
  - Bourne shell (`sh`)
  - Korn shell (`ksh`)
  - GNU-Bourne-Again shell (`bash`)
- `csh` for
  - C shell (`csh`)
  - TC shell (`tcsh`)
- `perl` for
  - perl scripts

As `sh` based shells are far more suitable for script programming than `csh` scripts, most of the examples in our text are Bourne shell scripts.

Example: Hello World!

Our first example will be a shell script that creates a window with the famous “Hello World!” message as shown in Figure 2.

```
#! /bin/sh
# sh script initialization
. ${FINESSEPATH-/usr/local/finesse}/fsshinit    #(1)
# Set up application server
Fsopen                                         #(2)
# Create window
Fsdisplay -w "FsWindow -btype o;" \
          -m "Hello World!"                    #(3)
# Terminate application server
Fsclose                                        #(4)
```

Figure 1 `sh/hello_world`

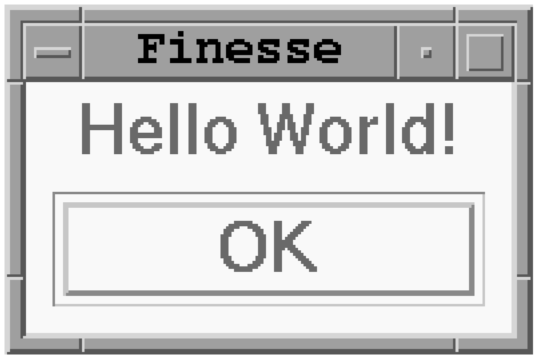


Figure 2 Display from `hello_world`

Finesse initialization

Figure 1 shows the general structure of a Finesse application inside a shell script.

When starting an application, there is a shell type dependent initialization script to read (1). Usually, this will be found on a path set by the variable `FINESSEPATH`, or if that variable has not been set, on the default directory `/usr/local/finesse`. The initialization script among other things defines the shell type (`sh`, `csh` or `perl`) so that subsequent Finesse commands can return window entries to the shell script with the correct syntax.

```
#! /bin/csh

# csh script initialization
if (! $?FINESSEPATH) \
  set FINESSEPATH = /usr/local/finesse
source $FINESSEPATH/fscshinit

# Set up application server
Fsopen                          # (2)

set hello = "Hello World\!"

# Create window
Fsdisplay -w "FsWindow -btype o;" \
          -m "$hello"    # (3)

# Terminate application server
Fsclose                         # (4)
```

Figure 3 csh/hello_world

Csh scripts have to read the file `fscshinit` in the place of `fsshinit`, with the same path preference. Apart from that, the corresponding csh script looks pretty similar. Unfortunately the hello text has to be put into a separate variable, in order that the exclamation mark is handled the same way by all plattforms. Finally the perl version of the hello world example looks like this:

```
#! /usr/local/bin/perl

# perl script initialization
$ENV{'FINESSEPATH'} = '/usr/local/finesse'
  if !$ENV{'FINESSEPATH'};
require("$ENV{'FINESSEPATH'}/fsperlinit");

# Set up application server
&Fsopen(@ARGV);

# Create window
&Fsdisplay("-w", "FsWindow -btype o;",
           "-m","Hello World");

# Terminate application server
&Fsclose;
```

Figure 4 perl/hello_world

How to generate a Finesse dialog

After script initialization, the Finesse application server must be started by the `Fsopen` command (2). Then you can enter any number of further Finesse commands, for example (3). The `Fsdisplay` argument list specifies the design of the window to create. To produce the “Hello World” window in our `sh` example only the following line is necessary.

```
Fsdisplay –w "FsWindow –btype o;" –m "Hello World!"
```

with the options:

`–w` defines the actual window description, which in the simple case above generates a window (`FsWindow`) with an OK button (-btype o) :

`–m` adds a message to the window.

The window description defines the layout of a Finesse generated window. It consists of widgets placed one below the other in the window. For display the window description is passed to the Fsdisplay command. In general, this transfer is possible by means of a command option, redirecting from a file, or with standard input:

Window description as a command option:

```
Fsdisplay –w “window description”
```

Window description out of a file:

```
Fsdisplay –f filename
```

```
Fsdisplay < filename
```

Window description from standard input:

```
Fsdisplay
(terminal input)
ctrl-d
```

Any sequence of Finesse commands quit Finesse by calling `Fsclose` (4). This command is necessary to close the Finesse application server. Otherwise the application server will continue running even after the shell script has executed the last command and quits. A Finesse dialog can be summarized as a sequence of Finesse commands that start with `Fsopen` and end with `Fsclose`. Inside one shell script you can repeatedly open and close the Finesse server, for example when there is a non-interactive interval between two window calls. Schematically, the general order of Finesse commands in a shell script looks like the following:

Finesse initialization by `fscshinit` and `fsshinit`, respectively

```
    Fsopen
    ...
    More Finesse commands
    ...
    Fsclose
```

```
Fsopen
...
More Finesse commands
...
Fsclose
```

and so on.

Option menus

One of the essentials of Finesse is the ability to set window contents from the shell and, vice versa, to return user entries to the shell. In this way you can easily enhance shell scripts demanding at least one interactive input by a mouse-oriented dialog. So in many cases handling, viewing, and interpreting input will be definitely improved.

In example 2, for a simple shell dialog we will look at an application that opens a window, where the user can select one of three different options out of a menu shown in Figure 5. After the choice is confirmed by the `OK` option the `echo` command displays the selected item.

```
#! /bin/sh

. ${FINESSEPATH-/usr/local/finesse}/fsshinit   #(1)

windef="
FsWindow   -btype o  -title 'Option Menu';
FsSeparator;
FsOptionMenu  -label Options:
              -items 'One Two Three'
              -var option=Three;
FsSeparator;"                                   #(2)

Fsopen
Fsdisplay -m "Select Option:" -w "$windef"     #(3)
```

Figure 5 `option_menu` script

The script starts with the sh initialization(1). To make calling `fsshinit` independent from the actual place in the directory tree, the path has again been set by the shell variable `FINESSEPATH`. The next step is to assign the window definition to the `windef` variable (2). In the subsequent Finesse dialog (3), the value of this variable is passed to the `Fsdisplay` command as an argument to the `-w` option. NotC shelle that in calling `Fsdisplay`, the option `$windef` has to be surrounded by double quotes (" ") to handle the whole definition as one argument.

After opening the window, the `Fsdisplay` command is blocked so that the user can choose the desired option: `One`, `Two,` or `Three` in the menu. Upon clicking the `OK` button the `Fsdisplay` command closes and returns to the calling shell script. The shell variable `option` now contains the chosen value, which is displayed by the `echo` command. As usual the Finesse dialog opens and closes with the `Fsopen` and `Fsclose` command, respectively.

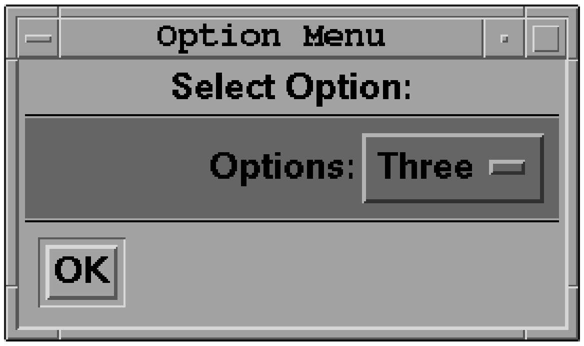


Figure 6 display from `option_menu`

Window declaration

In contrast to Figure 1, the window definition is rather long in Figure 5. Therefore we recommend assigning it to a separate shell variable. This assignment can take place at the beginning of the shell script, making it possible to separate declaration and actual dialog.

The `windef` definition of Figure 5 shows the typical structure of a window declaration. Starting with the keyword `FsWindow`, it contains the declaration of any number of widgets with widget dependent arguments, separated from each other by a semicolon (;). In our example, three widgets have been declared: two separators by `FsSeparator`, and an option menu by `FsOptionMenu`. The arguments used are determined by the widget type.

In the window, widgets are displayed below each other according to their order. Above the first user-defined widget, there is a label where you can put any text by means of the `Fsdisplay -m` option, such as, ”`Hello World`” or ”`Select Option`”. In the lower part of the window you may find standard push buttons like, for example, the `OK`-button.

Window declarations may contain comments. The comment character is `#`. All characters following `#` on the same line are ignored.

Widget declaration

Arguments for widget declaration as well as Finesse commands have the form:

–keyword value

Each widget possesses a number of keywords that are listed in the "User Commands" chapter. *Value* is the character string up to the next word separator (space, tab, carriage return) or up to the semicolon at the end of the widget declaration. Every *value* containing spaces or special characters (e.g., semicolon) can be surrounded by single quotes to disable the special meaning of these characters and make them part of a *value*. In our example, the character string `'Option Menu',` containing a space, is marked as one value by single quotes. Without quotes only the term `Option` would be interpreted as a value for the keyword `-title`; the following term, `Menu`, would be read as the next keyword and therefore cause the error message:

```
Window declaration syntax error: Menu
```

The value of the `-items` keyword regularly has to be put between single quotes, as it typically contains several entries that are separated themselves per default by spaces, tabs, or carriage returns.

The `-btype` keyword allows placement of standard push buttons in the bottom window area. Each one of these push buttons is marked by one single letter; for example, the `OK` button is referenced by the letter `o`.

The second `FsWindow` argument `-title 'Option Menu'` in example 2 replaces the heading Finesse with Option Menu in the window title bar. The option menu declaration has following arguments: `-label Options` which defines the menu name. The second argument is, `-items 'One Two Three'` which defines the available options the user has to select from. The third argument is, `-var option=two`, which specifies the window variable `option`, and the assignment `=Two,` the default value shown in the window display.

The `-items` and `-var` arguments are required in every `FsOptionMenu` declaration; the `-label` argument is optional.

Variable return

The `Fsdisplay` command terminates when the entries in the window are confirmed with `OK`. The window will close and the variable declared with `-var` will be set in the shell script. Subsequently, the values may be evaluated, for example in a subsequent echo command. Details about variable setting in the shell script will be explained in the section on initialization files.

Some widgets, like `FsOptionMenu`, require a `-var` declaration as an argument, because otherwise setting an option in the window would make no sense. Arguments such as `FsSeparator` don't have a `-var` option, because they will not return information.

Window declaration using the C shell

For the sake of comparison, here we show the C shell version of our example, which has a slightly different form:

```
#! /bin/csh

if ( ! $?FINESSEPATH ) \
set FINESSEPATH = /usr/local/finesse
source $FINESSEPATH/fscshinit

# Define window elements

set fswin = "FsWindow -title 'Option Menu' -btype o;"
set fssep = "FsSeparator;"
set fsoptmen = "FsOptionMenu -label Options:"
set fsoptite = "-items 'One Two Three' -var
option=Two;"
set fsopt = "$fsoptmen $fsoptite"

# windef has to be written as one line to support
# all cshs

set windef = "$fswin $fssep $fsopt $fssep"

Fsopen
Fsdisplay -m "Select Option:" -w "$windef"
Fsclose

echo "Option selected: $option"
```

Figure 7 `csh/option_menu` script

Apart from the shell dependent initialization with `fscshinit` mentioned above the essential difference to the Bourne shell example consists in the fact that the window elements first are defined separately and combined in the shell variable windef to build the window definition afterwards. The reason for this somewhat clumsy declaration is the difficulty that some csh shells on different platforms behave differently in handling **newline** characters in a window declaration. So this kind of declaration is sort of the smallest common denominator for the shells. Chapter **Window Hierarchies** shows a `csh` example with different window declarations that partially are made directly, similar to `sh` scripts.

Text fields

In situations similar to that in the previous section where window input is restricted to just a few values, an option menu will be the favored widget. Input can be minimized to a mouse click. Incorrect typing as a source of input errors is ruled out. When input consists of entering text, usually any value can be given. In this case, a text widget is the appropriate solution. Figure 8, asks for the user's name, as shown in Figure 8, to welcome him with a modification of the "Hello World" example in a second window.

```
#! /bin/sh

. ${FINESSEPATH-/usr/local/finesse}/fsshinit

windef="
FsWindow  -btype o;
FsLabel   -label 'Please enter your name:';
FsText    -label Name: -var name;
FsSeparator;"                                #(0)

Fsopen

Fsdisplay -w "$windef" -m "Hello!"           #window 1

Fsdisplay -w "FsWindow -btype o;"\
          -m "Hello $name"                   #window 2

Fsclose
```

Figure 8 `hello_name1` script

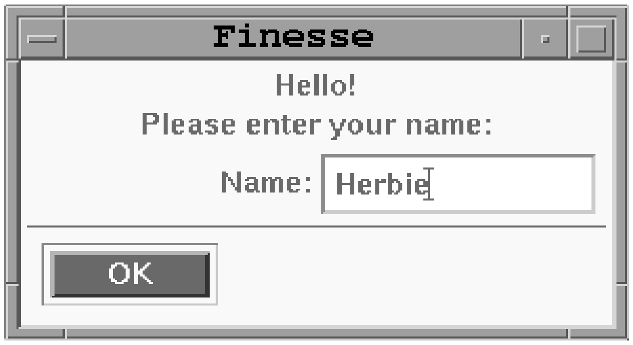


Figure 9 Entering text

Labels and texts

The window declaration contains two widgets that have not been used so far:

- `FsLabel` to write text, notes, etc. to the window
- `FsText` with labeled text widget to receive user input

The widget `FsLabel` does not allow user interaction. Its only argument is `-label` to specify the label text. The value of the `-label` keyword consists of several words separated by blanks and surrounded by single quotes.

`FsText` needs the `-var` argument as a minimum requirement to declare the shell variable receiving the text input, because user input that cannot be returned to the shell script is useless. Optionally with the `-label` argument you can label the text widget. If the label specification is missing the variable name will be set as the label.

To simplify input, the following additional commands help to move and to edit in the text widget:

Table 2 Movement and editing commands

Key	Function
ctrl-a	Move to the beginning of the line
Mod1-b	Move backward one word
Mod1-f	Move forward one word
ctrl-d	Delete next character
Mod1-d	Delete next word
Mod1-delete	Delete previous word
ctrl-e	Move to the end of the line
ctrl-j	Delete up to the beginning of the line
ctrl-k	Delete up to the end of the line
ctrl-u	Delete line

Exit status of Finesse commands

Similar to most UNIX commands, Finesse commands return an exit status that can be checked by the status variable `$?` in the Bourne shell and `$status` in the C shell. Successful completion of a Finesse command returns the value 0. For example, after the `OK` button has been clicked, the `Fsdisplay` command returns 0 for its exit status if it has been able to open the declared window and to return the variables correctly to the shell script.

```
Fsclose; exit 1
```

In the above example the statement closes the current Finesse dialog and the shell script if the status value is different from 0.

Push buttons

Example Figure 8 is expanded in example Figure 10, by requiring that the user actually enters a name in the text widget before confirming the input with `OK`. Otherwise a `while` loop in the shell script keeps opening the window until either an entry is given or the `Abort` button is pressed.

```
#! /bin/sh

. ${FINESSEPATH-/usr/local/finesse}/fsshinit
winnam=namewindow
windef="
FsWindow     -btype oxa -name $winnam;
FsLabel      -label 'Please enter your name:';
FsText       -label Name: -var name;
FsSeparator;"                                    # (0)

Fsopen

Fsdisplay -w "$windef" -m "Hello!"              # (1)
if [ "$fsbutton" != "o" ] ; then
Fsclose; exit
fi

while [ -z "$name" ]                             # (2)
do
Fsdisplay -n $winnam -m "Name is missing..."     # (3)
if [ "$fsbutton" != "o" ] ; then
Fsclose; exit
fi
done

Fsdisplay -w "FsWindow; FsPushButton -label \
              Goodbye;" -m "Hello $name"        # (4)

Fsclose
```

Figure 10 sh/hello_name2

Predefined push buttons

The `FsWindow` declaration in the variable definition (0) of has the argument `-btype oxa`. By means of `-btype`, predefined push buttons can be put in the bottom region of the window (Figure 10). Push buttons trigger a certain action.

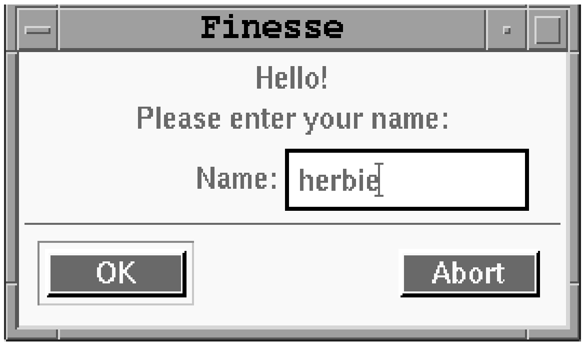


Figure 11 Text input with confirmation

With predefined push buttons the labels are fixed. For example:

`o` creates a button labeled `OK`

`a` creates a button labeled `Abort`

`x` generates empty space between adjacent push buttons

Apart from the label the action resulting from a click on the predefined push button is also fixed. For example:

`OK`

will close the corresponding window and return the window variable values to the shell script.

`Abort`

also will close the window, but variables are not returned in this case.

How to declare your own push buttons

Declaring the widget `FsPushButton` allows you to generate your own push buttons where features and actions can be defined within certain limits. In the second window has a button labeled `Goodbye` in the place of the `OK` button from the previous example. Labeling of a self-defined push button is possible through the command:

`FsPushButton -label` *argument*

If `-label` is missing, `OK` is the default value.

As the `FsPushButton` declaration of this window has no further arguments, the window is closed by default similar to the `OK` button. If the window had variables these would additionally be delivered to the shell script.

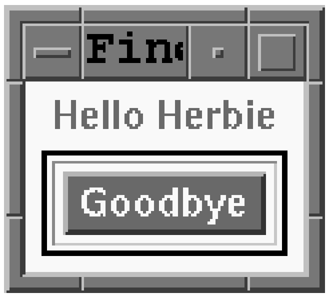


Figure 12 Push button used in hello_name2

The fsbutton variable

To be able to distinguish in the shell script which button has been pressed, the additional variable `fsbutton` will be set on successful completion of the `Fsdisplay` command, i.e., if the exit status is 0. The `fsbutton` value identifies the widget causing the return. Currently, returning to the shell script by user interaction is possible only through push buttons.

A predefined push button assigns the corresponding character of the `-btype` call to `fsbutton`. By checking the `fsbutton` variable for the value a, for example, you can decide whether the `Abort` button has been clicked. A push button defined with `FsPushButton` assigns the value of the `-label` argument to `fsbutton` by default. The `FsPushButton` option `-fsbutton` allows defining your own return values for the corresponding push button. If the `-label` as well as the `-fsbutton` argument are missing, `fsbutton` will get the return value `OK`.

Repeated window calls

is a typical case of an application where user input is checked before processing. When an error is detected the application re-requests user input.

On closing the window with `OK`, which has been opened with statement (1), statement (2) checks whether the string “`$name`” has length zero. If the `name` variable is empty, the `while` loop executes and reopens the window defined (1). The process is repeated until the name variable has a non zero value or the script is closed with `Abort` and subsequent `Fsclose`. The second window will only be opened if a name has been given in the first window.

The same window can be opened repeatedly by adding the argument `-name` *name* to its `FsWindow` declaration. The window can then be referenced by `Fsdisplay` with the option `–n` *name*:

```
Fsdisplay –n name
```

In contrast to this, a repeated call of the form

```
Fsdisplay –w "windef"
```

with the same window definition would each time open a new window looking like the previous one.

In a csh script line (2) of would have the form

```
while ( "$name" == "" )
```

A term like `while (! $?name)` would be incorrect, because on `OK, Fsdisplay` actually sets the window variable name as a shell variable, but without a value if the window entry is missing.

Radio and check boxes

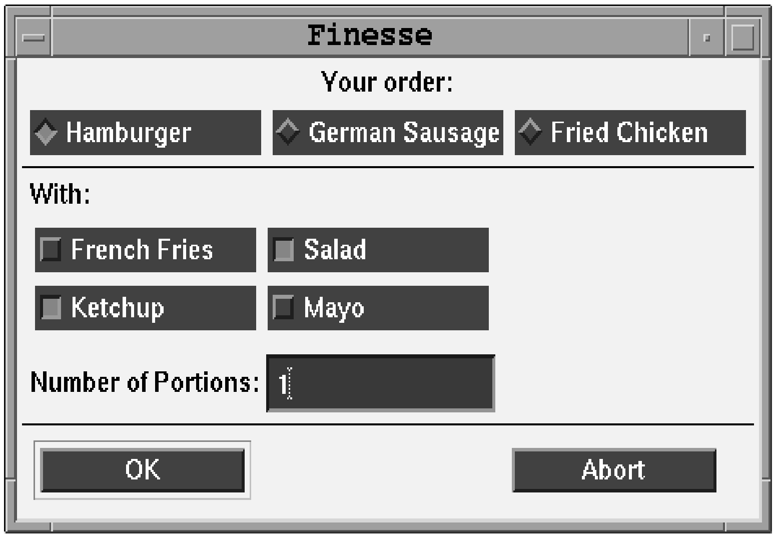


Figure 13 display from `snack` script

Radio boxes

"Option menus" section on page 24 has already shown an example for a user selection from a predefined menu. Radio boxes are an alternative way to achieve selecting exactly one out of several items. Figure 12 shows a hypothetical user interface for a snack bar with terminal input that contains an example of a menu with three options in its top part.

Whether you will use the option menu or the radio box depends on the number of options and the window design (apart from your personal preferences). Radio boxes have the advantage that all the possible options are visible at once, whereas option menus with an average number of options need less window space. With a large number of options, use a list widget (`FsList`). shows the shell script for the Figure 12 window. The keyword `FsRadio` starts a radio box declaration. Following the keyword `-items` the possible options are specified. As with option menus, the keyword `-var` is followed by the name of the variable containing the chosen option.

Finally, the keyword `-inputsep` defines the slash / as a separator for the radio box items, since spaces are not applicable as separators because of the option `German Sausage`.

```
#! /bin/sh

. ${FINESSEPATH-/usr/local/finesse}/fsshinit
```

```
windef="
 FsWindow -btype oxa;

 FsLabel  -label 'Your order:';

 FsRadio  -items
           'Hamburger/GermanSausage/FriedChicken'
          -var order=Hamburger
          -inputsep '/';

 FsSeparator;

 FsCheck  -items 'French Fries/Salad/Ketchup/Mayo'
          -label With:
          -inputsep '/'
          -outputsep '/'
          -var with
          -nrows 2;


 FsText   -label 'Number of Portions:'
          -var num=1
          -texttype int;

 FsSeparator;"

Fsopen -o 1 "$@"

Fsdisplay -w "$windef"

if [ "$fsbutton" != "a" ] ; then
echo Your order:
echo $order
echo with:
echo $with | tr '/' '\012'
sleep 5
fi

Fsclose
```

Figure 14 script `snack`

In a radio box you only can select one item. As soon as one of the items is clicked “on”, any previously clicked item is set “off”. In contrast to this a check box allows selecting any item independently from all the others. The items of a check box can be browsed like a checking list and can individually be set or not. The bottom part of Figure 12 shows a check box where any add-on combination is to choose. Visually you can distinguish a check box from a radio box by its square indicators (instead of diamonds).

You specify a check box with `FsCheck` in . When specifying variables, you can set several items as a default, which is different from `FsRadio`.

Checking on text entries

 shows how to check user input in the shell script before processing and how to recall it for correcting if necessary. In some cases it is possible to execute the checking before closing the window instead of in the shell script. In case of error the window stays open; the echo window shows a message line containing the incorrect entry and correction can take place immediately.

To allow the checking of entries Finesse needs information on admissible input. In , for the text widget labeled "`Number of portions:`" only integer entries greater than zero are valid. Therefore, the `FsText` specification in the window declaration contains the argument `-texttype int[1,]. int` only admits integers as an entry, with optionally a + or − sign. The first number in brackets gives a lower limit for the entry. A second number in the bracket, after the comma, could set an upper limit for the number of portions. Apart from `int,` there are several other keywords to allow limited input only. For a complete listing of options see the "FsSeparator" section on page 98.

Redirecting standard output

At the end the above script contains some `echo` commands that summarize the orders. Their output, however, does not appear on the terminal as usual but in the echo window itself. This standard output redirection is a result of the `-o 1 Fsopen` option at the beginning of the Finesse dialog. Option `-o 1` redirects standard output only, `-o 2` standard error only, `-o 3` standard output and standard error. Thus it is possible to handle all input and output of a shell script in Finesse windows. Redirection is done automatically without any user interaction. Furthermore, it is limited to the Finesse dialog between `Fsopen` and `Fsclose`, i.e., `echo` commands after `Fsclose` will appear in the terminal window.

In csh type scripts options `-o 2` and `-o 3` are not different from each other as this shell type does not support separate redirection of standard output and standard error. Option 1 is different from option 2 and 3 in that with 1, Finesse server error output is not redirected to the echo window. This can be important especially in cases where there are errors in the settings of X resources by means of the `-r Fsdisplay` option (see chapter **Setting X**

Resources). Furthermore, in csh scripts you have to make sure with every command that standard output and standard error are redirected to the echo window. This can be done by using the Finesse environment variable `$FSSTDOUTFILE`. The callbacks example in chapter **Window States** shows how to redirect standard output in a csh script.

Lists

Mainly lists are used to offer a choice out of a given lot of items. List entries appear one below the other. The list is displayed in a window segment that shows the complete list or parts of it via scroll bar.

Selecting and displaying elements

Typically, list elements are defined through the value of the `-items` option. Additionally, more items can be included through resource files or when refreshing a list. The argument `-nvisible` *num* determines how many items can be viewed at one time. The `-mode` option value defines whether only one or several, even non adjacent, items can be selected.

A list can be displayed inside the main window (default) or in a separate window (`-include no`). In the latter case the main window contains a composite widget that consists of a push button on the left hand side and a text widget on the right hand side. Clicking at the push button opens the separate list window. After confirming your selection the items are transferred to the text widget and separated by blanks. If several items can be selected the text widget is insensitive. In this case the text widget is only intended to be a help for orientation in the main window. If, on the other hand, you have set `-mode single`, the values of external lists can be set directly inside the text widget.

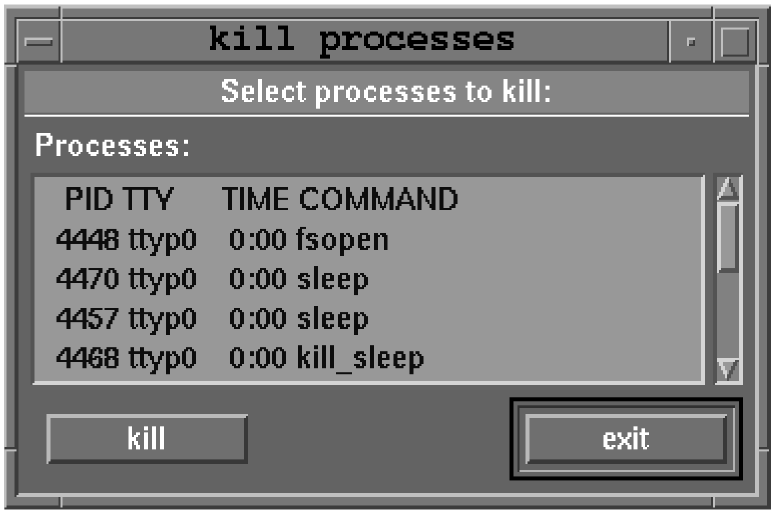
In an external list window all items can be selected and deselected by two special push buttons. If there are more than one list, the push button label, which is repeated at the top of the list window, shows which of the external list windows belongs to a certain text widget in the main window.

Items selected by a click are highlighted. You can deselect any highlighted item by clicking it once again. By clicking while pressing the **ctrl** key you can select any number of non adjacent list entries.

Elements on a list are separated by the `-inputsep` option value in the `-items` argument. Default are the separators space, tab, and carriage return. This is the reason why usually the `-items` option argument has to be surrounded by single quotes. When the selected list items are returned, they are separated by the `-outputsep` option value. This separator can be a single character or a character string. The default is separation of the list elements by a space.

Example program

The `kill_sleep` script shows a list of processes connected with the current terminal.
#



```
PID TTY     TIME COMMAND
4448 ttyp0  0:00 fsopen
4470 ttyp0  0:00 sleep
4457 ttyp0  0:00 sleep
4468 ttyp0  0:00 kill_sleep
```

Figure 15 display from `kill_sleep`

```
# Demo script for killing processes
# For demo purposes two sleep processes
# are created that may be safely killed

. ${FINESSEPATH-/usr/local/finesse}/fsshinit

# generate two processes for killing
#

sleep 60 &
sleep 60 &

# Declare window with list of processes
#

nl="
"
psitems="`ps`"

windef="
  FsWindow   -name killwin -bdefault exit
             -title 'kill processes';
  FsSeparator;
  FsList     -label Processes:
             -items '$psitems'
             -nvisible 5
             -mode multiple
             -inputsep '$nl'
             -outputsep '$nl'
             -var killlist
             -expert kill;
FsPushButton -label kill -fsbutton k
             -name kill;
FsPushButton -type x;
FsPushButton -label exit -nrows 1
             -name exit;"

# Open server and display window
#

Fsdisplay -w "$windef" \
          -m "Select processes to kill:"

# Kill selected processes
#

if [ "$fsbutton" = "k" -a -n "$killlist" ] ; then
  echo "$killlist" |
   awk '$1 !~ /PID/ {print $1}' |
   xargs kill
   sleep 5     # time to display kill message
fi
Fsclose
```

Figure 16 `kill_sleep` script

Default buttons

The default button of a window can be chosen by the `FsWindow` option

`-bdefault` *name*

name designates the name of the push button activated by pressing **return** inside the window. If the -`bdefault` option is missing, the leftmost push button within the row that was the last to be declared in the window definition is set as default. Setting a default only has an effect when the window is opened for the first time as the default button changes according to user input. In the example above, `exit` is set as default button to make sure that **return** after an erroneous selection does not create negative results.

Apart from the `kill` button, processes can be deleted by double clicking in the list. The option `-expert kill` connects double clicking with the push button named `kill`. So double clicking activates the push button specified. If the `-expert` option is omitted, double clicking activates the default button of the window, if it exists. In our example this would have been the `exit` button.

Modyfying list entries

This list has been generated as an argument to the `FsList -items` option by the command `ps`. The separator between items is the new-line character specified in the shell variable `nl`. This has the effect that each of the processes shown gets a line of its own, similar to terminal output. Those processes that are associated with selected lines are deleted by the `kill` command on pushing the `kill` button. In order to get processes for the purpose of testing the `kill` command without running into problems, two background sleep processes have been created at the beginning of the shell script.

A special property offered by a list is the possibility to change the list while running an application. You want to be able to replace, delete or insert any list item at any place. This is possible in Finesse through additional keywords when refreshing the list items with the `Fsdisplay -v` option. Table 3 gives syntax and meaning for each of these keywords:

Table 3 Updating list elements

value	action
a`[select]`	select list item *a*
`*[select]`	select all list items
a`[add]`	insert item *a* at the end of the list
a`[delete]`	delete list item *a*
`*[delete]`	delete all list items
a`[addselect]`	insert and select item *a*
a`[unselect]`	deselect list item *a*
`*[unselect]`	deselect all list items
a`[topinsert]`	insert item *a* at the top of the list
a`[replace]`*b*	replace list item *b* by item *a*
a`[after]`*b*	insert item *a* after list item *b*
a`[before]`*b*	insert list item *a* before list item *b*
a	insert item *a* if not existent and select it

A sequence of several items with the desired keywords can also be a valid list variable value. Separator is again one of the characters defined in `-inputsep` or the default.

Examples

The command

```
Fsdisplay -n name -v 'list=tick[replace]tack/toe[select]'
```

replaces the list item `tack` by the element `tick` in the window called `name` and at the same time selects the list item `toe`, if `-inputsep '/'` has been set in the `FsList` declaration in name.

The command

```
 Fsdisplay -n name -v 'list=*[delete] new[add]'
```

deletes all the current list items and includes the new item `new`.

File selection

generates a window containing the file selection widget `FsSelectionText` shown in . You will typically use this widget when a file name has to be given. The widget is composed of a push button and a text widget where the file name can be entered. On clicking the push button opens a file selection box where the desired file can be selected by a mouse click. Confirming the file selection in the selection box automatically transfers the file name to the text widget.

```
#! /bin/csh

if ( ! $?FINESSEPATH ) \
 set FINESSEPATH = /usr/local/finesse
source $FINESSEPATH/fscshinit

set tmpfile = /tmp/$$.tmp

Fsopen

cat << EOT > $tmpfile
FsWindow        -title 'Example File Selection'
-btype oxxa ;
FsSeparator;
FsSelectionText -var filename
-label 'Select file:';
FsSeparator;
EOT

Fsdisplay -f $tmpfile
if ("$fsbutton" != "o") then
Fsclose
/bin/rm -f $tmpfile
exit 1
endif

Fssave

if ( $filename != "" ) then
if ( -f $filename ) then
set longlist = `ls -l $filename`
Fsecho -r 3 -c 80 -t "Long listing of $filename"\:
Fsecho \ \ $longlist
else
Fsecho File not found.
endif
else
Fsecho No file selected.
endif
cat << EOT > $tmpfile
FsWindow -btype o;
FsSeparator;
```

```
EOT

Fsdisplay -m "OK to quit." < $tmpfile
/bin/rm -f $tmpfile

Fsclose
```

Figure 17 script `file_selection`

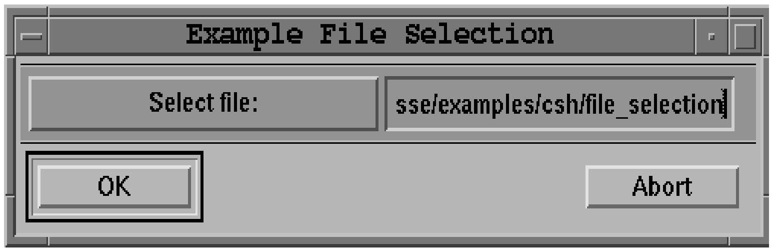
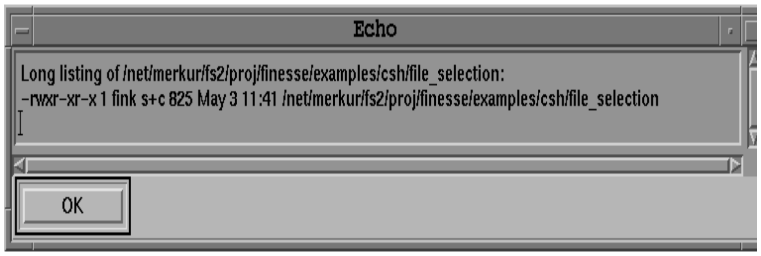


Figure 18 Display from `file_selection`

The echo window

The selected file name of will be used to create a long listing of the file (2). This is passed as an argument to the Finesse command `Fsecho` by way of a shell variable (3):

```
Fsecho \ \ $longglist
```



```
Long listing of /net/merkur/fs2/proj/finesse/examples/csh/file_selection:
-rwxr-xr-x 1 fink s+c 825 May 3 11:41 /net/merkur/fs2/proj/finesse/examples/csh/file_selection
```

Figure 19 Output from `file_selection` script

`Fsecho` opens its own "message window" displaying its arguments in the window similar to the UNIX echo command as shown in . On repeated calls the arguments will be written into the window one below the other.

The echo window has an OK button to close the window any time. In that case the current window contents will be cleared so that subsequent `Fsecho` calls only display new messages.

You can use `Fsecho` to display the whole contents of a file as well. The first `Fsecho` argument being `-f`, following arguments are taken as file names and their contents are written into the echo window:

`Fsecho -f` *filename*

Window description from a file

As an alternative way for transferring the window description, the file selection example does the passing not by means of the `-w` option but out of the file /tmp/$$.tmp. This will be achieved either with the `-f` option, as in the first window (1),

```
Fsdisplay -f /tmp/$$.tmp
```

or redirecting the input, as in the second window (4):

```
Fsdisplay -m "OK to quit" < /tmp/$$.tmp
```

Interactive window description

If the `Fsdisplay` function in the shell script does not contain a window description, you can give this description interactively by terminal input. By this method special windows can be created at runtime as Figure 20 shows.

```
#! /bin/sh

. ${FINESSEPATH-/usr/local/finesse}/fsshinit
Fsopen

# Define window via terminal input
echo "Insert your window definition (^D to end):"
Fsdisplay -m "Here is what you defined:"

Fsclose
```

Figure 20 `my_window` script

How to save window parameter settings

The `Fssave` command saves every window setting registered by the application server so far to a resource file in the `.finesse` subdirectory of the home directory. The file name is composed from the calling shell script name and the extension `.rsc`. For example, if the shell script name is `file_selection` the window settings are saved to the file `file_selection.rsc`.

Inside the resource file values are saved in the form

```
variable: value
```

Default settings when starting the application

Analogous to the saving of window settings, the default resource file name on calling a shell script named `xyz` is the file `xyz.rsc` in the `.finesse` subdirectory of the home directory, if a different resource file has not been explicitly specified for searching and reading. The last values saved are available to the application server and are set as current values when the application is restarted through the use of the file:

`$HOME/.finesse/`*scriptname*`.rsc`

Window states

In all our previous examples the `Fsdisplay` command not only had to pass the window declaration to the Finesse server but also had to wait for the confirmation of window entries and return them as shell variables to the shell script. Further processing of the shell script had been interrupted until necessary input had been entered and the window could be closed again.

In some cases, though, displaying a window does not need any user input or confirmation, for example, when some information is to be displayed graphically. shows such a situation: the shell script counts up all natural numbers one at a second, checks for indivisibility, and shows the number and the checking result in the window in .

```
#! /bin/sh
#
. ${FINESSEPATH-/usr/local/finesse}/fsshinit
#
# Extend path for SunOS 4.x and Linux
#
PATH=$PATH:/usr/games

# Definition

i=1; yesno=No
windef="
  FsWindow -name primewin;
  FsSeparator;
  FsText  -label Number:
          -var number==$i;
  FsRadio -label Prime:
          -var prime==$yesno
          -items 'Yes No';"

# Test

test_prime() {
  test_eq $1 `factor $1`
}
test_eq() {
  if [ -z "$3" ] ; then
   echo Yes
  elif [ "$1" -eq "$3" ] ; then
   echo Yes
  else
   echo No
  fi
}

# Update
```

```
Fsopen "$@"
while :
do
  Fsdisplay -w "$windef" -n primewin -s touch \
            -v number=$i -v prime=$yesno
  if [ "$fsbutton" = "a" ] ; then
   break
  fi
  sleep 1
  i=`expr $i + 1`
  yesno=`test_prime $i`
done
Fsclose
```

Figure 21 sh/display_primes

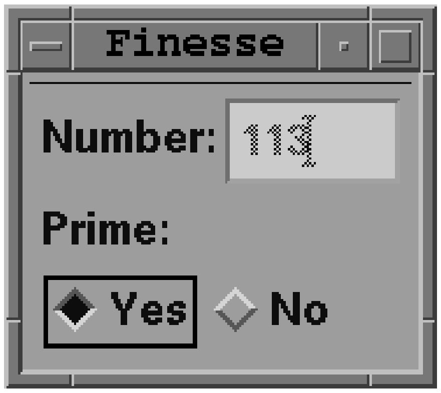


Figure 22 Automatic window update

Number and checking result are displayed by repeatedly calling the window with the `Fsdisplay` command:

```
Fsdisplay -w "$windef" -n primewin -s touch\
          -v zahl=$i -v prime=$yesno
```

The command contains the `-w` as well as the `-n` option. In these cases first the window named in the `-n` option is looked up. If existing, this window is displayed, otherwise the `-w` option definition is used to create the window. In our example, on the first call the `$windef` definition is read because so far there is no window named `primewin`. Subsequent `Fsdisplay` calls ignore the `-w` command line option and refer directly to the window named `primewin` created during the first call.

Window states

A Finesse window always is in one of three possible states: `input`, `close`, or `touch`. For a window newly generated by a

`Fsdisplay` call, the default state is `input`, and the window will be waiting for user interaction. Window states can change through `Fsdisplay` calls or with certain buttons. For example, clicking the `OK` button in the "`Hello World`" example changes the window state to `close`. The `touch` state keeps the window open but will not allow any input.

A window can be transferred from one possible state to any other. This can be done by means of the `Fsdisplay` command or of push buttons. The `Fsdisplay` command allows setting a window state by the `-s` option, push buttons by the `-winstat` option shown in Figure 23.

All the same, changing the state `input` by the `Fsdisplay` command is impossible, as the window is waiting for user interaction and disabling further shell script commands during this time. Vice versa, state changing by push buttons is possible in the input state only. An `Fsdisplay` call without an `-s` argument puts the corresponding window to state `input.`

In the example above, the window is called repeatedly in `touch` state. In this way the window is opened but immediately left again. The window does not admit any user input, which is symbolized by the one-way sign mouse cursor and shadowing of the window. Calling a window in `touch` state allows immediate execution of subsequent commands, for example, repeatedly refreshing window entries.

Refreshing window entries

Refreshing window entries is triggered by the `Fsdisplay -v` option. The argument is the variable to modify, followed by its new value. This is specified with the same format as in the window declaration:

name=value

`Fsdisplay` can have several arguments to modify different window entries in one call.

Priority of window settings

So far in our examples we have presented several possibilities to set window parameters: by simple (=) or by absolute (==) variable assignment in the window declaration, by the `Fsdisplay -v` option, or by means of the resource file. Absolute variable settings in the window declaration have the highest priority. They serve to set values appearing as a default every time the application is called. If a variable does not have an absolute setting in the window declaration, Finesse will search for a −v option variable assignment. If that is missing as well, the resource file entry has the next priority. A simple variable setting in the window declaration has lowest priority. Lacking this one, too, a default value will be assumed.

Window hierarchies

Our previous example has shown how to modify window entries from the shell script with the help of the `Fsdisplay` command state modification. The following shell script demonstrates the possibilities to change window states by push buttons:

```
#! /bin/csh
#
if (! $?FINESSEPATH) set FINESSEPATH =
/usr/local/finesse
source $FINESSEPATH/fscshinit

Fsopen -o 1

# csh window definition syntax depending on
operating system

if ( "`uname`" == "SCO_SV" ) then
 echo "csh not supported. Use 'tcsh' or convert to
'sh'."
 exit 1
else if ( "`uname`" == "OSF1" ) then
 set a = "FsWindow -name callback "
 set aa = "-title 'Example Shell Callback';"
 set b = "FsPushButton -label Button1 "
 set ba = "-fsbutton b1 -winstat touch;"
 set c = "FsPushButton -label Button2 "
 set ca = "-fsbutton b2 -winstat touch;"
 set d = "FsPushButton -label Button3 "
 set da = "-fsbutton b3 -winstat touch;"
 set e = "FsPushButton -label Button4 "
 set ea = "-fsbutton b4 -winstat touch -nrows 2;"
 set f = "FsRadio -items '1 2 3' -var num=1 "
 set fa = "-fsbutton 123 -winstat touch;"
 set g = "FsRadio -items '3 6 9' -var enum=3 "
 set ga = "-label 'times 3' -fsbutton 369 -winstat
touch;"
 set windef = "$a$aa $b$ba $c$ca $d$da $e$ea $f$fa
$g$ga"
else
 set windef = "\
  FsWindow -name callback -title 'Example Shell
Callback';\
  FsPushButton -label Button1 -fsbutton b1 -winstat
touch;\
  FsPushButton -label Button2 -fsbutton b2 -winstat
touch;\
  FsPushButton -label Button3 -fsbutton b3 -winstat
touch;\
  FsPushButton -label Button4 -fsbutton b4 -winstat
touch\
       -nrows 2;\
  FsRadio -items '1 2 3' -var num=1\
          -fsbutton 123 -winstat touch ;\
  FsRadio -items '3 6 9' -var enum=3\
          -label 'times 3' -fsbutton 369 -winstat
touch;"
endif

if ( "`uname`" == "AIX" && "`uname -v`" == "4" ) then
 set windefx = ($windef:x)
 Fsdisplay -w "$windefx" -m "Push Callback Button:"
else
 Fsdisplay -w $windef:q -m "Push Callback Button:"
endif

while (1)
 switch ($fsbutton)
  case b[1-4]:
   set sub = "FsWindow -name s$fsbutton -btype o;"
   echo Shell callback $fsbutton... > $FSSTDOUTFILE
   Fsdisplay -w "$sub" -n s$fsbutton\
             -m "Displaying subwindow $fsbutton..."
   breaksw
  case o:
   Fsdisplay -n callback
   breaksw
  case 123:
   set enum = `expr $num \* 3`
   echo Shell callback $fsbutton... > $FSSTDOUTFILE
   Fsdisplay -n callback -v enum=$enum
   breaksw
  case 369:
   set num = `expr $enum / 3`
   echo Shell callback $fsbutton... > $FSSTDOUTFILE
   Fsdisplay -n callback -v num=$num
   breaksw
  default:
   break
 endsw
end

Fsclose
```

Figure 23`callbacks` script

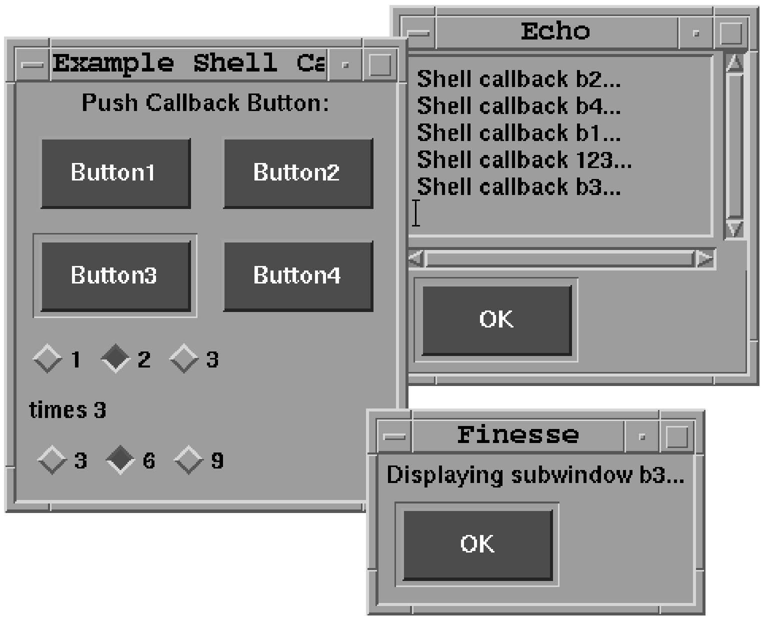
Platform dependent window definitions using csh

Before we can describe shell callbacks we want to talk about the platform dependent window declaration of this csh script. Excepting the SCO Open Server whose C shell has no built-in eval command and therefore isn't supported we have to differentiate between three cases. The first decision concerns DEC OSF1 (Digital Unix) where the window declaration, as previously shown, has to be in one long line without any <NEWLINE> characters. For all other platforms the window declaration of example 10.1 can be written directly to a variable (`windef`), similar to sh scripts. The declaration, however, stretching over several lines, has to continue with a backslash `\`, which is not very nice but ever-present in the C shell, as the C shell admits double quotes (") in one line only. Surrounded by double quotes, the string `\`<NEWLINE> is converted to a real <NEWLINE> character, i.e., this <NEWLINE> character stays in the windef declaration. So subsequent windef referencing with "$windef" would again cause a syntax error. On the other hand the -w option argument has to be transferred as one single argument when calling Fsdisplay. With the exception of AIX-4.1 this dilemma can be solved by using the C shell :q modifier that skips the " " syntax check but all the same puts $windef in double quotes. With AIX-4.1 the window declaration first has to be saved to a ++ word list ++ as a contiguous character string. Otherwise it would be separated into its arguments at the <NEWLINE> characters when handed over to Fsdisplay and so cause a syntax error.

Shell callbacks

The example is the prototype of a window where certain actionsare executed in the shell script without closing the window, according to the button clicked. So multiple connections between widgets can be realized. Similar to X-window terminology these actions can be termed "shell callbacks". Shell callbacks can be linked to push buttons as well as to menu fields `FsRadio`, `FsCheck`, and `FsOptionMenu`. The main window consists of four push buttons displayed in two rows by means of the `-nrows` option (Figure 23) and two radio menus. Each one of the elements has the option `-winstat touch` which means that on clicking this button the window is set to the `touch` state and the corresponding `Fsdisplay` command returns to the shell script. Here, according to the button, a certain message is written to the echo window and the corresponding action is executed. In the case of `FsPushButton` a subwindow will open, in the case of a radio menu the other one will be modified. In the first case the main window is reactivated on pushing `OK`, in the second case this happens at once. After that further callbacks can be started, or the loop can be exited via window decoration by the default switch.

Widgets with the option `-winstat touch` on activating return to the shell script. Window values are returned or not according to the `-data` option. `FsPushButton` widgets return the value set in the `-fsbutton` variable, otherwise the button label. If the `-fsbutton` option has been set with a menu, the variable `fsbutton` contains the corresponding value when returning. Otherwise the label of the menu field concerned is returned in the variable `fsbutton`. So in the script you can check flexibly either the menu or a single menu field.



```
Shell callback b2...
Shell callback b4...
Shell callback b1...
Shell callback 123...
Shell callback b3...
```

Displaying subwindow b3...

Figure 24 Window hierarchy used in the `callback` script

Window hierarchy

Any action may be executed as a shell callback. In our example a message is written to the echo window to show which push button has been clicked. Additionally a second window will open. This, too, could contain push buttons triggering shell callbacks and opening more windows, and so on. By the `-winstat` option a complete window hierarchy can be built as seen in Figure 23. Only one of these windows can be sensitive at a given time, typically the lowest one in the window hierarchy. The ability to build a window hierarchy is useful for designing submenus of a main window.

Modality of dialogs

Opening a submenu by one of the FsPushButton fields causes the input focus to change to this window. Typically, a Finesse dialog is modal, i.e. usually one and only one of the windows shown possesses the input focus, and only there user input is possible. This quality reflects the sequential course of scripts in the window dialog and makes possible programming the user dialog in the script in a simple and structured way. Windows are signaled inactive by a one-way cursor. Interaction with an inactive window is limited; in particular, you cannot return to the shell script from an inactive window. There are exceptions from modality: these are, for one part, the echo window and save window for saving variables. All the time both of these windows are accessible, i.e. input sensitive, independent of other windows. They can be closed anytime, for example. For the other part, external list windows and file selection windows are not modal. These windows can be called simultaneously and more than once from the main input window. Input in those windows can be made at the same time as in the main input window. When the main window is inactive or closed, the corresponding list and file selection windows are inactive or closed as well.

Finesse resources

Figure 25 shows the beginning of a rather long shell script named aba up to calling the first window aba1, Figure 25. With an additional Fsclose command at the end, this example can be completed to an entire Finesse script.

```
#! /bin/sh
#
# Shell script aba
#
. ${FINESSEPATH-/usr/local/finesse}/fsshinit

Fsopen "$@"

aba1="
  FsWindow         -name aba1
                   -title Aba1
                   -btype osa;
  FsSeparator;
  FsOptionMenu     -label 'Select program type:'
                   -items 'analyse restart postout
postfile savedata'
                   -var progtype=analyse;
  FsSeparator;
  FsSelectionText -var inputfile
                   -label 'Input File:';
  FsSeparator;
  FsText           -var projectno
                   -label 'Project Number:';
  FsText           -var timelimit
                   -label 'Time limit (Min):';
  FsText           -var memory
                   -label 'Memory (Mb):';
  FsSeparator;"

Fsdisplay -w "$aba1"
.
.
.
```

Figure 25 aba script segment

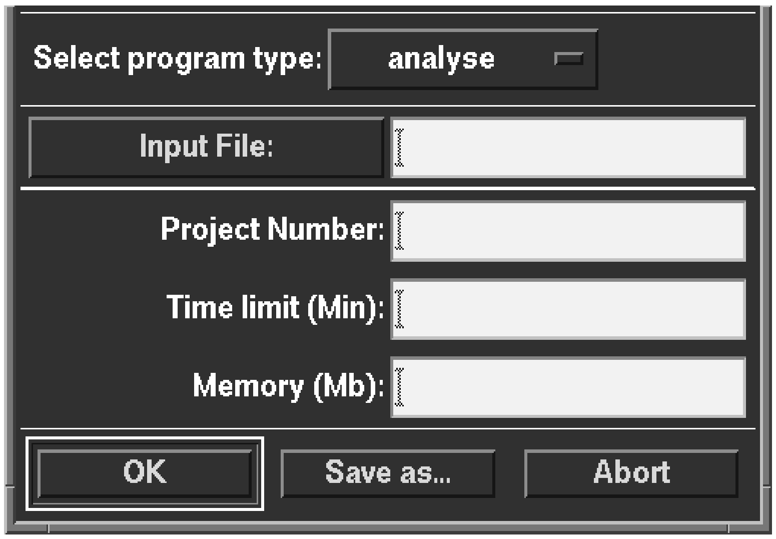


Figure 26 Window output from `aba` script

Finesse resources

The `file_selection` script has shown that current window parameter settings registered in the application server () can be saved as Finesse resources by the `Fssave` command to a Finesse resource file. A Finesse resource file will be read every time the `Fsdisplay` command is called. Thus the window parameter settings saved from a previous application call can be set automatically on restarting the application.

The Finesse resources are saved to a file in the `$HOME/.finesse` directory by default. The file name is composed of the calling shell script name and the extension `.rsc`.

The Finesse resources are defined using the format:

variable: value

On one side, the `Fssave` call saves every window variable with its current value registered in the Finesse server at this moment. A window variable is registered if its associated window has been opened at least once by a `Fsdisplay` call.

On the other side, window declarations can contain several more variable definitions apart from those defined with the `–var` option. These variables will be saved as well if their values can be modified by the user. For example, if the argument to the `–dirmask` option of `FsSelectionText` has the form *varname=value*, the varname variable is saved together with its current value. On the other hand, the variable that can be assigned to a label in a `FsLabel` declaration will not be saved because this label can be modified only by the shell script, but not by the user.

Finally the new resource file will take over without modification all those variable/value pairs that have not been registered in the Finesse server so far by a window call. This grants that, for example, Finesse resources from sub windows not opened in the current application will be saved.

Saving Finesse resources to any file

Saving window parameter settings by the `Fssave` command in the shell script is automatic. However, it is possible for users to save resources in runtime to a file with any name in any directory. When calling the window that shall handle the back up, one more field labeled `Save as ...` must be created. Our example does this with the `FsWindow` argument `-btype osa` where the letter `s` is the reason for the extra field between `OK` and `Abort`. Clicking at `Save as ...` will open a file selection window where you can choose or enter the desired resource file as shown in Figure 25. Pressing `OK` in the file selection window writes the variable/value pairs registered in the application server to the selected file.

Reading any Finesse resource file

Usually Finesse resources saved in any file can be read with the command line when the user starts the program. For example, the command

```
scriptname -r resourcefile
```

specifies a resource file named `resourcefile` for the script named `scriptname`. The `-r` option gives the opportunity to comfortably choose one out of different window settings irrespective of the `.rsc` file.

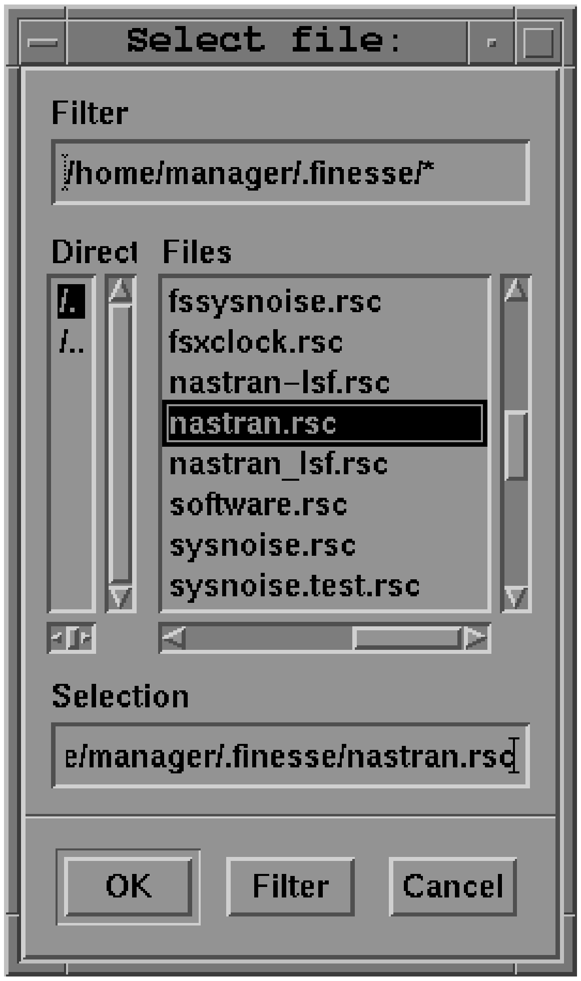


Figure 27 File selection box screen

If the `-r` option is used when calling the script, the selected file is searched for and read instead of the `.rsc` file. If the selected resource file can't be found, the `.rsc` file connected with the shell script name will be looked up and read.

`Fsopen` can read the specified file only if it is passed the option `-r resourcefile`. That is the reason `Fsopen` carries the `"$@"` argument containing the command parameters. If the `$@` variable does not contain the command parameters any more (for example, because of a shift command issued in the meantime), the command parameters have to be saved to be passed to the Fsopen command correctly. Calling `Fsopen` before changing any command parameters is essentially advisable. In an sh script the `Fsopen` command generally should contain the `"$@"` option.

In an csh script the list of arguments is available in the predefined shell variable `argv`. `Argv` usually will not be modified in the course of a shell script. Therefore in a csh-script `Fsopen` directly evaluates the command parameters with `argv` so that `Fsopen` does not need any transfer arguments. If for some reason `argv` should be modified before calling `Fsopen`, similarly to the sh script method, the command parameters have to be saved and then passed to `Fsopen` explicitly.

Container widgets

The examples in previous chapters are characterized by a simple widget arrangement: the widgets declared in one window definition are grouped one below the other. This vertical widget grouping is sufficient as a default for many of the smaller windows. There is a problem, however, with windows containing many widgets. Furthermore, it is often desirable for logical or ergonomic reasons to arrange certain widgets beside each other or to pool a group of widgets in an appropriate way.

Nastran script

```
#! /bin/sh
#
# Create simple front end for nastran jobs
# ---------------------------------------
#
# Use some NASTRAN Info
#---------------------

NAST_VER="68"
NAST_RC="nast${NAST_VER}rc"
EXE_NAME="NAST${NAST_VER}r2"

# Finesse Initialization
# ----------------------

. ${FINESSEPATH-/usr/local/finesse}/fsshinit# Declare
input window
# --------------------

yes=Yes
no=No

jidname="Input File"
rcfname="RC File"
sdiname="Scratch Directory"

naswin="
  FsWindow        -name naswin
                  -title Finesse;
  FsSelectionText -label '$jidname:'
                  -var jid;
  FsSelectionText -label '$rcfname:'
                  -var rcf=$NAST_RC;

  FsSeparator -name separator;

  FsForm -name form1 -orientation horizontal;
  FsForm -name form2 -parent form1;
  FsList -label '$sdiname:'
```

```
        -items '/net/ws1/scr
                /net/filesrv1/scr/scr1
                /net/filesrv1/scr/scr2
                /net/filesrv2/scr/scr1
                /net/filesrv2/scr/scr1'
        -var sdi=/net/filesrv1/scr/scr1
        -parent form2 -nvisible 4;

  FsSeparator -parent form2 -line no
              -name separator1;

  FsText   -label 'Total Memory (MB):'
           -var mem=50 -parent form2
           -name text1 -packing tight;

  FsText   -label 'Scratch Memory (MB):'
           -var smem=50 -parent form2
           -name text2 -packing tight;

  FsSeparator -name midsep -line dashed
              -parent form1;

  FsForm -name form3 -parent form1;

  FsRadio  -label 'Batch Job:'
           -items '$yes $no' -name radio
           -var bat=$yes -parent form3;

  FsRadio  -label 'Delete Scratch:'
           -items '$yes $no' -name radio
           -var scr=$yes -parent form3;

  FsRadio  -label 'Notify when job is done:'
           -items '$yes $no' -name radio
           -var not=$no -parent form3;

  FsRadio  -label 'Save old files:'
           -items '$yes $no' -name radio
           -var old=$yes -parent form3;

  FsSeparator -name separator;

  FsPushButton -label 'Start Job';
  FsPushButton -label 'Job status';
  FsPushButton -label Exit;
"

# Plausibility checks:
# --------------------

# Test filenames. $1: variable; $2: name
# ---------------
```

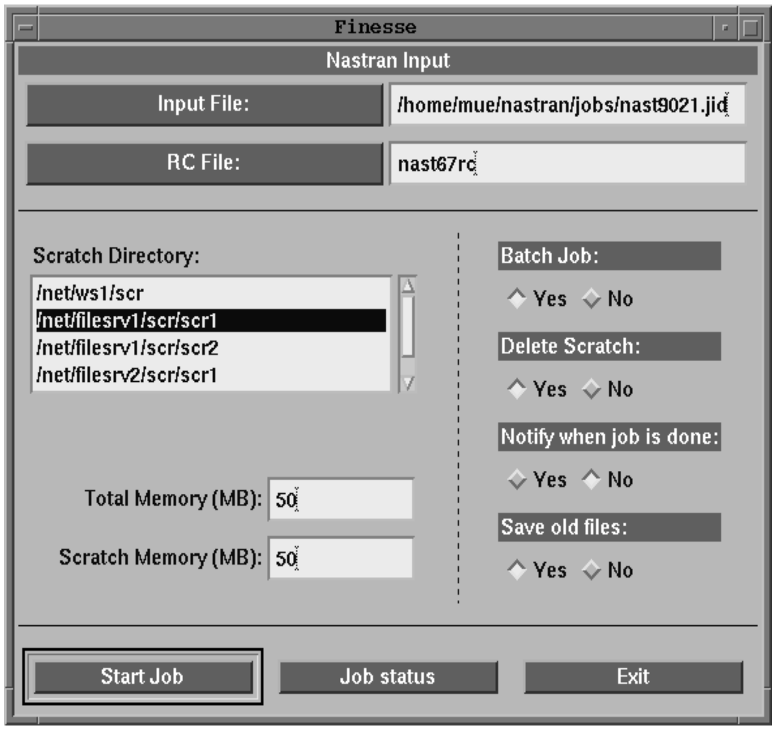
```
testname()
{
  if [ ! -z "$1" ] ; then
    return 0;     # correct
  else
    mesg="$2 missing..."
    return 1;     # erroneous
  fi
}
# Test Memory
# $1: value; $2: lower limit; $3: upper limit

testmem()
{
if [ -z "$1" ] ; then
  mesg="No Memory given..."; return 1;
else
  var=`expr $1 : '.*\([^0-9]\).*' `
  if [ -z "$var" ] ; then
    if [ $1 -lt $2 ] ; then
     mesg="$1<$2: Memory too small...";
     return 1; fi
    if [ $1 -gt $3 ] ; then
     mesg="$1>$3: Memory too large...";
     return 1; fi
    return 0;
  else
    mesg="Bad memory value..."; return 1;
  fi
fi
}

# Begin Finesse dialog, open input window
# -----------------------------------------

Fsopen "$@"

Fsdisplay -w "$naswin" -m "Nastran Input"if [ "$fsbutton" !=
"o" ] ; then
 Fsclose; exit 0; fi

# Check variables, on error reopen window
#-----------------------------------------

memlower=10
smemlower=10
memupper=100
smemupper=100

until testname "$jid" "$jidname" &&
      testname "$rcf" "$rcfname" &&
      testname "$sdi" "$sdiname" &&
      testmem $mem $memlower $memupper &&
      testmem $smem $smemlower $smemupper
do
  Fsdisplay -n naswin -m "$mesg"
  if [ "$fsbutton" != "o" ] ; then
   Fsclose; exit 0; fi
done

Fssave

# OK, start job
# -------------

# ...
# $EXE_NAME jid=$jid rcf=$rcf sdi=$sdi\   mem=$mem
smem=$smem bat=$bat scr=$scr\
   not=$not prt=$prt old=$old
# ...

Fsclose
```

Figure 28 `nastran` script

The `nastran` script generates a simple input window for a Finite Elements Package as shown in Figure 28. Two horizontal separators split the window in three parts. The middle part contains widgets that are arranged beside each other. At the left side you see a group of widgets consisting of a list as well as two text widgets. On the right side there is a group of Yes/No check buttons.

Inside any container element the default is again a vertical arrangement, so the radio buttons, as well as list and text widgets, appear one below the other on their respective sides. If you want to create a horizontal arrangement you have to add the option `-orientation horizontal` to `FsForm`. In our example we have set this option for the container element `form1` to arrange the container element `form2` at the side of `form3`. Both of these two elements are contained in `form1`, which shows that container elements can be nested, i.e. contained in other container elements.

Widget grouping is made possible by the `FsForm` widget. This container element enables you to design your windows in a very flexible way. It does not appear on the screen, but serves for taking in and grouping other widgets. The argument `-parent` *name* assigns a widget to a given container element, where *`name`* is the container element name given by the `FsForm -name` option. For example, the four radio buttons are assigned to the `FsForm` element `form3` by the `-parent form3` option; the list and the text widgets below are assigned to the container element `form2`.



Input File: /home/mue/nastran/jobs/nast9021.jid

RC File: nast67rc

Scratch Directory:
/net/ws1/scr
/net/filesrv1/scr/scr1
/net/filesrv1/scr/scr2
/net/filesrv2/scr/scr1

Total Memory (MB): 50

Scratch Memory (MB): 50

Batch Job: [✓ Yes] No

Delete Scratch: [✓ Yes] No

Notify when job is done: Yes [✓ No]

Save old files: [✓ Yes] No

Figure 29 `nastran` input window

The element `FsWindow`, which is present in every window definition, is also one of these container elements. It has to take in every widget not explicitly assigned to another container element and therefore is the default parent window. Another default is that its widgets are aligned vertically. As for `FsForm`, also for `FsWindow` you can change the alignment by the argument `-orientation horizontal`.

The `-packing` option defines the size of widgets inside the window. The default value for container widgets is `tight`. Thus every widget is assigned the smallest possible size. By contrast, every other widget with the `-packing` option has a default value of `equal`. This means that label and actual text widget have the same size, e.g. for `FsText` widgets. Internally, these widgets again consist of a container with additional elements contained inside.

You can also arrange widgets in an array with the help of `FsForm` and `FsWindow` elements. The number of rows (when oriented horizontally) or of columns (when oriented vertically) can be specified by the `-nrows` option. This only works if you have set the `-packing` option to `equal` because this is necessary to arrange array elements homogeneously. By giving the `-spacing` option, you can additionally define the horizontal and vertical distance between widgets.

Setting X resources

Sometimes arranging widgets by means of `FsForm` in various ways is not sufficient to give your windows the desired appearance. Without any specific control, widgets will get a size by default, for example, determined by contents or labels. This size cannot be changed in the window declaration. On the other hand, the resource mechanism of the X window system gives you the opportunity to set freely many properties of X applications, such as widget size or color. To this end the desired properties or resources of X applications are defined according to certain rules in order to let them take effect on starting the application. Although Finesse applications are shell scripts in the first place, Finesse fairly supports all of these possibilities in the usual way of X applications. First the programmer sets properties or resources when developing the application, but any user may change many or even all of the resources, depending on the application, to meet individual needs.

X resources can be defined at many places: by using the command line, inside the application, in certain files in user directories or in the X11 directory tree, in files specified by environment variables, or inside shell scripts. Finesse uses or supports all these possibilities to set X resources from the outside. As a complete discussion is impossible here, we restrict ourselves to giving several useful suggestions for the application programmer of how to set resource defaults and some hints concerning special features of Finesse scripts. A fully detailed discussion of all sources for setting resources and their order of priority can be found in the *X Window System User's Guide* or *OSF/Motif User's Guide*.

Syntax of resource settings

Any application consists of single widgets structured hierarchically and each having a number of resources to set. Every resource, every widget and also every application has an instance name as well as a class name. The instance name serves for characterizing individually each of these widgets. The class name specifies the general category to which the widget belongs. To be able to assign a value to a specific resource in a specified widget of an application all necessary names have to be given when specifying the resources.

A resource specification generally consists of several components in the following format:

```
application*widget_hierarchy*resource: value
```

The component names can be either instance names, or class names, or any combination of both. The widget hierarchy can consist of one or several widget names, and is optional, same as the application name. If the window hierarchy is not given, the resource concerned will be set for all appropriate widgets. If the application name is missing, the resource concerned will be set for all appropriate applications. An asterisk * separating two of the components or widgets means that this resource is valid also for all appropriate widgets or hierarchy levels in between. So not the entire order of hierarchy must be given.

A little example helps to visualize the range affected by resource settings:

```
*background:           DarkGreen
nastran*background:    MidnightBlue
*mytextfield*font: -adobe-helvetica-bold-r-normal--20-*iso8859-1
```

The first two lines specify the resource `background`. The widget hierarchy in the first line is missing, so the resource is used for all widgets and all applications. This means that everywhere the background color is set to `DarkGreen`. The file /usr/lib/X11/rgb.txt contains a list of predefined colors and their RGB values. In the second line the resource `background` is specified in detail for the application `nastran`. Because of the asterisk * between `nastran` and `background`, this resource is set for all widgets of the application which makes `MidnightBlue` the global background color. The last entry sets a Helvetica font with 24 pt size, which is much larger than the default, for widgets named `mytextfield` of all applications. The `xlsfonts` command produces a list of possible fonts and their exact names.

Instance names and class names of widgets

All Finesse windows and also various Finesse widgets themselves are structured hierarchically. For example, `FsForm` widgets contain other widgets, and a `FsText` widget is composed internally of three widgets belonging to two different levels. By specifying the window hierarchy accordingly, a resource can be restricted to some widgets or applied to a whole group of widgets. The following table shows the window hierarchy of all Finesse widgets and the corresponding class and instance names.

The table is subdivided in blocks. Each block describes the widget hierarchy of the widget specified in the corresponding heading. For example, if you create the widget `FsText`, internally one widget of the XmRowColumn class, one widget of the XmLabel class, and one widget of the XmText class are generated. The number of asterisks in front of the class name describes the depth of the hierarchy below the parent widget. Inside of one block, the widgets of one hierarchy level are children of the parent widget immediately above. For example, in an echo window two widgets of the XmScrollBar class and one widget of the XmText class are children of a widget of the XmScrolledWindow class. A number in square brackets means that the element maybe referenced by this number for resource specifications with the -r option of Fsdisplay.

Table 4 Class and instance names of Finesse widgets

Hierarchy	Class name	Instance name
	FsWindow **widget**	
*	TopLevelShell	*name* else FsWindow
**	XmForm	*name* else FsWindow
***	XmRowColumn	*name* else FsWindow
	FsForm **widget**	
*[0]	XmForm	*name*
**[1]	XmRowColumn	*name*
	Echo Window	
*	TopLevelShell	Echo
**	XmForm	Echo
***	XmScrolledWindow	EchoSW
****	XmScrollBar	HorScrollBar
****	XmScrollBar	VertScrollBar
****	XmText	Echo
***	XmPushButton	Echo
	Info label	
* [0]	XmLabel	Info
	FsLabel widget	
* [0]	XmLabel	*name* else FsLabel
	FsText **widget**	

Table 4 Class and instance names of Finesse widgets

Hierarchy	Class name	Instance name
* [1]	`XmRowColumn`	*name* else `FsText`
**[2]	`XmLabel`	*name* else `FsText`
**[0]	`XmText`	*name* else `FsText`
	`FsSelectionText` **widget**	
*[1]	`XmRowColumn`	*name* else `FsSelectionText`
**[2]	`XmPushButton`	*name* else `FsSelectionText`
**[0]	`XmText`	*name* else `FsSelectionText`
	`FsRadio` **widget**	
* [1]	`XmRowColumn`	*name* else `FsRadio`
**[2]	`XmLabel`	*name* else `FsRadio`
** [0]	`XmRowColumn`	*name* else `FsRadio`
*** [3...]	`XmToggleButton`	*label*
	`FsCheck` **widget**	
* [1]	`XmRowColumn`	*name* else `FsCheck`
**[2]	`XmLabel`	*name* else `FsCheck`
** [0]	`XmRowColumn`	*name* else `FsCheck`
*** [3...]	`XmToggleButton`	*label*
	`FsOptionMenu` **widget**	
*[1]	`XmRowColumn`	*name* else `FsOptionMenu`
**[0]	`XmRowColumn`	*name* else `FsOptionMenu`
***[2]	`XmLabelGadget`	`OptionLabel`
***[3]	`XmCascadeButtonGadget`	`OptionButto`
**	`XmMenuShell`	`popup_`*`name`* else `popup_FsOptionMenu`
*** [4]	`XmRowColumn`	*name* else `FsOptionMenu`
****[5...]	`XmPushButton`	*label*
	Sequence of push buttons	
*	`XmRowColumn`	*name* `von FsWindow` else `FsWindow`
** [0]	`XmPushButton`	*`name`* else `FsPushButton`
	Sequence of push buttons from `–type` **option**	

Table 4 Class and instance names of Finesse widgets

Hierarchy	Class name	Instance name
*	`XmRowColumn`	*name* `von FsWindow` else `FsWindow`
**	`XmPushButton`	*name* `von FsWindow` else `FsWindow`
	`FsSeparator` **widget**	
* [0]	`XmSeparator`	*name* else `FsSeparator`
	`FsList` **widget when list included**	
*[4]	`XmRowColumn`	*name* else `FsList`
**[5]	`XmLabel`	*name* else `FsList`
**[6]	`XmScrolledWindow`	*name*`SW` else `FsListSW`
*** [8]	`XmScrollBar`	`VertScrollBar`
***[10]	`XmList`	*name* else `FsList`
	`FsList` **widget in main window when list not included**	
*[1]	`XmRowColumn`	*name* else `FsList`
** [2]	`XmPushButton`	*name* else `FsList`
**[3]	`XmText`	*name* else `FsList`
	`FsList` **window when list not included**	
*	`TopLevelShell`	*name* else `FsList`
**	`XmForm`	*name* else `FsList`
***[4]	`XmRowColumn`	*name* else `FsList`
****[5]	`XmLabel`	*name* else `FsList`
**** [6]	`XmScrolledWindow`	*name* else `FsListSW`
*****[8]	`XmScrollBar`	`VertScrollBar`
*****[0]	`XmList`	*name* else `FsList`
****	`XmRowColumn`	*name* else `FsList`
*****	`XmPushButton`	*name* else `FsList`
*****	`XmPushButton`	*name* else `FsList`
****	`XmSeparator`	*name* else `FsList`
****	`XmRowColumn`	*name* else `FsList`
*****	`XmPushButton`	*name* else `FsList`
*****	`XmPushButton`	*name* else `FsList`
*****	`XmPushButton`	*name* else `FsList`

Table 4 Class and instance names of Finesse widgets

Hierarchy	Class name	Instance name
	File selection box for saving Finesse variables	
*	`TopLevelShell`	`Save`
**	`XmFileSelectionBox`	`Save`
***	`XmLabelGadget`	`Items`
***	`XmScrolledWindow`	`ItemsListSW`
****	`XmScrollBar`	`VertScrollBar`
****	`XmScrollBar`	`HorScrollBar`
****	`XmList`	`ItemsList`
***	`XmLabelGadget`	`Selection`
***	`XmTextField`	`Text`
***	`XmSeparatorGadget`	`Separator`
***	`XmPushButtonGadget`	`OK`
***	`XmPushButtonGadget`	`Apply`
***	`XmPushButtonGadget`	`Cancel`
***	`XmPushButtonGadget`	`Help`
***	`XmLabelGadget`	`FilterLabel`
***	`XmLabelGadget`	`Dir`
***	`XmTextField`	`FilterText`
***	`XmScrolledWindow`	`DirListSW`
****	`XmScrollBar`	`VertScrollBar`
****	`XmScrollBar`	`HorScrollBar`
****	`XmList`	`DirList`
	File selection box of `FsSelectionText` widget	
*	`TopLevelShell`	*name* else `FsSelectionText`
**	`XmFileSelectionBox`	*name* else `FsSelectionText`
***	`XmLabelGadget`	`Items`
***	`XmScrolledWindow`	`ItemsListSW`
****	`XmScrollBar`	`VertScrollBar`
****	`XmScrollBar`	`HorScrollBar`

Table 4 Class and instance names of Finesse widgets

Hierarchy	Class name	Instance name
****	XmList	ItemsList
***	XmLabelGadget	Selection
***	XmTextField	Text
***	XmSeparatorGadget	Separator
***	XmPushButtonGadget	OK
***	XmPushButtonGadget	Apply
***	XmPushButtonGadget	Cancel
***	XmPushButtonGadget	Help
***	XmLabelGadget	FilterLabel
***	XmLabelGadget	Dir
***	XmTextField	FilterText
***	XmScrolledWindow	DirListSW
****	XmScrollBar	VertScrollBar
****	XmScrollBar	HorScrollBar
****	XmList	DirList

Setting resources in Finesse windows

Using some resources of the `nastran` example, Figure 28 shows how you can set resources in Finesse windows with the help of the table. The file `examples/app-defaults/Nastran` with the resource list is shown below.

```
*background:                         Gray75
*foreground:                         Black
*Info*background:                    Cyan4
*Info*foreground:                    White
*XmList*background:                  AntiqueWhite
*XmPushButton*background:            SteelBlue
*XmPushButton*foreground:            White
*XmText*background:                  AntiqueWhite
*XmToggleButton*selectColor:         Yellow
*radio*XmLabel*background:           SteelBlue
*radio*XmLabel*foreground:           White
*midsep*orientation:                 XmVERTICAL
*midsep*width:                       50
*separator1*height:                  50
```

```
*separator*height:                   25
*FsOptionMenu*XmLabelGadget*width:   200
*FsEcho*XmPushButton*width:          100
*XmPushButton*width:                 173
*text1*XmLabel*width:                180
*text1*XmText*width:                 120
*text2*XmLabel*width:                180
*text2*XmText*width:                 120
*XmText*width:                       280
```

The file starts with two entries specifying foreground and background color. As the entries don't contain a widget hierarchy these resources are valid for all the widgets of this application. For some of the widgets these settings are modified by later entries. For example, the widgets of the `Info` class will not follow the generic rules since they have been specified with more specific definitions. From the table of the previous paragraph and the `-name` arguments of the window declaration we know that apart from the `Info` label there are no widgets with class name or instance name `Info`. Therefore these two resources set the colors exclusively for the `Info` widget.

The situation is different for the `XmPushButton` class colors set later on. Here, too, only the class name is given as a hierarchy, surrounded by two asterisks. So this resource is valid for all widgets of the `XmPushButton` class on all levels. This setting affects the push buttons in the `FsSelectionText` widget as well as the push buttons at the bottom of the window.

Finally by setting the resources

```
*radio*XmLabel*background:     SteelBlue
*radio*XmLabel*foreground:     White
```

also, the label colors for the radio switches are set to `SteelBlue` and `White`. The hierarchy in this case is given by the instance name `radio` defined by the `-name` option of `FsRadio` and the class name `XmLabel`. From the , it is seen that the widget specified by `radio` belongs to the `XmRowColumn` class. If instead of `radio` the class name had been chosen, the `FsText` widget labels and the list labels would get the same colors, because these widgets, too, have the same two-level hierarchy order.

The file contains some more entries, which refer to the orientation of the middle separating line and to width and height specifications. The selection of appropriate widgets is made the same way as for the color settings. The manual pages for a given X class name (e.g., `man XmText)` show what resources can generally be set for this class. You can find some more examples for setting resources in files in the `finesse/examples/app-defaults` directory. A comfortable way of setting resources for an application is the `editres` program that is part of the X distribution on some computers. It would lead too far to discuss further details of setting resources, such as tight and loose bindings. You can look up these details in the books mentioned above, *X Window System User's Guide* and *OSF/Motif User's Guide*.

Standard resource files

Every X application - and every Finesse application as well - should have its own resource file where the developer defines default values for the application. This file usually is located in the /usr/lib/X11/app-defaults directory and is called the app-defaults file. When an application is started, the app-defaults directory is searched for a corresponding resource file, and if it is found, this file is read in. The resource file name typically is the class name of the application. Finesse applications follow the X convention to form the class name by capitalizing the first letter of the name of an application or, the first one being an `x`, the first two letters, respectively. For example, if any user writes a Finesse script named `nastran` the corresponding resources should be stored in a file called `Nastran`, and this file should be moved to the directory `/usr/lib/X11/app-defaults` when installing the application. On starting the application this file will be read automatically.

While developing a script, you can easily test X resource settings by means of the environment variable `XENVIRONMENT` .

```
setenv XENVIRONMENT tmpres
```

sets as a X resource file the file called `tmpres` in the current working directory, i.e. every X application started later on - with any name - will read this file at the beginning. When the settings in this file have been adapted to the script being developed the file can be copied with its correct name to the directory `/usr/lib/X11/app-defaults` and variable `XENVIRONMENT` can be deleted from the environment.

The Finesse resource file

The file `Finesse` is part of the distribution. It contains a few resource settings that give the default values for any Finesse application, but can be changed if necessary. This is why these resources have not been integrated into the Finesse code. One of these resources is the editing functions for text widgets, which are similar to the Gnu-Emacs editor. Users preferring other editors will thus be able to change these editing functions. Here a minor bootstrap is that the resources from the file `Finesse` are set only if in the `app-defaults` directory no file specific for this application is found. On the other hand, this contradicts the suggestion to offer one such file for every application. To solve this dilemma and keep the Finesse resources either you can copy the contents of Finesse into the app-defaults file, or you include them starting from X11R5 with

```
#include "/usr/local/finesse/Finesse"
```

for example.

Changing resources within the script

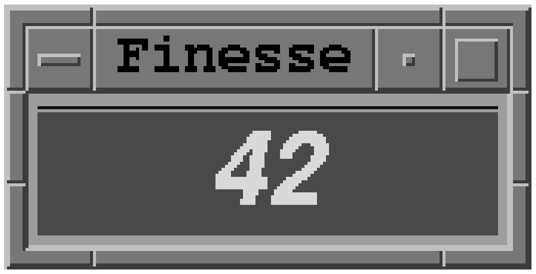


Figure 30 display from example `csh/display_primes`

The Finesse developer has the opportunity to convert X resources in Finesse elements during runtime by using the `-r Fsdisplay` option. Thus it is possible, e.g., to change the background color for text fields with wrong input in order to give users an optical help for correcting their input on recalling the window. Calling

`Fsdisplay -r` *name*`:`*resource*`:`*value*

converts the X resource *resource* in widget *name* to the value *value*.

The following example re-implements the display_primes example in a simpler way, using the `-r` option. For a change this example is written as a csh script:

```
#! /bin/csh
#
#
if (! $?FINESSEPATH) set FINESSEPATH =
/usr/local/finesse
source $FINESSEPATH/fscshinit
```

```
set path = ($path /usr/games)

# Definition

set i = 1
set prf = "tf:background:black"
set prf4 = "tf:foreground:red"
set prf2 = "tf:labelString:Starting..."

set windef = \
"FsWindow -name pw;FsSeparator;FsLabel -label $i
-name tf;"

Fsopen
while (1)
  Fsdisplay -w $windef:q -n pw -s touch \
            -r $prf -r $prf2 -r $prf4
  if ( "$fsbutton" == "a" ) then
    break
  endif

  sleep 1
  set i = `expr $i + 1`
  set wl = (`factor $i`)
  shift wl
  if ( "$wl[1]" == "$i" ) then
    set prf = "tf:background:black"
    set prf4 = "tf:foreground:red"
  else
    set prf = "tf:background:blue"
    set prf4 = "tf:foreground:yellow"
  endif
  set prf2 = "tf:labelString:$i"
end
Fsclose
```

Figure 31 `csh/display_primes` script

For simple Finesse elements such as `FsLabel` *name* defines the name given in the window declaration by the `-name` option. For compound elements such as `FsText`, consisting of label field and text field, a number enclosed in square brackets can be added if the desired X resource is not to be set for the main element. The `FsText` main element is the text field, the label field has number 2. Calling

```
Fsdisplay -r mytext:background:red \
           -r mytext[2]:background:blue
```

will set the background colors for text element `mytext` to different values for label field and text field. Earlier in this chapter you will find a table where column 1 tells you which element is the main element ([0]) and what are the numbers (in square brackets) of other elements. Fields in menus (`FsRadio`, `FsCheck`, and `FsOptionMenu`) are also referenced by the name of their Finesse element and receive successive numbers. An element with no name set in the window declaration cannot convert settings of X resources. Elements designed to convert resource settings must not contain colons in their names.

Setting X resources using the command line

X applications should allow the user to set resources when invoking the application, too. Primarily, this method serves for specifying standard application resources. One of these is the X server used. For example, with the command

```
aba -display orion:0
```

the `orion` workstation X server can be used to display the shell script `aba`. Correspondingly, the call

```
aba -fg red -bg yellow
```

sets the foreground color of application `aba` to `red` and the background color to `yellow`.

With the help of the `-xrm` command option any X resource can be specified if the application allows this. The `-xrm` option argument is a resource setting in the usual variable/value syntax. For example, the command

```
aba -xrm ’*background: yellow’
```

sets the window background to yellow. The resource statement is surrounded by single quotes ’ (acute) so that it will be passed as one argument and the * sign in a csh environment is not interpreted prematurely by the calling shell. By repeating the `-xrm` option, several resources may be set when invoking an application.

Finesse supports both of the possibilities to set X resources when calling a script. The shell script arguments only have to be passed to the Finesse server to be taken into account when windows are created. In csh scripts this is done automatically in the initialization files, whereas in sh scripts the server must be called with the argument `“$@”`:

```
Fsopen “$@”
```

Here it is important not to forget the double quotes. They cause every shell script argument to be preserved and passed to the server without any alteration. Without the double quotes the second argument `'*background: yellow'` would be passed as two separate arguments `*background:` and `yellow`, which would make the `-xrm` resource argument incomplete. This demonstrates that the argument `"$@"` should not be left out from any Finesse server call in an sh script.

The previous explanations show that with the help of the -xrm option, X resources can be set inside a Finesse script as well. They only have to be set as additional arguments when the Finesse server is invoked. You could as well set the background color to yellow for an application by calling

```
Fsopen -xrm '*background: yellow'
```

As resource settings by the `-xrm` option have higher priority compared to equivalent specifications in a resource file, you should try to avoid setting resources inside the shell script. In this case the user has no opportunity to choose his or her own global background color. This kind of resource setting is useful only if the aim definitely is to prohibit any change in the resource value. It is assumed to be good style to allow resource settings by the user as often as possible.

Resource settings by users

Users, too, have several different possibilities to set resources for their applications. This can be done with the help of the environment variables `XUSERFILESEARCHPATH` or `XAPPLRESDIR` or in files specific for the application with the application class name. Often users specify all their resource parameters in the `.Xdefaults` file in the home directory. As entries for all applications of a user are located there, resource entries restricted to a certain application have to start with the instance or class name of this application. By default the instance name is identical to the application name, but can be changed to any other value with the command option `-name`. After the instance or class name, the widget hierarchy, the resource name, and its value are given in the usual way. For example, the `.Xdefaults` entries

```
*background:          SlateGray
nastran*background:   RoyalBlue
abaqus*background:    RosyBrown
```

specify `SlateGray` as the standard background color for all applications, but `RoyalBlue` and `RosyBrown` for the applications `nastran` and `abaqus`, respectively.

Actually, the resources specified in `.Xdefaults` are not generally read when an X program is called. Many X configurations automatically set the `RESOURCE_MANAGER` property of the root window of the server used by calling the `xrdb` program from a startup script when logging in. `Xrdb` loads resources from files specified as arguments (for example, `.Xdefaults` or `.Xresources`) to the X server. When an X application is called, only the list of resources loaded to the server is searched but not any more the `.Xdefaults` file. In those cases where the `RESOURCE_MANAGER` property is set when logging in the user best will save his or her resource entries to one of the files loaded by `xrdb`.

Only if the `RESOURCE_MANAGER` property has not been set `.Xdefaults` will be read, searched for entries concerning the current application, and the appropriate values will be set. By means of the command

```
xrdb -merge .Xdefaults
```

new resources added to `.Xdefaults` can be loaded to those present in the server and are then instantaneously available for subsequent X program calls.

Initialization files

At the beginning of a Finesse shell script, a short initialization script is called which is named `fsshinit` (sh) or `fscshinit` (csh) depending on the shell type. Using the initialization script allows writing Finesse scripts as described in the preceding sections. On the other hand using the initialization scripts in their existing form is not at all mandatory. For partial modification— if desired—some deeper insight into your initialization scripts is necessary. Therefore the sections that follow will deal with these scripts in more detail. If you are interested in programming Finesse applications only in the form shown in our examples, then skip the remainder of this section.

The fsshinit initialization script

The initialization script `fsshinit` for sh scripts has the form

```
# Finesse definitions for sh-type scripts

SHELLTYPE=sh export SHELLTYPE
PATH=${FINESSEPATH-/usr/local/finesse}/bin:$PATH
export PATH
null=$0

Fsclose()    { eval "`fsclose "$@"`" ; return $? ; }
Fsdisplay()  { eval "`fsdisplay "$@"`" ||
                { Fsclose; exit 1 ; } }
Fsecho()     { eval "`fsecho "$@"`" ; return $? ; }
Fsopen()     { eval "`fsopen -n "$null" -p $$ \
                    ${1+"$@"}`"|| exit 1 ; }
Fssave()     { eval "`fssave "$@"`" ; return $? ; }
```

The first line sets the shell variable `SHELLTYPE` to `sh`. As the shell variable `SHELLTYPE` will be read by every subsequent Finesse command, it has to be set first.

Finesse function definitions

After the shell type is set, the definition of the shell functions `Fsopen`, `Fsclose`, `Fsdisplay`, `Fsecho`, and `Fssave` are given. As a shell function in the shell script is called without the parenthesis of its definition, this means that the Finesse "commands" `Fsopen`, `Fsdisplay`, etc. actually are function calls in the shell script. Calling a function will execute the list of commands defined by the function.

The command list structure looks similar for all functions. First, `eval` is executed, which has a command substitution as an argument. Finally, either `return $?` returns the exit status of the `eval` command to the calling shell script as the exit status of the function, or, in case of errors within `Fsdisplay` and `Fsopen` the script is terminated by the `exit` command.

Basic Finesse commands

In order to understand the meaning of the function definitions let's have a look at the `eval` command. First the `eval` command reads its arguments and combines them with a command. Among others, command substitutions are executed on reading. Command substitution means that a command enclosed in single back quotes will be executed and substituted by its standard output. The generated command will be read and executed subsequently if it is non vanishing. So taken all together the `eval` command arguments will be read twice.

The fsdisplay function

When working with Finesse commands you will notice corresponding Finesse functions. These functions are called by the Finesse command of the same name, but with an initial capital letter. For example, the `eval` command of `Fsdisplay` calls `fsdisplay` to do the actual work.

So `fsdisplay` is the actual command that opens a window when `Fsdisplay` is called. The `Fsdisplay` arguments, for example, the window description, are passed to `fsdisplay` by means of the `"$@"` option. The `fsdisplay` command therefore has the same argument options as `Fsdisplay`.

On successful completion `fsdisplay` writes a line to the standard output which in the case of the text input example of Figure 8 could look like this:

```
fsbutton='o' && name='Herbie'
```

After that the above line will be read by `eval` and executed as a list of commands, a method by which the shell variable `fsbutton` gets the value `o` and the variable `name` gets the value `Herbie` in our example. By this mechanism `Fsdisplay` sets the variables defined by the window declaration to the values specified in the window. The variable values are returned enclosed in single quotes ` ` which simplifies returning special characters, among others. Even with no variable specified in the window declaration, at least the `fsbutton` variable will be set when `fsdisplay` terminates correctly.

The outer double quotes of the `eval` command finally prohibit that declaration carriage returns, if existent, are lost during the first `eval` phase, which is important with complex list output. As other Finesse commands do not return any variables, they would not need these additional quotes.

The fsopen command

In a way similar to `fsdisplay` the basic commands `fsopen`, `fsclose`, `fsecho`, and `fssave` of the remaining function definitions are evaluated. The `fsopen` command deserves a special comment as it can be called with various command options. Like the `fsdisplay` command it possesses the `"$@"` argument option. So `Fsopen` and `fsopen` understand the same arguments.

The command `fsopen` contains two options that have not yet been mentioned in the `Fsopen` description in previous sections.

`-n` *name*

Passes the Finesse shell script name as its value. This name is necessary to identify the Finesse and X resources assigned to this particular application.

`-p PID`

specifies id of the exit process to the application server.

Termination of this process will automatically cause the termination of the application server as well. The initialization file by `$$` sets the current shell script process as the exit process. In this way any unexpected shell script interrupt, for example, following a script syntax error, leads to automatic termination of the application server, even if an `Fsclose` command has not been executed explicitly. See the “fsopen” section on page 91 for a complete list of possible `fsopen` options.

Exit status of Finesse function definitions

As mentioned above, the exit status of the `eval` command is returned as the exit status of Finesse functions such as `Fsdisplay`. Usually the `eval` command exit status is the exit status of the command executed after reading the arguments. If the result after reading the arguments is zero, i.e., there is no command to execute, the `eval` exit status is the exit status of the command executed last when reading. Typically in these cases this is a command substitution command. If inside of the `eval` command no command has been executed, given a command without any arguments, for example, the `eval` command exit status is `True`, that is 0.

Basic Finesse commands do not write to standard output if they are exited incorrectly, i.e., with an exit status other than 0. In this case the `eval` command returns the incorrect error state that will be passed back by the subsequent `return $?` as the function definition exit status. Therefore a status check of the Finesse function definition can test whether the underlying basic call is correct.

In the case of `Fsopen` and `Fsdisplay`, status checks are even more rigid. If the `fsopen` command is not terminated successfully, the script is terminated immediately. Continuing with the script would make no sense because additional errors of subsequent commands would follow. Termination also occurs for errors within the `Fsdisplay` command. The main reason for this is to avoid endless loops in cases where `Fsdisplay` is called within a loop while `Fsopen` is terminated. This situation occurs, for example, if someone logsout, while a Finesse application is running. Then `Fsopen` as an ordinary X application is terminated, too, and as a result `Fsdisplay` looses its connection and terminates also. To avoid repeated calls of `Fsdisplay`, e.g. in a `while` loop, the script must be terminated, too, in such a situation.

The fscshinit initialization script

csh shells do not have function definitions like Bourne shells. They do have the `alias` command in which you can map the Finesse commands. This `alias` definition takes the place of the function definition shown in the `fsshinit` script.

```
# Finesse definitions for csh-type scripts

setenv SHELLTYPE csh
if (! $?FINESSEPATH) \
 set FINESSEPATH = /usr/local/finesse
set path = ($FINESSEPATH/bin $path)

alias Fsclose   'eval `fsclose \!*`'
alias Fsdisplay 'eval `fsdisplay \!*` || ( Fsclose;\
exit 1; )'
alias Fsecho    'eval `fsecho \!*`'
alias Fsopen    'eval `fsopen -n "$0" -p $$ \!* \
$argv:q` || exit 1'
alias Fssave    'eval `fssave \!*`'
```

An additional status return as a second command is not necessary here, because when an `alias` definition is called, the command written behind it is executed immediately. Except for a moderately different shell syntax, the `eval` commands mostly correspond to those used in the `fsshinit` file. New-line characters cannot be returned by the above syntax because of the unsatisfactory dealing with line feed in csh shells, which implies that in cases where these characters are important, you shouldn't use a csh shell. Another qualitative difference to the sh definitions is that `fsopen` contains an additional argument `argv:q` which disposes of passing on the shell script arguments to `fsopen`.

The fsperlinit initialization script

```
# Finesse definitions for perl scripts

$ENV{'SHELLTYPE'}='perl';
$ENV{'PATH'} =
"$ENV{'FINESSEPATH'}/bin:$ENV{'PATH'}";

sub Fsclose   { local($a)=join("\" \"",@_) ;
                eval `fsclose "$a"`; !$@ && !$? ; }
sub Fsdisplay { local($a)=join("\" \"",@_) ;
                eval `fsdisplay "$a"` ;
                !$@ && !$? || die "$@Error in
Fsdisplay"; }
sub Fsecho    { local($a)=join("\" \"",@_) ;
                eval `fsecho "$a"`; !$@ && !$? ; }
sub Fsopen    { local($a)=join("\" \"",@_) ;
                eval `fsopen -n $0 -p $$ "$a"` ;
                !$@ && !$? || die "$@Error in
Fsopen"; }
sub Fssave    { local($a)=join("\" \"",@_) ;
                eval `fssave "$a"`; !$@ && !$? ; }
```

The perl initialization script resembles the sh script interface. Here, too, the (perl) `eval` command serves for returning window values to the script. As perl function calls have lists as arguments, those lists by means of the `join` function first have to be translated into command arguments separated by blanks. As with the other initialization scripts you have to take care that arguments are quoted correctly. Perl returns the last system command status in the `$?` variable, the last eval command status in the `$@` variable. Each one of these can be `true` or `false` independent from the other one, therefore both are checked. If `Fsdisplay` or `Fsopen` contains an error the `die` command terminates the script with an error message.

Error messages 5

Finesse tries to anticipate erroneous input and to correct forthcoming problems on its own. When this is not possible or not useful Finesse tries to indicate possible reasons by means of messages to the user to facilitate correcting the error.

Warnings and fatal errors are written to standard error and appear in the terminal window. The following list contains the most important error messages:

- `*progname*: Version mismatch. Exiting.`

  Application server and client program are of a different release. Release numbers can be checked by

  progname `-x`
- `fsopen: Bad licence. Exiting.`

  Finesse licensing is incorrect.
- `fsopen: Connection already open. Exiting.`

  A second application server has been started without terminating the first one.
- `*progname*: No connection setup. Exiting.`

  No connection could be established to the application server, usually because the application server was not running.
- `*progname*: No data. Exiting.`

  No data connection between *progname* and the application server exists, for example, because the application server was killed.
- `*progname*: Bad or missing shell type. Exiting.`

  The environment variable `SHELLTYPE` has been set incorrectly or not at all.
- `*progname*: Syntax error. Exiting.`

  Error on calling the program. When *progname* is `fsdisplay`, syntax error may refer also to an error in the window declaration.

- `fsdisplay: Bad or missing variable name. Exiting.`

  A widget declaration in the window definition is lacking a required variable option.

- `fsdisplay: Bad window name. Exiting.`

  The window name corresponding to the `-n` option has not been found when calling the program.

- `fsdisplay: Unexpected end of window declaration. Exiting.`

  Error in window declaration. Typically a semicolon is missing at the end of the window declaration.

- `fsdisplay: Window declaration syntax error:`*`text`*`.`

  Error while scanning window declaration. The string *text* was not expected at this point.

- `fsopen: No exit process registered.`

  `fsopen` has not registered an exit process whose termination would indicate for the application server to terminate as well. Under certain circumstances this may result in a left over application server. An exit process is registered automatically by the initialization script `fsshinit` or `fscshinit`.

User commands

6

This chapter describes the Finesse commands and their syntax.

Within choices, argument values given in boldface indicate default values.

fsclose

Description

The last command of every Finesse shell script dialog.

`fsclose` closes the application server previously opened by `fsopen`.

Syntax

```
fsclose [ -x ]
```

The `-x` option returns the release number.

fsdisplay

Description

Finesse command to display a window. To call a window the first time `Fsdisplay` needs the window description that contains the complete information on window layout, widgets, default settings, and variable names.

Syntax

`fsdisplay` [*arg* ...]

`-f` *file*

Window declaration from out of a file.

`-m` *text*

Text of the info label.

`-n` *window*

Referencing an existing window.

`-r` *name*`:`*resource*`:`*value*

Referencing an existing window.

`-s { `**`input`**` | close | touch }`

Window status.

`-v` *name=value*

Variable with new value.

`-w` *window_definition*

Window declaration.

`-x`

Obtain release number.

The default window status is `input`, i.e., the window is waiting for input and confirmation after display. The `close` option is possible only in combination with the `-n` option and closes the window referenced by `-n`. The `touch` option opens the desired window but does not admit any input.

Given no arguments `fsdisplay` assumes a window description from standard input.

fsecho

Description

Command writing its arguments as messages to the Finesse echo window, similar to the `echo` command.

Syntax

```
fsecho [ { -x | text |                \
   { -t text |-f file | -e | -i }  \
   [ -r number ] [ -c number ]        \
   [ -p { end | beginning } ] } ]
```

text

Text in the echo window.

`-c` *number*

Number of columns displayed in the window.

`-e`

Clear window.

`-f` *filename*

File in the echo window.

`-i`

Read from standard input.

`-p { `**`end`**` | beginning }`

Position at the beginning or at the end of the text.

`-r` *number*

Number of rows displayed in the window.

`-t` *text*

Text in the echo window.

`-x`

Obtain release number.

fsopen

Description

The first command of a Finesse shell script dialog starting the Finesse application server.

Syntax

`fsopen [` *arg ...* `]`

`Arguments`:

`-n` *name*

Name of calling shell script, set the class name as well as the resource name.

`-o { 1 | 2 | 3 }`

Redirect standard output within script.

`-p` *PID*

Process ID of exit process.

`-r` *file*

Reading resources out of file.

`-x`

Obtain release number.

The initialization scripts automatically set the three arguments `-r`, `-n`, and `-p`. At least one of the possible options is necessary. Therefore, when calling `Fsopen`, usually only the -o option is of interest . Additionally, resource specifications may be given as arguments to `Fsopen`, although this is not recommended. Furthermore, in Bourne shells it is generally useful to set "$@" as an argument in order to pass resource specifications from the shell script command line, if existent, to the Finesse server. In an csh script this is done automatically. So `Fsopen` calls typically look like this:

`Fsopen`	(csh shells)
`&Fsopen(@ARGV)`	(perl scripts)
`Fsopen "$@"`	(sh shells)

fssave

Description

Command to save all variables and corresponding values that are registered in the server up to that moment.

Syntax

```
fssave [ -x ]
```

The variable/value pairs are saved in a file stored in the `$HOME/.finesse` directory and called *scriptname*`.rsc`. In a similar way, when calling a shell script without explicitly giving a resource file, a file with a corresponding name is looked up and read, if existent, in the `.finesse` subdirectory of the home directory.

`-x`

Returns the `Fssave` release number.

User commands

Widget reference

7

This chapter describes the Finesse widget elements and their syntax.

FsCheck

Description

This menu allows selecting several items by switching on the corresponding indicators.

Syntax

`FsCheck -var` *name* [*arg* ...]

`-data` { **`yes`** | `no` }
: Variable and value return, if `-winstat touch` is set.

`-fsbutton` *value*
: Return value of `fsbutton` variable.

`-inputsep` { *char* }$_+$
: Separating characters for `-items` input.

`-items` *item1 item2* ...
: Item names.

`-label` *name*
: Title of check menu.

`-name` *name*
: Instance name for specifying X resources for the widget.

`-nrows` *number*
: Number of rows for alignment.

`-outputsep` *string*
: Separating string for `-items` output.

`-parent` *name*
: Name of parent window. *name* could be the corresponding `FsWindow` window name or the name of any existing `FsForm` container element. By this method the widget can be assigned to the container element given. Lacking the option or any valid name, the widget is assigned to the `FsWindow` element by default.

`-var` *name*[=[=]*value*]
: Variable name with optional default value.

`-winstat` { **`input`** | `touch` }
: Window status after clicking.

FsForm

Description

`FsForm` is a container element where other widgets can be arranged horizontally, vertically, in rows, or in columns. `FsForm` helps you designing your windows as you like.

Syntax

`FsForm -name` *name* [*arg* ...]

`-name` *name*

Instance name of the widget `FsForm`. With this option `FsForm` is made available for referencing as parent of other widgets.

`-nrows` *number*

If `-orientation horizontal`, number of rows.

If `-orientation vertical`, number of columns.

`-orientation { horizontal |` **`vertical`** `}`

Orientation of widgets within `FsForm`.

`-packing {` **`equal`** `| tight }`

Sizing of child widgets.

`-parent` *name*

Name of parent window.

`-spacing` *number*

Distance between widgets in `FsForm`. The `-spacing` option defines the horizontal and vertical distance between widgets.

FsLabel

Description

Widget without interaction. Typically `FsLabel` is used to display user information. The label text is specified by giving a `-label` argument:

Syntax

`FsLabel` [*arg ...*]

`-alignment {  beginning` | **`center`** | `end  }`

Position of text, default is `center`.

`-label` [*varname*=]*text*

Text to be placed in label. Quotes around *text* are needed only when special characters are used. *varname*= is used to assign the label text to a variable that can be changed by subsequent window calls.

`-name` *name*

Instance name for specifying X resources for the widget.

`-parent` *name*

Name of parent window.

FsList

Description

`FsList` generates a list widget that allows choosing entries from the list. The `-include` parameter determines whether the list will appear in the main window (default) or in a separate window. In the latter case, the main window will show a push button and a text widget. Pressing the push button will open a separate window containing the list widget.

Syntax

`FsList -var` *name* [*arg* `...`]

`-alignment {  beginning` | **`center`** | `end  }`
: Position of label text, default is `center`.

`-expert` *name_of_pushbutton*
: Default action for double click on list item.

`-include {` **`yes`** | `no }`
: List display in main window or in separate window.

`-inputsep` { *char* }$_+$
: Separating characters for list items.

`-items` *item1 item2* ...
: List entries.

`-label` *name*
: List title.

`-mode { single` | **`multiple`** `}`
: Select one or several list items.

`-name` *name*
: Instance name for specifying X resources for the widget.

`-nvisible` *number*
: Number of visible list items.

`-outputsep` *string*
: Separating string for `-items` output.

`-packing {` **`equal`** | `tight  }`
: Spacing between push button and text widget. Default value is `equal`, which means both are equally sized.

`-parent` *name*
: Name of parent window.

`-sensitive { yes | no }`

Items selectable or not.

`-spacing` *number*

Defines the horizontal and vertical distance between widgets.

`-title` *name*

Window heading if displayed separately.

`-var` *name*[=[=]*value*]

Variable name with optional default value.

When updating list items, the corresponding action is specified for each list entry by appending one of the following keywords:

Table 5 `Fslist` keywords

Command	Action
a`[add]`	Add item *a*
a`[addselect]`	Add and select item *a*
a`[after]`*b*	Insert item *a* after item *b*
a`[before]`*b*	Insert item *a* before item *b*
a`[delete]`	Delete item *a*
a`[replace]`*b*	Replace item *b* with item *a*
a`[select]`	Select item *a*
a`[topinsert]`	Add item *a* at beginning of list
a`[unselect]`	Unselect item *a*

The keywords `select`, `unselect` and `delete` may be used with a wildcard character "*" to specify all the list items.

Example

```
Fsdisplay -n name -v 'list=a[delete] b[add]'
```

deletes the list entry `a` and adds the new list entry `b` in the window called `name`.

```
Fsdisplay -n name -v 'list=tick[replace]tack'
```

the item `tack` is replaced by the item `tick` in the window `name`.

```
Fsdisplay -n name -v 'list=*[unselect] new[topinsert]'
```

unselects all existing items in the window `name` and adds the item `new` at the beginning of the list.

FsOptionMenu

Description

Widget with menu items to choose “one out of many”.

Syntax

`FsOptionMenu -var` *name* `-items` *item1* [*item2* ...] [*arg* ...]

`-data {` **`yes`** `| no }`
  Variable and value return, if `-winstat touch` is set.

`-fsbutton value`
  Return value of `fsbutton` variable.

`-inputsep {` *char* `}`$_+$
  Separating characters for `-items` input.

`-items` *item1 item2* ...
  Possible options.

`-label` *name*
  Option menu label.

`-name` *name*
  Instance name for specifying X resources for the widget.

`-parent` *name*
  Name of parent window.

`-var` *name* [=[=]*value*]
  Variable name with optional default value.

`-winstat` { **`input`** | `touch` }
  Window status after clicking.

FsPushButton

Description

`FsPushButton` allows you to create your own push buttons that can be equipped with certain properties.

Syntax

`FsPushButton` [*arg ...*]

`-data {` **`yes`** `| no }`

Variable/value return.

`-fsbutton` *value*

Value returned in variable `fsbutton`.

`-label` *name*

Push button label.

`-name` *name*

Instance name for specifying X resources for the widget.

`-nrows` *number*

Number of rows if several push buttons are specified consecutively.

`-packing {` **`equal`** `| tight }`

Width of consecutive push buttons. Eighter same width or width according to button label.

`-parent` *name*

Name of parent window.

`-type { o|a|s|x }+`

Predefined push buttons. After the keyword `-type` any sequence of the above letters is possible. Table 7 shows a list of the letters and their actions.

`-winstat { touch |` **`close`** `}`

Window status after clicking.

Table 6 `FsPushButton` `–type` keywords

Value	Action
`o`	OK button
`a`	Abort button
`s`	`Save as ...` button
`x`	empty space

FsRadio

Description

This menu allows choosing one item out of many by setting the associated indicator.

Syntax

`FsRadio -var` *name* [*arg* ...]

`-data {` **`yes`** `| no }`
: Variable/value return.

`-fsbutton` *value*
: Value returned in variable `fsbutton.`

`-inputsep` { *char* }$_+$
: Separating characters for `-items` input.

`-items` *item1 item2* ...
: Radio items.

`-label` *name*
: Title.

`-name` *name*
: Instance name for specifying X resources for the widget.

`-nrows` *number*
: Number of rows.

`-parent` *name*
: Name of parent window.

`-var` *name* [=[=]*value*]
: Variable name with optional default value.

`-winstat {` **`input`** `| touch }`
: Window status after clicking.

FsSelectionText

Description

Widget for selecting files.

This widget combines two widgets: on the left side a push button to select the files, on the right side a text widget to enter file names. Pressing the push button opens a file selection box. On confirmation the desired file is transferred to the text widget automatically.

Syntax

`FsSelectionText -var` *name* [*arg ...*]

`-alignment { beginning |` **`center`** `| end }`

Position for label text, default is `center`.

`-bdefault` *push_button_name*

Default action for **Return** in text field.

`-dirmask` [*varname*=]*value*

Specifies the directory mask for the file selection box. `dirmask` may be specified as a variable by using the *varname*= option.

`-label` *text*

Label of push button.

`-name` *name*

Instance name for specifying X resources for the widget.

`-packing {` **`equal`** `| tight }`

Spacing of the text widgets. Default value is equal, which means both text lists are equally sized.

`-parent` *name*

Name of parent window.

`-spacing` *number*

Defines the horizontal and vertical distance between widgets.

`-title` *name*

Heading of file selection window.

`-var` *name*[=[=]*value*]

Variable name with optional default value.

FsSeparator

Description

Widget creating a horizontal line in the window. This is useful for graphically separating widgets that are different in their type.

Syntax

`FsSeparator [` *arg* ... `]`

`-line {no|`**`solid`**`|soliddouble|dashed|dasheddouble}`
  Line style of separator. The default is `solid`.

`-name` *name*
  Instance name for specifying X resources for the widget.

`-parent` *name*
  Name of parent window.

FsText

Description

Widget to enter text.

The name variable receives the input text that will be passed back to the shell script. If no label is given, the variable name will be set as a label.

Syntax

`FsText -var` *name* [*arg ...*]

`-alignment {  beginning` | **`center`** | `end  }`

Position for label text, default is `center`.

`-bdefault` *push_button_name*

Default action for **Return** in text field.

`-label` *text*

Label of text widget. If an empty string is used (''), no space is reserved for the label text, moving the input field to the left edge of the window.

`-name` *name*

Instance name for specifying X resources for the widget.

`-packing {` **`equal`** | `tight }`

Spacing of the text widgets. Default value is `equal`, which means both text lists are equally sized.

`-parent` *name*

Name of parent window.

`-sensitive {` **`yes`** | `no }`

Widget may be edited or not.

`-spacing` *number*

Defines the horizontal and vertical distance between widgets.

`-texttype` *type*[`[`[*a*]`,`[*b*]`]`]

Type in the `-texttype` option can take the values `int`, `num`, `digit`, `alpha`, or `alnum`; the bracketed term following it is optional. If the bracketed term is given each one of the values *a* or *b* is optional. The values *a* and *b* restrict the region of valid input in the text widget. They denote lower and upper limit for input values or the minimum and maximum length of the input string, respectively. If one of the limits is missing input is not restricted as far as this limit is concerned.

`-var` *name*[=[=]*value*]

Variable name with optional default value.

FsWindow

Description

Keyword at the beginning of a window declaration.

Syntax

`FsWindow` [*arg* ...]

`-bdefault` *push_button_name*

Default action for **Return** in window.

`-btype { o|a|s|x }`$_+$

Push buttons in the lower bottom of the window. After the keyword `-btype` any sequence of the above letters is possible. Table 7 describes the letters and their actions.

`-name` *name*

Instance name for specifying X resources for the widget.

`-nrows` *number*

Number of rows or columns, respectively. This option defines the number of rows or columns in the window, depending on the `-orientation` value. This option only takes effect if the `-packing` option has a value of `equal`.

`-orientation { horizontal |` **`vertical`** `}`

Widget orientation. This option allows you to define how widgets should be arranged inside the window. Default is one widget below the other (`vertical`).

`-packing {` **`tight`** `| equal }`

Widget size. This option controls the widget size inside the window. Default is `tight`, which shrinks each widget to its minimal size. The `equal` argument results in equal sizes of the widgets.

`-spacing` *number*

Distance between widgets. The `-spacing` option determines the distance between widgets in the window, with composite widgets the distance between the two widgets.

`-title` *name*

Window title.

Table 7 `Fswindow -btype` keywords

Value	**Action**
`o`	OK button
`a`	Abort button
`s`	`Save as ...` button
`x`	empty space

Example programs 8

This chapter contains additional example programs that use various Finesse features.

The master demo examples_sh

The master demo `examples_sh` can be called immediately after Finesse has been installed. It allows comfortably starting of all examples located in subdirectory `sh`, i.e., all Bourne shell examples. This demo itself has some interesting features that justify looking more closely:

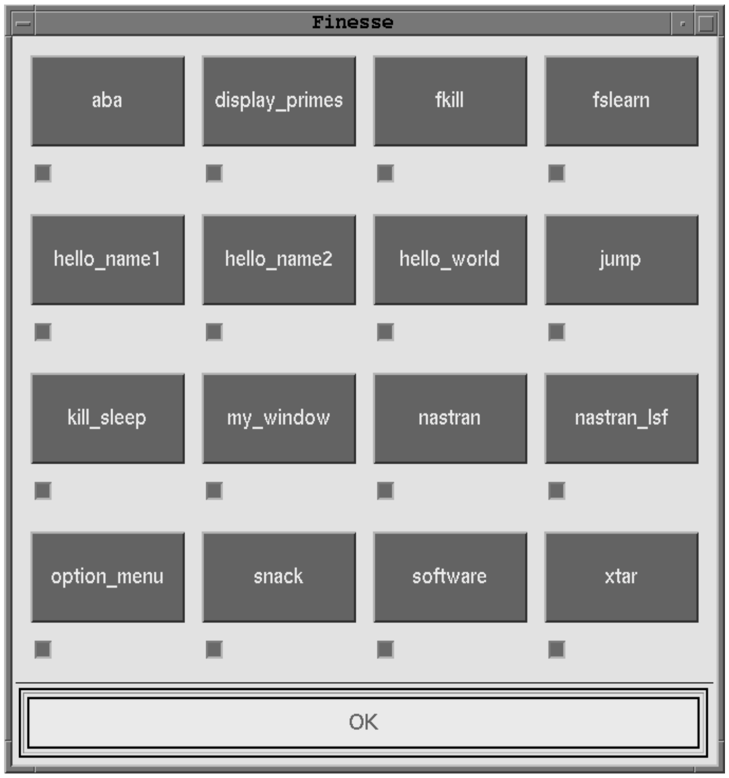


Figure 32 examples_sh

```
 sh script initialization
. ${FINESSEPATH-/usr/local/finesse}/fsshinit

examplespath=${FINESSEPATH-/usr/local/finesse}/exampl
es

shdir=$examplespath/sh
XUSERFILESEARCHPATH=\
$examplespath/app-defaults/%N:$XUSERFILESEARCHPATH
export XUSERFILESEARCHPATH

# define examples and main window

nrows=4; ftag=f; ctag=ck
```

```
j=0; k=0
examples="FsForm -name mainform;"

for i in `ls $shdir`
do
  k=`expr $j / $nrows`
  if expr $j % $nrows > /dev/null
  then
    formvar=
  else
    formvar="FsForm -name $ftag$k
               -orientation horizontal
               -parent mainform;"
  fi
  examples="$examples $formvar
            FsForm -name $ftag$i -parent $ftag$k;
            FsPushButton -label $i -parent $ftag$i
              -name $i -winstat touch;
            FsCheck -var $ctag$i -items 'x'
              -name $ctag$i -parent $ftag$i
              -fsbutton $ctag$i -winstat touch;"
  rmcklabel="$rmcklabel -r $ctag$i[3]:labelString:"
  j=`expr $j + 1`
done

mainwin="
 FsWindow -name main;
 $examples
 FsSeparator;
 FsPushButton -label OK;"

# Finesse dialog

Fsopen
Fsdisplay -w "$mainwin" $rmcklabel
while :
do
  case $fsbutton in
    OK|a) break;;
    $ctag*) updtck= ;;
    *) updtck="-r $ctag$fsbutton[3]:set:True"
       (unset FSREADFILE FSWRITEFILE;
          $shdir/$fsbutton);;
  esac
  Fsdisplay -n main $updtck
done
Fssave
Fsclose
```

Figure 33 Master demo `examples_sh`

The script essentially consists of three parts: short initialization, window declaration, and window dialog. Apart from the usual `sh` initialization, the first part contains the definition of the environment variable `XUSERFILESEARCHPATH` to make sure that examples called from the window can find the X resource files without any user interaction. As mentioned before, this should not be understood as global tactics for setting resources but only as a means justified to facilitate the first Finesse window call for demonstration only.

The second part contains the window declaration. This is done mainly automatically. Window elements are generated using the information of the `ls $shdir` command, i.e. without explicitly naming the examples. For parameterization the name `$i` of the example concerned is used. Each of the examples consists of a `FsPushButton` located above a check menu with selection field. Both are held together in a `FsForm` container element. `$nrows` examples are collected in horizontal containers defined by the `$formvar` shell variable. Those containers are positioned one below the other in the main container `mainform`. The collection of all examples got recursively from the for loop is stored in the `$examples` variable and inserted below in the definition of the main window `mainwin`.

After this, the Finesse dialog follow. The main window first is called once with the window definition, afterwards several times corresponding to the `$fsbutton` variable in a `while` loop. At the first call the selection field labels are erased by the shell variable `$rmcklabel`. This has been prepared in the `for` loop for constructing window elements. When updating resources with `-r $ctag$fsbutton[3]:set:true` the menu field corresponding to an example button that has been pushed can be identified and marked without a label.

The `Fssave` command at the end of the dialog saves the markings. On recalling the window, the examples called earlier are marked. Clicking at the check menu field unmarks a marked example.

Finally, the calling of examples should be explained: This is done in its own subshell after the Finesse intern environment variables `FSREADPIPE` and `FSWRITEPIPE` have been deleted. Those are necessary for internal communication between Finesse server and Finesse clients. As soon as in a shell script several independent Finesse servers shall be started these have to be enabled to communicate with their respective clients using separate Finesse variables. One Finesse server, however, will be sufficient for most of the Finesse dialogs.

The master demo examples_csh

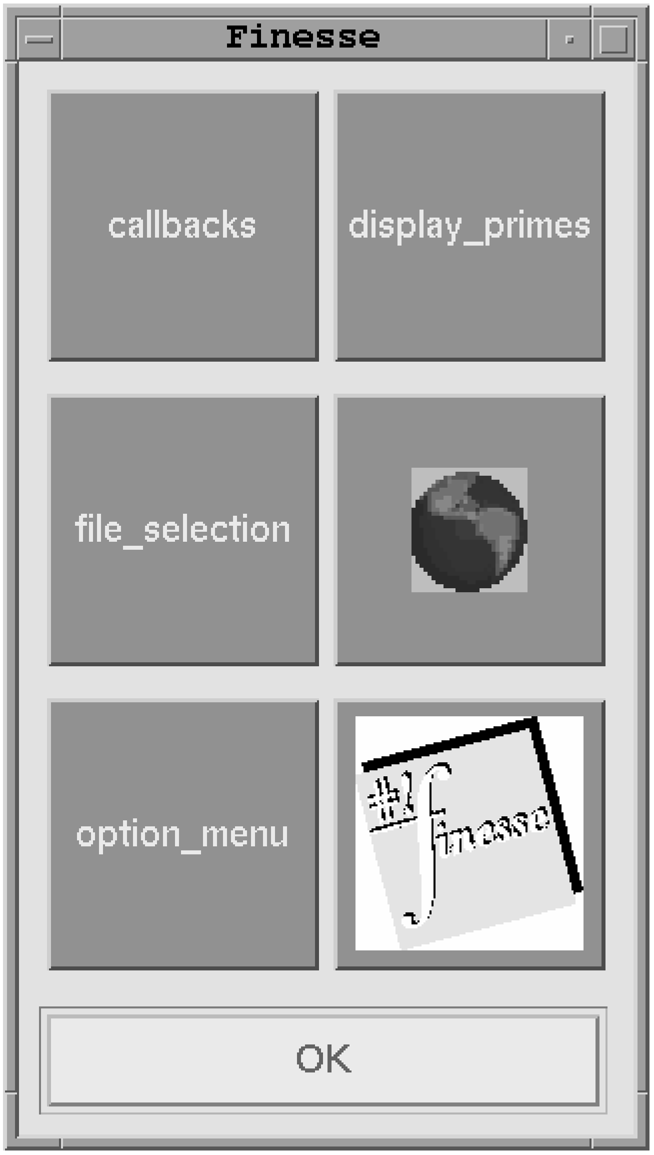


Figure 34 examples_csh

```
#! /bin/csh

# csh script initialization
if (! $?FINESSEPATH) \
  set FINESSEPATH = /usr/local/finesse
source $FINESSEPATH/fscshinit

set examplespath = $FINESSEPATH/examples
set examplespath = /proj/finesse/examples
set appdefpath = $examplespath/app-defaults
set bitmapspath = $appdefpath/bitmaps
set cshdir = $examplespath/csh

# define bitmap/pixmap files

set logo = $bitmapspath/logo
set logoxbm = $bitmapspath/logo.xbm
set logoxpm = $bitmapspath/logo.xpm
set hw = $bitmapspath/hello_world
set hwxbm = $bitmapspath/hello_world.xbm
set hwxpm = $bitmapspath/hello_world.xpm

# make sure XAPPLRESDIR is used to get bitmaps

unsetenv XUSERFILESEARCHPATH
setenv XAPPLRESDIR $appdefpath

# define examples and main window

set nrows = 2; set ftag = f

set j = 0; set k = 0
set examples = "FsForm -name mf;"

foreach i (`(cd $cshdir; ls)`)
  set k = `expr $j / $nrows`
  expr $j % $nrows > /dev/null
  if ( $status == 0 ) then
    set form1 =
    set form2 =
  else
    set form1 = "FsForm -name $ftag$k"
     set form2 = "-orientation horizontal -parent
mf;"
  endif
  set form = "$form1 $form2"
  set push1 = "FsPushButton -label $i -parent
$ftag$k"
  set push2 = "-name $i -winstat touch;"
  set push = "$push1 $push2"
  set examples = "$examples $form $push"
  set j = `expr $j + 1`
end

set lb = "FsPushButton -name s+c -winstat touch"
set lbargs = "-fsbutton s+c -parent $ftag$k;"
set okb = "FsPushButton -label OK;"
set mainwin = \
  "FsWindow -name main; $examples $lb $lbargs $okb"
```

```
# use either bitmap or pixmap files

/bin/rm -f $logo $hw

if (( "`uname`" == "OSF1" ) || \
    ( "`uname`" == "AIX" && "`uname -v`" == "4" ) ||
\
    ( "`uname`" == "SunOS" && \
      "`uname -r | cut -d. -f1`" == "5" )) then
  ln -s $logoxpm $logo; ln -s $hwxpm $hw
else
  ln -s $logoxbm $logo; ln -s $hwxbm $hw
endif

# Finesse dialog

Fsopen
Fsdisplay -w "$mainwin"
while (1)
  switch ($fsbutton)
    case OK:
    case a:
      break
    case s+c:
      Fsecho "Finesse by:"
      Fsecho "science+computing gmbh"
      Fsecho "Hagellocher Weg 71"
      Fsecho "D-72070 Tuebingen"
      Fsecho "Germany"
      Fsecho "Phone: (49) 7071/9457-0"
      Fsecho "Fax:   (49) 7071/9457-27\c"
      set updt = "-r s+c:background:IndianRed3"
      breaksw
    default:
      ( unsetenv FSREADFILE FSWRITEFILE; \
        $cshdir/$fsbutton )
      set updt = "-r
${fsbutton}:background:IndianRed3"
      breaksw
  endsw
  Fsdisplay -n main $updt
end
Fssave
/bin/rm -f $logo $hw
Fsclose
```

Figure 35 master demo `examples_csh`

In the same way as in `examples_sh`, with this script the examples located in the `csh` subdirectory can be called. The script looks a bit simpler, but X bitmaps and X pixmaps are giving it a new aspect. Two of the action buttons show bitmaps or pixmaps instead of text labels. One is the Finesse logo, the other a globe for the `hello_world` example. These pictures are embedded by setting resources in the resource file `Examples_csh` as follows:

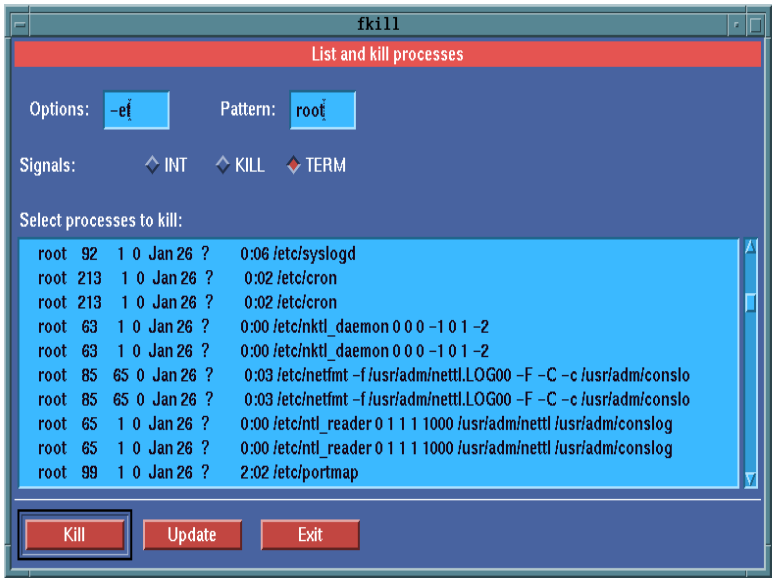
```
*hello_world*labelType: XmPIXMAP
*hello_world*labelPixmap: hello_world
*s+c*labelType: XmPIXMAP
*s+c*labelPixmap: logo
```

Here the shell script differentiates between several cases, because not every X implementation of the different platforms supports multicolored or grayscaled pixmaps, in contrast to bitmaps with only two colors. Depending on architecture the file names `hello_world` and `logo` specified in the resource file are linked to the real files `hello_world.xpm` and `logo.xpm` or `hello_world.xbm` and `logo.xbm`, respectively. These files are located in the `$FINESSEPATH/app-defaults/bitmaps` subdirectory. Setting the environment variable `XAPPLRESDIR` to this path makes sure that the files are found because `$XAPPLRESDIR/bitmaps` is one of the paths followed when looking for bitmap files. You can find more on installation paths for bitmap files in the manpage for `XmGetPixmap` or in vendor specific documentation.

Furthermore, bitmaps can be used as icons to distinguish between different iconified applications. One possibility to link an application to a bitmap is to store in the `$XAPPLRESDIR/bitmaps` directory a bitmap file named exactly as the application itself. Your shipment contains a bitmap file `examples_csh` in the bitmaps subdirectory so that the Finesse logo appears in the icon when iconifying the application. The application `hello_world`, too, will have its proper globe icon by the link set for the label pixmap.

A window for listing and killing of processes

The script fkill script, Figure , is a more elaborate version of the kill_sleep example in the section on lists. The list of processes shown may be defined by both of the text fields in the upper part of the window shown in Figure 36. The input for the `Options:` field may be any argument of the `ps` command on the given platform. It is given to the `ps` command "as is", which in turn generates the entries for the process table shown. The Pattern: text field serves as a filter for the list of processes shown. Example: In a System V environment the string –ef in the `Options:` text field and `root` in the `Pattern:` text field may be used to restrict the processes shown in the list to those of user `root`. The `kill` button kills all selected processes with the signal chosen. The `update` button may be used to refresh the listing.



```
root   92    1  0  Jan 26  ?     0:06 /etc/syslogd
root  213    1  0  Jan 26  ?     0:02 /etc/cron
root  213    1  0  Jan 26  ?     0:02 /etc/cron
root   63    1  0  Jan 26  ?     0:00 /etc/nktl_daemon 0 0 0 -1 0 1 -2
root   63    1  0  Jan 26  ?     0:00 /etc/nktl_daemon 0 0 0 -1 0 1 -2
root   85   65  0  Jan 26  ?     0:03 /etc/netfmt -f /usr/adm/nettl.LOG00 -F -C -c /usr/adm/conslo
root   85   65  0  Jan 26  ?     0:03 /etc/netfmt -f /usr/adm/nettl.LOG00 -F -C -c /usr/adm/conslo
root   65    1  0  Jan 26  ?     0:00 /etc/ntl_reader 0 1 1 1 1000 /usr/adm/nettl /usr/adm/conslog
root   65    1  0  Jan 26  ?     0:00 /etc/ntl_reader 0 1 1 1 1000 /usr/adm/nettl /usr/adm/conslog
root   99    1  0  Jan 26  ?     2:02 /etc/portmap
```

Figure 36 Window for listing and killing processes

fkill script

```
#! /bin/sh
# Demo script for killing processes with sh script

. ${FINESSEPATH-/usr/local/finesse}/fsshinit

# get list of processes and PID column number
#
list_proc()
{
list=`( ps $opt | awk 'NR==1 { i=1; while ($i !~ /^PID$/)
i++;
                     print "col="i;
                     print " proclist='\''"; }
          NR>1 && $0 !~ /awk/ && /'"$patt"'.*/
                        { print $0; }
          END   { print "'\''"; }'
      ) 2>errfile`
if [ -s errfile ] ; then
  cat < errfile; /bin/rm errfile
  exit 1
else
  eval "$list"
  /bin/rm errfile
fi
}

# update list of processes in kill window
#
update_list()
{
  deletelist='*[delete]'
  list_proc
  addlist="`echo "$proclist" |             awk '{ print
$0"[add]" }'`"
  updatearg="-v"
  updatelist="procsinout=$deletelist$cr$addlist"
  windowarg=
  windowlist=
}

# check input
#
case "$1" in
  ?|help)
    echo "Usage: fkill \
['<psoptions>'] [<pattern>]" >&2
    exit 2 ;;
  -*) opt=$1; comm="ps $opt" ; shift ;;
  *)  comm=ps;;
esac
patt="$1"

# get initial list of processes
# and declare window
#list_proc
cr="
"
windef="
  FsWindow   -name killwin
             -title `basename $0`;
  FsSeparator -line no;
  FsForm     -name form1
```

```
             -orientation horizontal
             -packing tight
             -spacing 30;
  FsText     -label Options: -parent form1
             -var opt='$opt' -spacing 10;
  FsText     -label Pattern: -parent form1
             -var patt='$patt' -spacing 10;
  FsRadio    -label Signals:
             -items 'INT KILL TERM'
             -var signal==TERM;
  FsSeparator -line no;
  FsList     -label 'Select processes to kill:'
     -items '$proclist'
             -nvisible 10
             -mode multiple     -inputsep '$cr'
     -outputsep '$cr'
     -var procsinout='';
  FsSeparator;
  FsPushButton -label Kill
               -fsbutton k -winstat touch;
  FsPushButton -label Update
               -fsbutton u -winstat touch;
  FsPushButton -label Exit
               -fsbutton e -nrows 1;"

# Open server and display window
#
Fsopen -o 3 "$@"
# Kill selected processes and/or update list
#
updatearg=
updatelist=
windowarg="-w"windowlist="$windef"

while :
do
Fsdisplay "$windowarg" "$windowlist"\
           "$updatearg" "$updatelist"\
           -m "List and kill processes"\
           -n killwin
 if [ $? -ne 0 ] ; then
  Fsclose; exit 1
 fi
 case $fsbutton in
  k) case $signal in
       INT) SIG=2 ;;
       KILL) SIG=9 ;;
       TERM) SIG=15
     esac
     if [ "$procsinout" != "" ] ; then
       kill -$SIG `echo "$procsinout" |
       awk '{ print $'$col'}'`
```

```
          update_list
        else
          updatelist=
          updatearg=       Fsecho "No processes selected..."
        fi ;;
     u) update_list ;;
     e) Fsclose; exit ;;
    esac
    windowarg=
    windowlist=
   done
```

Resource file

The example listing is followed by a listing of the complete resource file `Fkill` that should be placed in the `app-defaults` directory, typically `/usr/lib/X11/app-defaults`. The resource file also contains the resources from the Finesse resource file `Finesse`, since these backup resources are only set by the application if no app-defaults file is found. If X11R5 is already installed the resources from the Finesse file need not be loaded explicitly but may be included simply by inserting the line

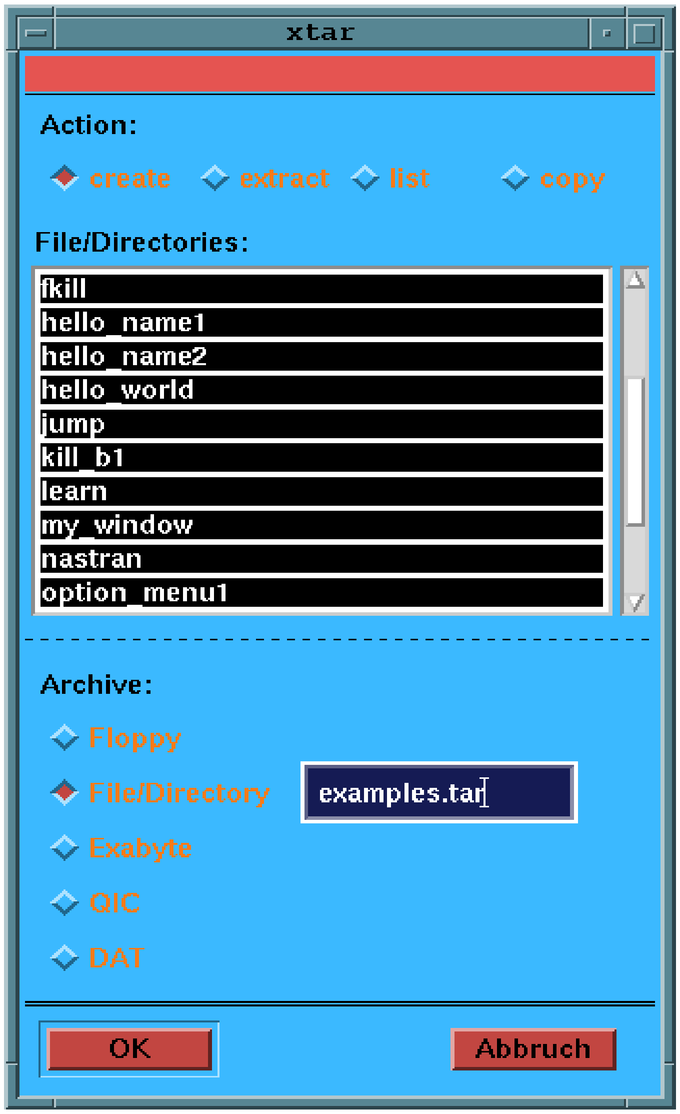
```
#include "/usr/local/finesse/Finesse"
```

at the beginning of `Fkill`.

```
! Resource file Fkill
!
! Resources from file Finesse
!
*font:         -adobe-helvetica-bold-r-normal--14-*iso8859-1
*fontList:     -adobe-helvetica-bold-r-normal--14-*iso8859-1
*title:                 Finesse
*Echo.title:            Echo
*Save.title:            Save*XmText*translations:
#override \n\
Ctrl<Key>A:      beginning-of-line()\n\
Mod1<Key>B:      backward-word()\n\
Ctrl<Key>D:      delete-next-character()\n\
Mod1<Key>D:      delete-next-word()\n\
Mod1<Key>Delete: delete-previous-word()\n\
Ctrl<Key>E:      end-of-line()\n\
Mod1<Key>F:      forward-word()\n\
Ctrl<Key>K:      delete-to-end-of-line() \n\
Ctrl<Key>J:      delete-to-start-of-line()\n\
Ctrl<Key>U: delete-to-end-of-line()delete-to-start-of-line()
*XmText*baseTranslations: #override \n\
Ctrl<Key>A:      beginning-of-line()\n\
Mod1<Key>B:      backward-word()\n\
Ctrl<Key>D:      delete-next-character()\n\
Mod1<Key>D:      delete-next-word()\n\
Mod1<Key>Delete: delete-previous-word()\n\
Ctrl<Key>E:      end-of-line()\n\
Mod1<Key>F:      forward-word()\n\
Ctrl<Key>K:      delete-to-end-of-line() \n\
Ctrl<Key>J:      delete-to-start-of-line()\n\
Ctrl<Key>U: delete-to-end-of-line()delete-to-start-of-line()
!
! application-specific resources!
 background:               SteelBlue
*foreground              White
*XmList*background:       DeepSkyBlue
*XmText*background:       DeepSkyBlue
*XmList*foreground:       Black
*XmText*foreground:       Black
*Info*background:         IndianRed2
*XmPushButton*background: IndianRed3
*form1*XmLabel*width:     60
*form1*XmText*width:      80
*FsRadio*orientation:     XmHORIZONTAL
*FsRadio*XmLabel*width:   120
*XmSeparator*height:      10
*XmPushButton*width:      105
*FsRadio*selectColor:     IndianRed3
```

A window for file archiving

The example script `software` is a prototype Finesse version for use of the Unix `tar` command. Like the previous example it operates with function definitions which improves the structuring of shell scripts. The window generated allows to choose an action that will be applied to all files selected in the list as shown in Figure 37. In the lower part of the window an archive must be given for the files in question. If the archive is a file or a directory its name has to be inserted in the text field to the right of the `File/Directory` toggle button. The command generated from the script is not really carried out but just displayed in the echo window. As in the previous example, also the contents of the resource file are shown. Note that the correct placement of the text field is achieved by setting the `height` resource for the (invisible) separator named `sep2`. Of course, by customizing the device file names for the tape devices to your site and making the file or directory listing more flexible this prototype may be used as a simple OSF/Motif `tar` interface.



Action:

[✓ create] extract list copy

File/Directories:

fkill
hello_name1
hello_name2
hello_world
jump
kill_b1
learn
my_window
nastran
option_menu1

Archive:

Floppy
[✓ File/Directory] examples.tar
Exabyte
QIC
DAT

Figure 37 A window for delivering software

software script

```
#! /bin/sh
#
. ${FINESSEPATH-/usr/local/finesse}/fsshinit#
create=create
extract=extract
list=list
copy=copy
```

```
Floppy=Floppy
Exabyte=Exabyte
QIC=QIC
DAT=DAT
DV=File/Directory
DVe=File/Directories

windef="
  FsWindow        -name tarwin
  -title xtar
          -btype oxa;

  FsSeparator;

  FsForm          -name form1;
  FsRadio         -label Action:
                  -items '$create $extract $list
$copy'                  -parent form1 -var key==$extract;

  FsList          -label $DVe:
          -items '`ls`'
                  -nvisible 10
          -include yes
                  -mode multiple
          -var filelist=='`ls`';
  FsSeparator     -name sep1 -line dashed;

  FsForm -name form2 -orientation horizontal;

  FsRadio         -label Archive:
                  -nrows 5
                  -parent form2
                  -var archive
                  -items '$Floppy $DV $Exabyte $QIC $DAT';

  FsForm -name form3 -parent form2;

  FsSeparator -line no -name sep2 -parent form3;
 FsText -var tarfile -parent form3;
  FsSeparator -line soliddouble;"

Fsopen "$@"

Fsdisplay -w "$windef"
if [ $? -ne 0 -o "$fsbutton" != "o" ] ; then
  Fsclose; exit
fi

# Certain conditions must be met:

# 1) list of files must be given
```

```
testfilelist()
{
  if [ -z "$filelist" ] ; then
    mesg="Please specify $DVe ..."
    return 1   # Error
  else
    return 0   # OK
  fi
}
# 2) For "Copy" or "File/Directory",
# tar file must be given

testtarfile()
{
  if [ "$key" = "$copy" -o "$archive" = "$DV" ] ; then
    if [ -z "$tarfile" ] ; then
      mesg="Please specify $DV ..."
      return 1;  # Error
    else
      return 0;  # OK
    fi
  else
    return 0;    # don't mind
  fi
}
# test conditions

until testfilelist &&
      testtarfile do
  Fsdisplay -n tarwin -m "$mesg"
  if [ $? -ne 0 -o "$fsbutton" != "o" ] ; then
    Fsclose; exit
  fi
done

# Device file for archive

case $archive in
  $Floppy)            dev="f /dev/floppy";;
  $Exabyte)           dev="f /dev/exa";;
  $DAT)               dev="f /dev/dat";;
  $QIC)               dev="f /dev/qic";;
  $DV)                case $key in
                        $copy) dev="$tarfile";;
                        *) dev="f $tarfile";;
                      esac;;
esac

# Now the command may be put together

case $key in
  $create) com="tar cv$dev $filelist";;  $extract)
```

```
com="tar xv$dev $filelist";;
  $list) com="tar t$dev $filelist";;
  $copy)  com="tar cf - $filelist | (cd $dev; tar xf -)";;
esac

# final hint

case $key in
  $create|$extract|$list)
    Fsdisplay -w "FsWindow -btype oa;" -m "Medium inserted?"

    if [ $? -ne 0 -o "$fsbutton" != "o" ] ; then
      Fsclose; exit
    fi;;
esac

# just pretend to do
#

Fsecho "Command:"
Fsecho   $com
Fsecho started.

sleep 5Fsclose
```

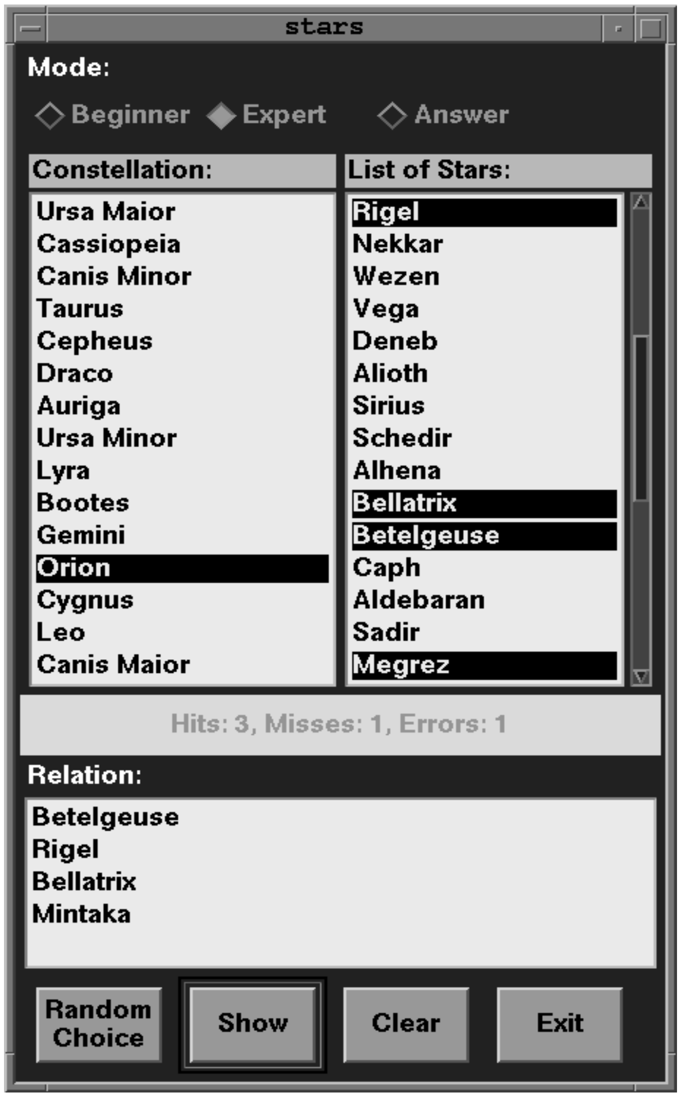
Resource file

The corresponding resource file XTar assumes an X11R5 environment because of the #include statement at the beginning:

```
! Resource file XTar
!
#include "/usr/local/finesse/Finesse"

*background:                      DeepSkyBlue
*Info*background:                 IndianRed2
*Echo*XmPushButton*background:    IndianRed3
*Echo*XmPushButton*width:         100
*XmList*background:               SteelBlue
*XmList*background:               White
*XmPushButton*width:              90
*XmPushButton*background:         IndianRed3
*XmText*background:               MidnightBlue
*XmText*width:                    150
*XmToggleButton*foreground:       Orange
*XmToggleButton*selectColor:      IndianRed3
*sep1*height:                     20
*sep2*height:                     50
```

Example stars

Showing list handling, example `stars` demonstrates additional possibilities of using perl as a script language. Perl is optimal particularly for lists as two of its advantages, using arrays and handling character strings, can be exploited:



Ursa Maior, Cassiopeia, Canis Minor, Taurus, Cepheus, Draco, Auriga, Ursa Minor, Lyra, Bootes, Gemini, Orion, Cygnus, Leo, Canis Maior

Rigel, Nekkar, Wezen, Vega, Deneb, Alioth, Sirius, Schedir, Alhena, Bellatrix, Betelgeuse, Caph, Aldebaran, Sadir, Megrez

Hits: 3, Misses: 1, Errors: 1

Relation:

Betelgeuse
Rigel
Bellatrix
Mintaka

Figure 38 window stars

```
#!/usr/local/bin/perl
$ENV{'FINESSEPATH'} = '/usr/local/finesse'
    if !$ENV{'FINESSEPATH'};
require("$ENV{'FINESSEPATH'}/fsperlinit");
```

```
# read primitive database and prepare lists

require("$ENV{'FINESSEPATH'}/examples/perl/.starsdb")
;
&make_random_lists;

# define user interface

$choicelabel = "Random
Choice";

$windef = "FsWindow -name starswin
           -title stars;
  FsRadio  -var fsmode -label Mode: -winstat touch
           -fsbutton fsmode
           -items 'Beginner Expert Answer';
  FsForm   -name topform
           -orientation horizontal;
  FsList   -label Constellation:
           -inputsep '/'
           -outputsep '/'
           -mode single
           -items '$list_of_consts'
           -var const
           -name const
           -expert show
           -parent topform
           -nvisible 15;
  FsList   -label 'List of Stars:'
           -inputsep '/'
           -outputsep '/'
           -items '$list_of_stars'
           -nvisible 15
           -var fsstars
           -name fsstars
           -expert show
           -parent topform
           -mode single;
  FsLabel  -label
           'Press \\'Random Choice\\' to select
constellation:'
           -name fsresult;
  FsList   -label 'Relation:'
           -inputsep '/'
           -outputsep '/'
           -items ''
           -var rel
           -name rel
           -nvisible 5;
  FsPushButton -label '$choicelabel'
               -name choice -fsbutton Choice
               -winstat touch;
  FsPushButton -label Show -name show
               -winstat touch;
  FsPushButton -label Clear -name clear
               -winstat touch;
  FsPushButton -label Exit -fsbutton a;
";

&Fsopen(@ARGV);
&Fsdisplay("-w", "$windef", "-n", "starswin");

while () {
  if ( $fsbutton eq "fsmode" ) {

      # Change of mode in radio box 'Mode'

      @fsdargs = &get_fsmode_args($fsmode);

  } elsif ( $fsbutton eq "Choice" ) {

      # Button 'Choice' was pressed

      if (( $fsmode eq "Beginner" ) ||
          ( $fsmode eq "Expert" )) {
          @fsdargs = &get_choice_args($fsmode);
      }
  } elsif ( $fsbutton eq "Show" ) {

      # Button 'Show' was pressed

      if ( $fsmode eq "Beginner" ) {
          @fsdargs =

&get_beginner_args($allstars{$const},$fsstars);
      } elsif ( $fsmode eq "Expert" ) {
          @fsdargs =

&get_expert_args($allstars{$const},$fsstars);
      } elsif ( $fsmode eq "Answer" ) {
          @fsdargs =
&get_answer_args($const,$fsstars);
      }
  } elsif ( $fsbutton eq "Clear" ) {
          @fsdargs = @clear;
  } else {
      last;
  }
  &Fsdisplay("-n", "starswin", @fsdargs);
}

&Fsclose;

#
```

```
# functions
#

sub make_random_lists {

# make list of stars and constellations

    srand(time|$$);

    foreach $value (values (%allstars)) {
        @all_stars = (@all_stars, split('/',
$value));
    }
    @all_consts = keys(%allstars);

# make lists random for window declaration

    push(@rand_stars, splice(@all_stars, rand
@all_stars, 1))
        while (@all_stars);
    $list_of_stars = join('/', @rand_stars);

    push(@rand_const, splice(@all_consts, rand
@all_consts, 1))
        while (@all_consts);
    $list_of_consts = join('/', @rand_const);

# prepare clearing of lists

    @clear = ("-v", "const=*[unselect]",
              "-v", "fsstars=*[unselect]",
              "-v", "rel=*[delete]",
              "-r", "fsresult:labelString:");
}

sub get_fsmode_args {

    # prepare window setup according to selected mode

    if ( $_[0] eq "Beginner" ) {
        $fslabel = "Push Choice to Select
Constellation:";
        $constpol = "XmBROWSE_SELECT";
        $starspol = "XmBROWSE_SELECT";
    } elsif ( $_[0] eq "Expert" ) {
        $fslabel = "Push Choice to Select
Constellation:";
        $constpol = "XmBROWSE_SELECT";
        $starspol = "XmEXTENDED_SELECT";
    } elsif ( $_[0] eq "Answer" ) {
        $fslabel = "Select Constellation and Stars:";
        $constpol = "XmEXTENDED_SELECT";
```

```
            $starspol = "XmEXTENDED_SELECT";
        }

        ("-r", "fsresult:labelString:$fslabel", @clear,
         "-r", "const:selectionPolicy:$constpol",
         "-r", "fsstars:selectionPolicy:$starspol");
}

sub get_choice_args {

    #  make random choice in list of stars

    if ( $_[0] eq "Beginner" ) {
        $fslabel =
            "Select corresponding main star and
press 'Show'";
    } elsif ( $_[0] eq "Expert" ) {
            "Select corresponding stars and press
'Show'";
    }
    ("-v", "rel=*[delete]", "-v",

"const=*[unselect]/$rand_const[rand(@rand_const)]",
     "-r", "fsresult:labelString:$fslabel");
}

sub get_answer_args {

    # determine answer for selections

    local(%MARK, $entry, @chosenstars);

    # begin with chosen constellations

    if ("$_[0]") {
        @const = split(/\//,$_[0]);
        foreach (@const) {
            @stars = split(/\//, $allstars{$_});
            push(@chosenstars, @stars);
        }
    }

    # add chosen stars not yet processed

    if ("$_[1]") {
        @stars = split(/\//, $_[1]);
        grep($MARK{$_}++, @askedstars);
        @remainingstars = grep(!$MARK{$_}, @stars);
        push(@chosenstars, @remainingstars);
    }

    foreach (@chosenstars) {
        $entry = "$entry$_: $genitive{$_}/";
    }

    ("-v", "rel=*[delete]/$entry*[unselect]");
}

sub get_beginner_args {

    # compare main star with guess

    if ("$_[0]" && "$_[1]") {
        ($answer) = split(/\//,$_[0]);
        if ( "$answer" eq "$_[1]" ) {
            $fslabel = "Congratulations!" ;
        } else {
            $fslabel = "Sorry." ;
        }
        ("-v", "rel=*[delete]/$answer:
$genitive{$answer}/*[unselect]",
         "-r", "fsresult:labelString:$fslabel");
    } else {
        $fslabel =
            "Select constellation and main star
first." ;
        ("-r", "fsresult:labelString:$fslabel");
    }
}

sub get_expert_args {

    # compare all stars of const with guess

    if ("$_[0]" && "$_[1]") {
        local(%MARK);

        @answer = split(/\//,$_[0]);
        @guess = split(/\//,$_[1]);

        grep($MARK{$_}++, @answer);
        $hits = grep($MARK{$_}, @guess);
        $misses = @answer - $hits;
        $errors = @guess - $hits;
        $fslabel =
            "Hits: $hits, Misses: $misses, Errors:
$errors";

        ("-v", "rel=*[delete]/$_[0]/*[unselect]",
         "-r", "fsresult:labelString:$fslabel");
    } else {
        $fslabel = "Select constellation and stars
first." ;
        ("-r", "fsresult:labelString:$fslabel");
    }
}
```

Figure 39 listing stars

The first part of the script essentially contains window definition and Finesse dialog. At the beginning the require function reads a primitive data base file assigning stars to constellations in the form

```
$allstars{"Orion"}="Betelgeuse/Rigel/Bellatrix/
Mintaka";
```

The remainder of the script contains various subroutines that are called during the dialog. Thus using subroutines the dialog part is simple and well structured: after the window is displayed for the first time the course splits according to the push button pressed (`$fsbutton`) and an action is executed corresponding to the mode set currently (`$fsmode`). Every action is defined in a perl subroutine and returns as a result the array of arguments for the next `Fsdisplay` call. The last array expression of a subroutine is assigned to array `fsdargs` as return value.

One more of the important characteristics of perl, besides arrays, is the large number of built-in functions; in particular, the randomized sequence of stars and constellations (in subroutine `make_random_lists`) as well as the random selection of a single star in the question modes `beginner` and `expert` (subroutine `get_choice_args`) is possible only in perl. The simple evaluation of correct, missing and wrong star selections in the expert mode (subroutine `get_expert_args`) shows arithmetic qualities in perl that are far superior to shell calculations.