XF

Design and Implementation of a Programming Environment for Interactive Construction of Graphical User Interfaces

Technische Universität Berlin
Institut für Angewandte Informatik
Fachbereich 20 (Informatik)
Lehrgebiet Softwaretechnik

vorgelegt von:
Sven Delmas
garfield@cs.tu-berlin.de **Abstract**

XF is an integrated programming environment that supports the development of graphical user interfaces. The described programming system enables developers who don't want to dive into the complex task of window system programming, to construct sophisticated graphical applications in a very short time. *XF* takes advantage of *Tk*, a $Motif^{TM}$-like widget set that is accessible through *Tcl*, a very efficient interpreted programming language. The flexibility of this approach allows to build or modify application programs while they are running. This makes it possible to test the effect of modifications immediately without incurring costly recompilation cycles.

Many users of *XF* have reported that the tool is very easy to use, and allows rapid construction of graphical interfaces. Nevertheless it does not restrict the developer when he wants to manipultate the graphical interface in more detail. Support for libraries of reusable interface components and functions leads to further reductions of the development time, and also supports a standardized look & feel. It is possible to merge external code into the application, and to reuse pre-existing code.

Berlin, 19 March. 1993

Acknowledgements

I owe gratitude to many people who supported and inspired me when implementing the program and writing this paper.

Special thanks to Wilfried Koch and Axel Mahler who both attended my work and made suggestions for *XF* and the paper. Andreas Lampen, who also contributed many ideas, both for the paper and for *XF*. Many people around the world who have used *XF* and reported bugs, suggestions and comments on the program.

The *BYO* team, that developed *BYO* which was the trigger for the development of *XF*. This program contains the basic concept for *XF*, and showed how easy it is to write a powerful interface builder based upon *Tcl/Tk*.

Very special thanks to John Ousterhout, the inventor of *Tcl/Tk*. The power and flexibility of these packages made the development of *XF* possible (and his answers to my questions and suggestions).

My sister Britta. She corrected my numerous spelling errors, and my lousy English, partially supported by ispell. And finally a big (unnamed) soda pop producer, for providing the coffein that was necessary to keep my brain going (The choice of an old generation).

Sven Delmas

Contents

1 Introduction **6**
1.1 Roadmap . . . 6
1.2 Why use an interface builder ? . . . 6
1.3 Why use Tcl/Tk ? . . . 7
1.4 Why use XF ? . . . 8

2 The Design of XF **9**
2.1 Interface builders . . . 9
2.2 Existing interface builders . . . 10
2.3 The XF design . . . 12

3 Tcl/Tk **14**
3.1 Tcl . . . 14
3.1.1 Syntax . . . 14
3.1.2 Datatypes . . . 14
3.1.3 Variables . . . 14
3.1.4 Commands . . . 15
3.1.5 Quoting . . . 15
3.2 Tk . . . 15
3.2.1 Widget classes . . . 15
3.2.2 Inserting widgets . . . 15
3.2.3 Widget commands . . . 16
3.2.4 Widget configuration . . . 16
3.2.5 Geometry handling . . . 16
3.2.6 Other Tk commands . . . 18
3.2.7 XF relevant commands . . . 18

4 Using XF **20**
4.1 The first steps . . . 20
4.1.1 Start . . . 20
4.1.2 Inserting widgets into the widget tree . . . 21
4.1.3 Layouting widgets . . . 22
4.1.4 Configuring widgets . . . 22
4.1.5 Procedures . . . 25
4.2 Advanced features . . . 25
4.2.1 Templates . . . 25
4.2.2 Toplevel windows . . . 26
4.2.3 Source modules . . . 26
4.2.4 Levels for procedures and bindings . . . 27

5 The Implementation of XF 29
5.1 The generated code . . . 30
5.2 XF constraints . . . 31
5.2.1 Restrictions . . . 31
5.2.2 Pitfalls . . . 32
5.3 Extending XF . . . 32
5.3.1 Supporting new widgets . . . 32
5.3.2 The procedure XFExternalInitProc . . . 33
5.3.3 XF startup file . . . 33
5.3.4 Adding procedures named XFProc . . . 33

6 Conclusion 34
6.1 Epilogue . . . 34
6.2 Missing features . . . 34

A External Tools 36
A.1 Edge . . . 37
A.2 Editors . . . 38
A.2.1 TkEmacs . . . 38
A.2.2 Emacs . . . 38
A.2.3 vi . . . 38
A.3 ShapeTools . . . 39
A.3.1 Saving procedures . . . 39
A.3.2 Retrieving modules . . . 39
A.4 xfappdef . . . 40
A.5 xfhardcopy . . . 41
A.5.1 Hardcopy commands . . . 41
A.6 xfhelp . . . 42
A.6.1 Help pages . . . 43
A.7 xfpixmap . . . 44
A.8 xftutorial . . . 45
A.8.1 The script files . . . 45

B XF User's Guide 46
B.1 Main . . . 47
B.1.1 XFProcMain . . . 47
B.2 File . . . 48
B.2.1 XFProcFileEnterTCL (Enter TCL code) . . . 48
B.2.2 XFProcFileInsert (Insert...) . . . 48
B.2.3 XFProcFileLoad (Load...) . . . 49
B.2.4 XFProcFileNew (New) . . . 49
B.2.5 XFProcFileQuit (Quit) . . . 50
B.2.6 XFProcFileSave (Save) . . . 50
B.2.7 XFProcFileSaveAs (Save as...) . . . 50
B.3 Configuration . . . 51
B.3.1 XFProcConfAddCurrentItem . . . 51
B.3.2 XFProcConfBinding (Binding) . . . 51
B.3.3 XFProcConfBindingAll (Binding for all widgets) . . . 52
B.3.4 XFProcConfBindingClass (Binding for selected Class) . . . 52
B.3.5 XFProcConfConfigure . . . 52
B.3.6 XFProcConfGeometryDefault . . . 52
B.3.7 XFProcConfInsertTemplate . . . 52
B.3.8 XFProcConfInsertWidgetDefault . . . 52
B.3.9 XFProcConfInsertWidgetConfig . . . 52

B.3.10 XFProcConfLayout (Layout) . . . 53
B.3.11 XFProcConfPacking (Packing) . . . 54
B.3.12 XFProcConfParametersDefault (Parameters) . . . 54
B.3.13 XFProcConfParametersGeneral (Parameters (general)) . . . 55
B.3.14 XFProcConfParametersGroups (Parameters (widget groups)) . . . 55
B.3.15 XFProcConfParametersSmall (Parameters (small)) . . . 56
B.3.16 XFProcConfParametersSpecial (Parameters (special)) . . . 56
B.3.17 XFProcConfPlacing (Placing) . . . 56
B.4 Edit . . . 58
B.4.1 XFProcEditClearCut (Clear Cutbuffer) . . . 58
B.4.2 XFProcEditCopy (Copy) . . . 58
B.4.3 XFProcEditCut (Cut) . . . 58
B.4.4 XFProcEditDelete (Delete) . . . 58
B.4.5 XFProcEditLoadCut (Load Cutbuffer) . . . 58
B.4.6 XFProcEditLoadTemplate . . . 58
B.4.7 XFProcEditMakeAProc (Make a procedure) . . . 58
B.4.8 XFProcEditPaste (Paste) . . . 59
B.4.9 XFProcEditSaveCut (Save Cutbuffer) . . . 59
B.4.10 XFProcEditSaveCutAsTemplate (Save Template (cut buffer)) . . . 59
B.4.11 XFProcEditShowCut (Show Cutbuffer) . . . 59
B.5 Programming . . . 61
B.5.1 XFProcProgCommands (Commands) . . . 61
B.5.2 XFProcProgEditScript (Edit script) . . . 63
B.5.3 XFProcProgEndSrc (End source) . . . 63
B.5.4 XFProcProgErrors (Error status) . . . 63
B.5.5 XFProcProgGlobals (Global variables) . . . 64
B.5.6 XFProcProgProcs (Procedures) . . . 65
B.5.7 XFProcProgShowScript (Show script) . . . 66
B.5.8 XFProcProgStartupSrc (Startup source) . . . 66
B.5.9 XFProcProgWidgetTree (Widget tree) . . . 67
B.6 Misc . . . 68
B.6.1 XFProcMiscAliases (Aliases) . . . 68
B.6.2 XFProcMiscAppDefaults (Application defaults) . . . 68
B.6.3 XFProcMiscEdge (Widget tree (edge)) . . . 68
B.6.4 XFProcMiscHardcopy (Hardcopies) . . . 68
B.6.5 XFProcMiscModules (Module structure) . . . 69
B.6.6 XFProcMiscPixmaps (Pixmaps) . . . 70
B.6.7 XFProcMiscSaveEdge (Dump tree (/tmpPath/....grl)) . . . 70
B.6.8 XFProcMiscTestProgram (Test program) . . . 70
B.7 Options . . . 71
B.7.1 XFProcOptionsBindings (Bindings) . . . 71
B.7.2 XFProcOptionsGeneral (General options) . . . 72
B.7.3 XFProcOptionsIconBar (Iconbar configuration) . . . 74
B.7.4 XFProcOptionsInterpreter (Interpreter options) . . . 74
B.7.5 XFProcOptionsMenuBar (Menubar configuration) . . . 74
B.7.6 XFProcOptionsPathFile (Path/file names) . . . 75
B.7.7 XFProcOptionsSaveClassBindings (Save class bindings) . . . 77
B.7.8 XFProcOptionsSaveModuleList (Save module list) . . . 77
B.7.9 XFProcOptionsSaveOptions (Save options) . . . 77
B.7.10 XFProcOptionsSavePositions (Save window positions) . . . 77
B.7.11 XFProcOptionsSource (Source options) . . . 78
B.7.12 XFProcOptionsVersion (Version control options) . . . 79
B.7.13 XFProcOptionsWindow (Window options) . . . 82

B.8 Help . . . 84
B.8.1 XFProcHelpAbout (About) . . . 84
B.8.2 XFProcHelpHelp . . . 84
B.8.3 XFProcHelpTutorial (Tutorial) . . . 84

C Templates 85

C.1 Combined . . . 86
C.1.1 CanvasLS, CanvasRS . . . 86
C.1.2 EntryL, EntryLLS, EntryLS, EntryS . . . 87
C.1.3 HypertextLS, HypertextRS . . . 88
C.1.4 ListboxLS, ListboxRS . . . 89
C.1.5 PhotoLS, PhotoRS . . . 90
C.1.6 TextLS, TextRO, TextROLS, TextRORS, TextRS . . . 91
C.1.7 TkEmacsLS, TkEmacsRS . . . 92
C.2 Procedures . . . 93
C.2.1 AlertBox, AlertBoxFd, AlertBoxFile . . . 93
C.2.2 ClearList, ClearText . . . 94
C.2.3 ColorBox . . . 95
C.2.4 CursorBox . . . 96
C.2.5 FSBox . . . 97
C.2.6 FdInList, FileInList, FdInText, FileInText . . . 99
C.2.7 FontBox . . . 100
C.2.8 IconBar . . . 101
C.2.9 InputBox . . . 104
C.2.10 IsADir, IsAFile, IsASymlink . . . 106
C.2.11 KeysymBox . . . 107
C.2.12 MakeMButton . . . 108
C.2.13 MenuBar . . . 109
C.2.14 ReadBox . . . 111
C.2.15 TextBox, TextBoxFd, TextBoxFile . . . 112
C.2.16 YesNoBox . . . 113
C.2.17 fileselect . . . 114
C.3 Widgets . . . 115
C.3.1 MListbox . . . 115
C.3.2 Menubar . . . 116
C.3.3 OptionButtonE, OptionButtonL . . . 117
C.3.4 Popup1, Popup2, Popup3, PopupC-1, PopupM-1, PopupS-1 . . . 118

Chapter 1

Introduction

Developing graphical user interfaces using graphical environments like the X *window system*^{TM} provides the programmer with high flexibility, much functionality, but often big confusion as well. Facing thousands of pages of documentation, most programmers decide to leave the programming of graphical user interfaces to "experts". Even these experts often use only parts of the functionality the window system offers, and above all, developing a graphical user interface is a time consuming task.

While everybody wants a graphical interface for his applications, as they make the handling easier and reduce the learning effort, nobody likes implementing applications with graphical interfaces. A programmer who implements a graphical user interface by hand, writes or changes some interface code, compiles and tests it. Afterwards, he often finds out that the changes had the wrong impact on the interface. The programmer has to handle numerous variables and statements containing widget initialization, configuration and geometry information. The resulting application code mainly implements the interface, and the functionality of the application is thrust into the background. Studies have shown that about 50 to 80 per cent of the code of an application with a graphical interface are used to implement the graphical interface.

The introduction of OOP techniques for interface building makes the implementation of graphical interfaces much easier. Toolkits like Interviews[10], Theseus++[4] and others can drastically reduce the complexity and size of the interface code. But for many developers, even this reduced complexity of the interface code is not acceptable. They don't want to care about the implementation of the interface, they want to work on the application itself. For those users, a graphical interface builder can be the solution.

1.1 Roadmap

The first chapter, describes the motivation behind the development of *XF* and the aim of this thesis. The second chapter describes the design of *XF* and compares it with other interface builders. The third chapter chapter provides a brief introduction into *Tcl/Tk*. Users that are familiar with *Tcl/Tk* can skip this chapter. Subsequently the concepts and the general handling of *XF* is described, followed by a short view inside of *XF*. Finally, there is a conclusion, containing a short overview of the current situation and the future development. The appendix contains a description of the external Tools which are used by *XF*, a explanation of the dialog components that are provided and a description of all templates that are part of the *XF* distribution.

1.2 Why use an interface builder ?

One way to make the development of graphical interfaces more attractive and more efficient is to use a graphical tool that supports this development process. Such a tool should provide access to the facilities of a widget set combined with an easy way to handle those widgets, and to implement the application's functionality.

The design of a "good" graphical interface is a hard job. And writing the interface code "by hand" changes the focus of the developer from the implementation of the application's functionality to the implementation of the application's interface. The main work (and the biggest piece of code) when developing a program with a graphical user interface is often writing the interface code. The code that implements the functionality of the program is only a fraction of the resulting code (as written above, studies have shown that about 50-80 per cent of the application's code with a graphical interface are used to implement the graphical interface). The complexity is, among other reasons, caused by geometry dependencies between widgets and numerous widget parameterizations that were often found by trial and error instead of design. Changing this code often leads to trouble and sometimes to a re-implementation. This makes it hard to react to changed requirements that occur during the development of the application.

A frequently chosen way to get a more flexible handling of the design and implementation of a graphical user interface is the use of graphical interface builders. These tools usually provide access to a certain widget set. The development is supported by features like:

- Widget creation.
  The different widget types can be inserted into the resulting interface by moving (dragging) them into the workspace.
- Widget parameterization.
  For the various widget types, special configuration dialogs give access to the widget parameters (like colors, fonts etc.).
- Widget layout.
  The layout of the widgets can be modified interactively.
- Implementation of application functionality
  Code implementing the functionality of the program can be written and bound to widgets.

The first three topics mainly describe the design and implementation of the static aspects for the resulting application. The functionality supporting this can be found in almost every interface builder. But an interface builder must also support the user at the implementation of the functionality and, more generally, the dynamic behavior of the application program. Without a way to implement functionality, the interface builder is more a prototype tool than an real interface builder. In fact, the ability to implement functionality makes an interface builder more an application builder for graphical applications.

The support of the definition of dynamic aspects of the user interface is often unsatisfying. It may happen that the re-usability of the generated code is restricted (caused by the fact that the automatically generated code may not be changed by the user), or testing of the created functionality is only possible after a time consuming building process.

It is important that the resulting code is fast and simple to be handled. This means it should be possible to adapt the code by hand, without losing the support of the interface builder for this program. The turn around time should not be too long, too. A change in the application program should not cause a long rebuild. Instead, the change must be visible and operational immediately. One (costly) way to get short turn around times is to always buy the most recent hardware. Another way can be to use... *Tcl/Tk* in combination with *XF*.

1.3 Why use Tcl/Tk ?

Tcl/Tk is a package containing a shell-like interpreted language (*Tcl*), and a $Motif^{TM}$-like[1] widget set (*Tk*). It runs under various $UNIX^{TM}$[2] environments, and is based upon the *X window system*^{TM}[3][19]. The handling of *Tcl* as an implementation language is very easy, and *Tk* provides an easy access to the implementation of graphical user interfaces. Compared to interfaces implemented with traditional widget sets, the code written in *Tk* is usually much smaller, while the performance is in most cases at least as good

[1] MOTIF is a registered trademark of the Open Software Foundation
[2] UNIX is a registered trademark of AT&T Bell Laboratories
[3] X Window System is a registered trademark of The Massachusetts Institute of Technologie

as with the traditional widget set. Another advantage is that *Tcl/Tk* code is interpreted, so the turn around time is zero. Changes to *Tcl/Tk* code (including the graphical interface) can be made while the application code is interpreted.

Functionality not available under *Tcl/Tk*, or very speed dependent tasks can be added to *Tcl/Tk* using the very simple C-interface. New widgets are also added by this mechanism. Furthermore there already exists a wide range of additional widgets.

The community of *Tcl/Tk* users is constantly growing. Already, a great number of programs is freely available, and there is an increasing number of commercial applications, as well.

1.4 Why use XF ?

XF is a programming environment for *Tcl/Tk* applications. Although the usage of *Tcl/Tk* is very simple, there is a big number of options and commands that have to be mastered. Using a program like *XF* to compose and configure the interface of an application can reduce the development time, and result in better code. Currently, *XF* only supports the development of *Tcl/Tk* based applications, but it is possible that future versions also support other widget sets and/or languages. The main design aims for *XF* are:

- Rapid construction of interactive user interfaces.
  The interface designer can access all widgets by selecting them from a list. Layout and configuration of widgets are supported by interactive dialogs. The implementation and binding to the interface of the application's functionality are supported as well. These features cover the basic functionality that each interface builder usually has.

- Immediate access to the resulting interface.
  The work space where the application is built *is* the resulting application (interface and functionality). At any point of the construction, the user can use and test the functionality of the resulting application immediately. This is possible due to the great flexibility of *Tcl/Tk*.

- High flexibility for later changes.
  All aspects of the resulting application (interface and functionality) can be changed with *XF*, or by editing the generated *Tcl*-code directly. This includes the possibility to merge external *Tcl/Tk* code into the application.

- Support for group development.
  Complex programs are very often developed in groups. While the interface is developed by one part of the group it can happen that the functionality of the program is developed by another part. Much attention was given to the support of distributed development.

- Support for "standard" interfaces.
  The development of complex dialogs and functionality is a time consuming task. To reduce the development time and to have a common interface style among different applications, *XF* supports the storing, retrieval and sharing of interface and function components.

- Fun when developing with *XF*.
  Probably the most important feature of *XF* is to have fun when developing. This means that the beginner gets enough support to be able to build an application, while the expert can manipulate every aspect of the resulting application if he wants to. It is possible to develop simple interfaces just by playing around with *XF*. This does not mean that users *must* or only *can* play around with *XF*, but it is not necessary to read thousands of pages to be able to use *XF*. For more complex programs the user will need good planning, and careful study of the *Tcl/Tk/XF* related documentation.

Chapter 2

The Design of XF

The development of graphical interfaces becomes more and more important, as more and more graphical oriented computers are used. Many users are still working with traditional commandline tools, and only use the graphical capabilities of their computers to display several command shells. Of course, this may be appropriate for some tasks, and users but using an application with a graphical interface usually improves the user's productivity. Usually graphical interfaces provide a more intuitive access to applications. This is possible by the use of graphical symbols, color, various fonts etc.. This does not mean that graphical interfaces must be totally different from traditional interfaces, but the graphical environment gives more freedom for the design of the application.

The range of applications that are supplied with a graphical interface goes from complex software packages (like word processors) to small interface components (like alert boxes) for "toy" applications. The complexity of the graphical environment makes it necessary that developers get some kind of support. First, to allow a fast development of applications, and second to support the design of "good" graphical interfaces.

The developer of a complex software package usually needs support for the design of a consistent interface. It is important that the process of the implementation is supported. Changeability of the design, support for product maintenance and project management are important.

On the other hand, the existance of commonly used and well-known traditional applications makes it interesting for the user to create graphical frontends for existing commandline oriented tools. An example for this kind of encapsulation of traditional applications into a graphical frontend is the *HP Encapsulator*^{TM} [3] and [5]. This is a language that can be used to embed existing non graphical applications into a graphical environment. Such an interface must be built easily, and the resulting code must be easy to handle. A programmer implementing such an interface usually does not want to learn how to use complex windowing systems.

2.1 Interface builders

To support the development of graphical user interfaces, the usage of graphical development tools seems reasonable. The design of a graphical interface is a very important but also a very difficult task. Creating the code that implements the interface by constructing it interactively in a graphical tool can reduce the implementation expense. Great parts of the code implementing the interface can be generated automatically by the interface builder. This allows the developer to pay more attention to the design of the graphical interface. The inexperienced user has the chance to construct an interface by playing around, while the experienced user is supported to access the complete functionality that the graphical environment offers. It can also prevent coding bugs, as the automatically generated code is usually more structured, and better tested.

There exists a wide range of interface builders that are based upon different widget sets and on different implementation languages. Most of the interface builders provide the same basic functionality that can be summarized to the following four features:

- Object (widget) creation.
  The objects that form the interface of the application program can be created and deleted.
- Object (widget) parametrization.
  The objects that form the interface of the application program can be parameterized.
- Object (widget) layout.
  The objects that form the interface of the application program can be arranged.
- Functionality implementation.
  The code that implements the functionality of the application program can be created, or at least the automatically created code is prepared to be enriched by the functionality code.

But there are also some differences between the various interface builders. The differences occure because the interface builders support different widget sets and different languages. The targeted groups of users and application types vary as well.

- The implementation language.
  This point is not important for the automatically generated code, but it is important for the code that has to be written by hand. It also implies the general concept of the interface builder, as there exist compiled languages and interpreted languages. Compiled languages can lead to long turn around times, while interpreted languages can cause slow program execution.
- The supported widget set.
  This difference is not very important. Most widget sets provide the same basic functionality, but the supported widget set influences the look & feel of the interface builder and of the created applications.
- The level of integration.
  The application builders differ in the kind of objects they handle. In some interface builders, these objects are simulations of toolkit elements. The objects do not represent objects of a concrete widget set. Other interface builders represent objects of a concrete widget set, but the objects are not functional. The resulting application is in a special design mode. And another type of interface builders manipulate concrete objects that are fully functional. The resulting application is functional while it is built, there is no special design mode.

2.2 Existing interface builders

The following list of interface builders contains commercial and academic interface builders. The list is not complete, but it gives a short impression of some of the available tools.

- DEC VUIT (Digital Equipment Corporation)
  The VUIT interface builder supports the $Motif^{TM}$ widget set, and additional user defined widgets. The application interface is shown in a work area, where it can be constructed interactively. VUIT allows the inserting, configuring and layouting of the widgets. The generated interface code (C, Fortran, ADA or Pascal) can call user defined functions, that are stored separately from this code. The interface builder supports the internationalization of the resulting program.
- HP Interface Architect (Hewlett-Packard Company)
  This interface builder is based upon the $Motif^{TM}$ widget set. Widgets can be inserted interactively, modified and positioned in a work area. For various resource types, additional dialogs support the specification of the resource value. The program has an internal C interpreter that reduces the turn around time, and allows it to test the changes immediately.
- NeXT Interface-Builder (NeXT Computer Inc.)
  This system is completely integrated into the NeXT environment (Nextstep). This makes it more powerful than many other interface builders, as the communication between the interface builder, the

application to be built, other applications and the system environment is more sophisticated. The user can interactively create, configure and layout objects in a work area. The objects that are manipulated are instances of the NeXT interface objects. The classes that implement the interface objects are implemented in objective C. These instanciated objects communicate with each other via messages. The interface builder supports the definition of connections between these objects. The automatically generated objective C code contains all definitions and declarations that are needed for the application interface. The user can enrich this code with the functionality that is required by the program.

- Ibuild[21]
  Ibuild was developed by John Vlissides (IBM T.J. Watson Research Center), Steven Tang (Stanford University) and Charles Brauer (Fujitsu Network Transmission Systems, Inc.). The concept of Ibuild is different from that of most of the other application builders. The program is based upon Unidraw, a framework for building direct manipulation editors. The interface that is built by the user is simulated. This allows an abstraction from concrete widget sets (toolkits). The user can develop interfaces independently from widget sets (toolkits) that may not even exist. The application is constructed in a more drawing like way, where the objects are layouted in a WYSIWYG style and the relationships of the objects are specified interactively. As the system is based upon a framework that is designed for drawing and direct manipulation, it is easy to build applications that handle graphical input and direct manipulation. The program generates C++ code for the InterViews and UniDraw toolkits.

- SUIT
  SUIT was developed by Randy Pausch, Matthew Conway and Rober DeLine from the University of Virginia. It is an interface toolkit, which was designed to be easy to use, and to be available at various platforms. An interface built with SUIT is a collection of objects that have a property list (describing the state of the object), a C procedure that examines the state of the object, and a C procedure that handles the user input and changes the object state. To interactively manipulate objects, a property editor gives access to the different properties of the available objects. The application can be modified while the program is running. To avoid two different modes, one for working and the other for editing, the manipulation of objects is done with a (keyboard) modifier, while the normal work with the objects is done without a modifier.

- GINA
  GINA was developed at the GMD Germany. It is an interface builder based upon the $Motif^{TM}$ widget set, and upon Common Lisp. Widgets can be created and manipulated in a work space. The parametrization and layouting of the widgets is supported by special dialogs, and it is possible to add code that implements the application functionality at running time. The program produces a lisp or a C++ code file that implements the interface. The user can add functionality to the program by adding the code to a second (special) file that is also part of the resulting application.

- BYO
  BYO is a *Tcl/Tk* based interface builder that was developed at the Victoria University of Wellington, New Zealand by Andrew Robinson, James Noble, Peter Wood, Roanne Steele, Alexander Leadbeater, Alan Young and Paul Scheffer. The basic concept of this program is to have an interface builder that directly manipulates a running application. To allow this, the interface builder takes advantage of a simple communication feature that is part of the *Tk* system. It is possible to manipulate several applications at the same time. There are various interface builder dialogs supporting the creation, parametrization and layouting of the widgets. The current status of the manipulated application is retrieved and saved to a *Tcl/Tk* file.

2.3 The XF design

Thanks to the great flexibility and the simple handling of *Tcl/Tk*, and a very instructive and conceptionally clean predecessor (named *BYO*), it was easy to implement *XF*. There exists no real design of *XF*. It was inspired by the simple (but very powerful) concept of *BYO*, and a further development that I have done named *XASK*. This program provided a simple language for building a certain type of simple graphical interfaces. These interfaces were connected with traditional (non graphical) applications. As there was no interface builder for this system and the language was very limited, the arrival of *Tcl/Tk* and *BYO* lead to the decision to switch to this more powerful environment.

Three points were most important for the design of *XF*. The programm was supposed to be able to manipulate a running application. This was important because this is the only way to guarantee that the user really gets what he wants, and can test the application without having to compile it or to switch to a simulation mode. This restricted the choice of implementation languages, as the application must be open for modifications while it is running. One way would have been to simulate the running application by writing an interpreter for the implementation language (like C). The other way was to use an interpreted language. The second alternative was chosen. At this point, there were two alternative languages available. The first language was *Tcl/Tk*, a package containing a scripting language and a $Motif^{TM}$-like widget set. The second language was elk, a scheme interpreter with an interface to the X toolkit and widget set based upon the X toolkit (This system is part of the X contributed software). Due to a greater acceptance in the community (and greater acceptance by the author of *XF*), *Tcl* was chosen.

The idea of manipulating an application with an interface builder while it is running was taken from *BYO*. This approach has the advantage that it reduces the amount of data that has to be used to store the contents of the currently developed application. As all changes are directly applied to the program, the program itself contains all information which is necessary to create a *Tcl/Tk* file containing the program definition. *BYO* achieves this by using a *Tk* feature to communicate with the application running in a different interpreter (see figure 2.1). This approach allows the simultaneous manipulation of several applications.

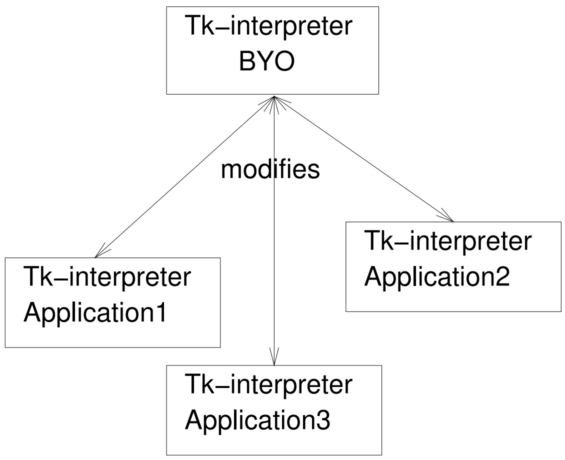


Figure 2.1: BYO design

XF chooses a different approach. It uses one interpreter for both programs (*XF* and the application to be built). The name spaces of both programs (variables, procedures and widgets) are separated by naming convention (see figure 2.2). This was originally done to reduce the communication traffic, and to reduce the complexity of the application. As the approach of *BYO* is more flexible, *XF* will probably be adapted to (also) support this type of manipulating an external application.

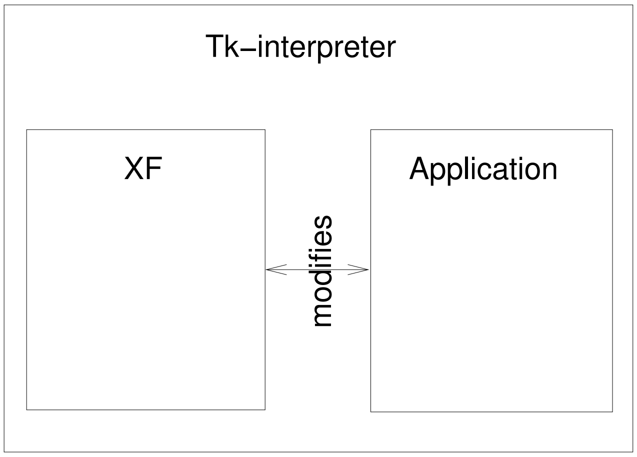


Figure 2.2: XF design

The second very important point was that the developer is supported as much as possible. Newly created widgets are created with reasonable default parameters, to prevent the need of changing every newly created widget. Nevertheless, every aspect of the application can be changed with *XF*. If this is not possible in a certain situation, or the user does not want so much support, it is always possible to change the code directly (by hand in an editor). While many interface builders don't allow the manipulation of the generated code, directly changed code can be reused with *XF* without any restriction.

The third important point for the design of *XF* was to allow the extension of *XF* . There are several well defined internal interfaces, where additional features (like new layouting dialogs, or new widget configuration dialogs) can be added to *XF*. Furthermore, the user is able to adapt most aspects of *XF* via interactive dialogs. This includes the menubar and iconbar layout, the bindings that are used to manipulate the application program and various aspects of the *XF* dialog boxes.

The design of the interface of *XF* itself was also a very important task. Like most interface builders, *XF* displays the available widget classes in some kind of main window. From here, all features of *XF* are activated. The various features are implemented as additional dialogs, that are popped up when they are activated. All configuration and layouting dialogs are nonmodal, so that the user can change parameters and the layout of various widgets simultaneously.

Chapter 3

Tcl/Tk

XF is implemented by using the *Tcl/Tk* package, and the code that is generated is also *Tcl/Tk* code. *Tcl* is a shell-like scripting language. *Tk* is a windowing toolkit offering access to a $Motif^{TM}$-like widget set via *Tcl* commands. Both packages have been developed by John Ousterhout.

The short description of *Tcl/Tk* in this paper is mainly based upon the draft of a book from John Ousterhout [15], slides that John Ousterhout used in his tutorial at the 7th Annual X Technical Conference [18] and other publications of John Ousterhout [17], [16].

When developing applications with *XF*, a basic understanding of some aspects of *Tcl/Tk* is needed. To implement the functionality of the application, the user should be able to write *Tcl* code. Although *XF* provides support for the geometry management of the widgets, the user should also know how the geometry management in *Tk* works.

3.1 Tcl

Tcl stands for a language and a library containing an interpreter. It is a general purpose language designed to be used as an extension language for different applications. By adding the *Tcl* interpreter to an application, the general functionality of a command language like variable handling, control structures, command invocation etc. can be reused. Different applications can use the same language, reducing the implementation expense for the developer, and the learning expense for the user. A *Tcl* interpreter embedded into an application can be easily extended with application specific functions to fit different needs of a specific application.

3.1.1 Syntax

The syntax of *Tcl* is simple. It is a compromise between a shell style language and a lisp like language. The simple structure of commands is typical for a shell language, and is one reason for the acceptance of *Tcl* as an implementation language. A command is formed by words separated by spaces. The first word is the command and the following words are arguments. Commands are separated by newlines or semi-colons.

3.1.2 Datatypes

The only datatype in *Tcl* is string. This makes it easy to exchange data and also allows the use of data as executable *Tcl* code and vice versa. A special type of strings are lists. This is the lisp-like part of *Tcl*. A list is a string formated in a special way. The *Tcl* library contains commands that support the handling of lists.

3.1.3 Variables

Tcl allows the definition of variables with the *set* command. The command gets two parameters. The first parameter is the name of the variable, and the second parameter is the new value of the variable. To refer to the value of a variable, the dereferring symbol "$" is used. Alternatively, the set command called with no new value returns the current value of the variable.

3.1.4 Commands

Commands can return strings as a result. To substitute a command with its return value, the command is included into square brackets. The *Tcl* library contains a rich set of commands that cover most requirements of a programmer. There are commands to handle lists, strings, file I/O, arithmetic expressions etc.. The control structures that *Tcl* provides (like if, while etc.) are also normal *Tcl* commands. They take *Tcl* scripts as arguments. This makes the *Tcl* syntax much simpler.

A programmer can define new commands that are handled like the built in commands by writing *Tcl* code, or by embedding new C function in the *Tcl* interpreter. This is very easy, and the restriction to only one datatype (string) makes the passing of arguments very simple. All core commands and control structures that *Tcl* provides are added as C functions to the *Tcl* interpreter.

3.1.5 Quoting

Quoting of words can be used to suppress the meaning of special characters (like dollar, curly braces etc.). While quoting with "" only suppresses the special meaning of space (as a word separator), curly braces suppress the meaning of all special characters. Quoting a parameter with curly braces delays its interpretation to the execution of the called command.

3.2 Tk

Tk is a $Motif^{TM}$-like widget set that gives access to the widgets via *Tcl* commands. The widget classes use *Tcl* to implement parts of their own functionality, and to implement application specific functionality. This allows it to create complex user interfaces in an interpreted scripting language. The performance of the resulting code is excellent, and the use of the commands is simple.

3.2.1 Widget classes

A *Tk* interface is built by *widgets*. Widgets are grouped into *classes*. The class of a widget defines its appearance on the screen and the functionality of the widget. The *Tk* widget set provides classes like: Button, Label, Frame, Text, Scrollbar etc..

3.2.2 Inserting widgets

When widgets are created, they are inserted into the already existing *widget tree*. The widget tree has a root, named ".". All inserted widgets are children of ".", or a descendant of ".". Usually the higher-level widgets are container widgets (like frames) that define a layout structure for the interface, and the leaves of the widget tree are the widgets that the user uses to interact with the application (like buttons).

An inserted widget has a qualified name, formed by the *widget path* in the widget tree. The name of a widget looks like a (complete) filename in the $UNIX^{TM}$ file system. The "/" from the filesystem is replaced by a ".". The widget name reflects the location of the widget inside the widget tree.

Depending on the widget class, the user calls a command to actually create the widget. The command that is called usually is the name of the class in downcase letters. The widget creation command may contain configuration parameters that set certain widget resources. The syntax of these parameters is described below. To create a button as a direct child of the root, the command looks like this:

```
button .b1 -text Quit -command destroy .
```

.b1 is the widget path of the widget. The following options are used to configure the button at the creation time. This command creates the widget, and creates a new *Tcl* command named *.b1*, which is used to access the widget (i.e. for additional configuration, or other command invocations for this widget). The widget is not displayed when it is created. It is mapped when a geometry manager is used to arrange the widget (see below).

3.2.3 Widget commands

Each widget class has a set of commands (*widget commands*) which are special to that widget class. To invoke a widget command for a specific widget, the *Tcl* command representing the widget is called. This command invocation is followed by the widget command name to be executed and optional parameters. All widget classes have the widget command *configure* which is used to configure the widget. There are several widget class specific commands, like the *flash* command for button widgets. The widget command invocation looks like this:

```
.b1 configure -background blue
.b1 flash
```

3.2.4 Widget configuration

As described above, all widget classes have a widget command named *configure*. This commands gives the user access to the widget resources. Usually, widgets have resources like: *foreground*, *background*, *font* etc.. To set a specific resource, the *Tcl* command representing the widget is called, followed by the configure widget command. Then follows the resource name. Resource names begin with a "-". Behind the resource name, the new value is specified. If no new value is specified, the current value of the resource is returned.

Some widgets (like buttons) have resources that allow it to bind functionality to them. The resources are called *-command*, *-xscrollcommand* etc.. These resources contain *Tcl* scripts which are evaluated when the command is activated (for example when a button is pressed).

```
.b1 configure -foreground red
.b1 configure -command ''puts stdout exit''
```

3.2.5 Geometry handling

Widgets that are inserted into the widget tree, are not displayed. To display the widgets, and to give them a position and size, the *Tk* geometry managers are used. There are two geometry managers, the *placer* and the *packer*.

The placer

The placer allows the direct specification of (absolute/relative) widget coordinates and sizes. It does not really make a geometry handling, and leaves the responsibility for the layout to the user. Here is a small example of a layout, created with the placer. First, the code to place the widgets:

```
place .b1 -relx .5 -y 30 -width 60 -height 40 -anchor c
place .s1 -x 0 -y 50 -relwidth .1 -relheight .7
place .l1 -relx .1 -y 50 -relwidth .9 -relheight .7
```

The resulting layout has a button at the top, named .b with a fixed width (60 pixel) and height (40 pixel), a fixed y position (30 pixel) and a relative x position of 50 percent of the parent's width. So the button has always the specified size, is always 30 pixel away from the top border, and always centered in the middle of the parent. The scrollbar and the listbox use 70 percent of the parents height and (together) the complete width of the parent. They are placed below the button.

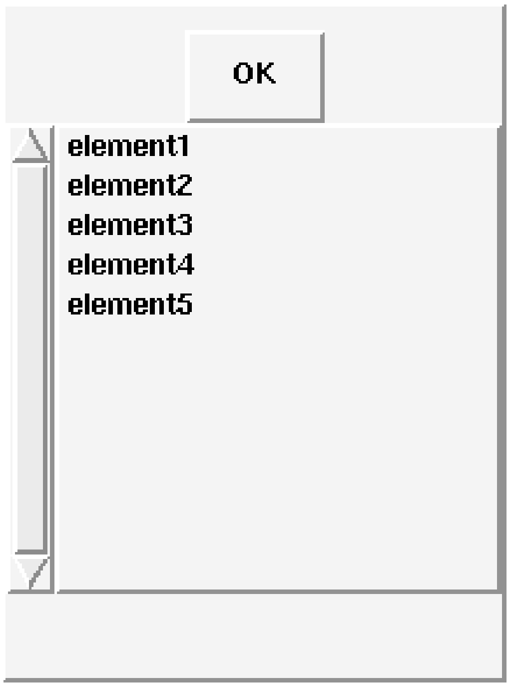
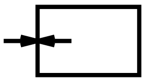


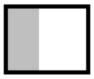
Figure 3.1: Placed widgets

The packer

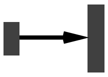
The *packer* is much more powerful than the placer. The children of a widget are automatically arranged around the edges of the parent's cavity. The user can control where, and how the children are packed, but the layout itself is done by the packer. This is done in several steps:



Pick a side of the parent widget (*master*).
The widget (*slave*) that is packed into the master
is packed to this side of the master.



Slice off a frame for slave.
This means, that the packer reserves a area in
the master for the slave. This frame occupies the
complete side of the master.



Possibly grow slave to fill frame.
If the packing options specify it, the slave widget
is resized to fill the frame in one or both axis.

Position slave in frame.
The slave is packed into the master. The position
inside the reserved frame can be specified.

When all this is done, the widget occurs in the parent. Packing and placing can be combined. Here is a small example of a layout, created with the packer. First the code to pack the widgets:

```
pack append .  .b1 {bottom fillx}
pack append .  .s1 {left filly}
pack append .  .l1 {right fill expand}
```

The resulting layout has a button packed to the bottom of the parent filling the complete width of the parent. The scrollbar is packed to the left side and occupies the height of the parent, and the listbox uses the rest of the available space.

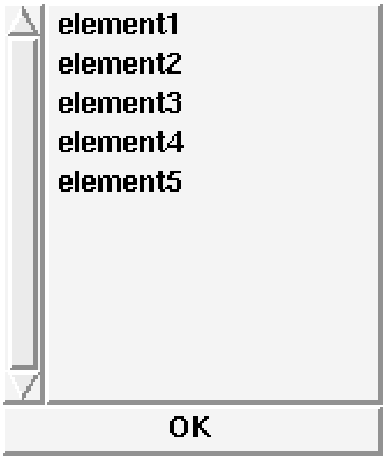


Figure 3.2: Packed widgets

3.2.6 Other Tk commands

Beside the widget related commands, *Tk* offers commands for more general functionalities. This includes commands to define event bindings or to handle the X selection. The commands are explained in the *Tcl/Tk* book by John Ousterhout [15].

3.2.7 XF relevant commands

XF tries to support the development of *Tcl/Tk* programs as well as possible. This does not mean that the user does not have to know anything about *Tk* . To access parts of the interface, the developer must know some *Tk* commands.

The interface allows it to ask the user for information, but the code that requires this information must retrieve it from the interface. So here is a list of the most important commands to retrieve and set information from/to the interface.

Text widgets

For text widgets, the following commands are important. The first command deletes the current contents of the widget. The second command inserts the string “Insert this text” into the widget, and the last command retrieves the current contents of the widget.

```
widgetCommand delete 1.0 end
widgetCommand insert 1.0 Insert this text
widgetCommand get 1.0 end
```

Entry widgets

For entry widgets, the following commands are important. The first command deletes the current contents of the widget. The second command inserts the string “Insert this text” into the widget, and the last command retrieves the current contents of the widget.

```
widgetCommand delete 0 end
widgetCommand insert 0 Insert this text
widgetCommand get
```

Scale widgets

For scale widgets, the following commands are important. The first command sets the scale to 10, and the second command retrieves the current value of the scale.

```
widgetCommand set 10
widgetCommand get
```

Checkbutton widgets

To set and retrieve the current state of a checkbutton, these widgets have a resource named (-*variable*). This variable represents the current state of a checkbutton. It is set to 0 or 1 by default, depending on the current state of the button. It is possible to specify for each button which value is assigned to the variable when each button is pressed. This is done with the resources -*onvalue* and -*offvalue*).

Radiobutton widgets

To set and retrieve the current state of a radiobutton, these widgets have a resource named (-*variable*). This variable represents the current selection of a group of radiobuttons. Radiobuttons are grouped by giving them the same global variable. It is possible to specify which value is assigned to the variable when each button is pressed. This is done with the resource (-*value*).

Send

One of the most powerful features in *Tk* is the ability to send commands directly to other *Tk* applications, using the *send* command. This command takes a *Tcl/Tk* script as a argument, and evaluates it in the specified interpreter. The result of the evaluation is returned to the calling program. This feature makes it easy to spilt functionality used by an application (like color selection, file selection etc.) between different applications, as the communication between these applications, is almost the same as it would be in one complete application (procedure calls). This can lead to a model of applications, where many specialized *Tcl/Tk* application communicate together to provide functionality to other (more general) programs.

Chapter 4

Using XF

XF may be used for prototyping, and for the complete implementation of an application. To implement a complete application, it is necessary that the user is able to write *Tcl/Tk* code. *XF* can free the user a lot of implementation work, but parts of the functionality have to be implemented by the user. So, a basic understanding of *Tcl/Tk* is required.

First, the user should think about the purpose and the look of his application. A rough layout of the application should be drawn, where the dialog structure and the distribution of dialog components to different toplevel (dialog) windows is planned. Of course, it is also possible to use *XF* without a concrete idea of the resulting application. Widgets can be rearranged at any time to see what the resulting interface looks like.

The developer of a graphical user interface must be very careful not to overload the application interface with neat little features which can confuse the user. Many graphical user interfaces tend to be confusing, because the interface tries to offer too much functionality/information, or presents the functionality/information in a wrong way. A graphical user interface is not "good", just because it is graphical. It has to be carefully designed and constructed.

The following step-by-step introduction to *XF* is idealized. The user will probably never really work this way. There are always changes to the widget structure, layout or functionality that occur in the middle of the work, due to enhancements or necessary changes. Nevertheless, this view of designing and implementing applications with *XF* can be a helpful baseline for the work. The dialogs listed in this chapter are only described shortly. There is a special chapter about the dialog components in *XF*, where all dialogs are explained in a more detailed way (see chapter 6).

4.1 The first steps

4.1.1 Start

After starting *XF* , two toplevels appear on the screen. The empty toplevel window is the workspace where the application will be built. The second window is the main *XF* window (see figure 4.1).

The main window is split horizontally into several parts. At the top of the window, there is a section where the functionality of *XF* can be activated. Below this section, three lists represent the available widget classes and templates. The items in the left list represent widget classes that are part of the *Tk* distribution. The middle list represents additional widget classes that are available as extensions to the standard *Tk* widget set. They require a modified *Tk* command interpreter. The right list represents complex widget structures and/or procedures that form some kind of dialog element. These pieces of *Tcl/Tk* source are called "templates". The two buttons at the bottom are used to insert (create) a widget of the currently selected widget class or template into the application.

Typically, a *Tcl/Tk* application has one main dialog window (the *Tk* main window). From here, the other dialog windows and the functionality of the application are activated (displayed). Dialog components that do not need to be in the main dialog window can be placed in additional toplevel windows (like option settings or alert boxes). A toplevel window is almost the same as the main *Tk* window, but they can be destroyed,

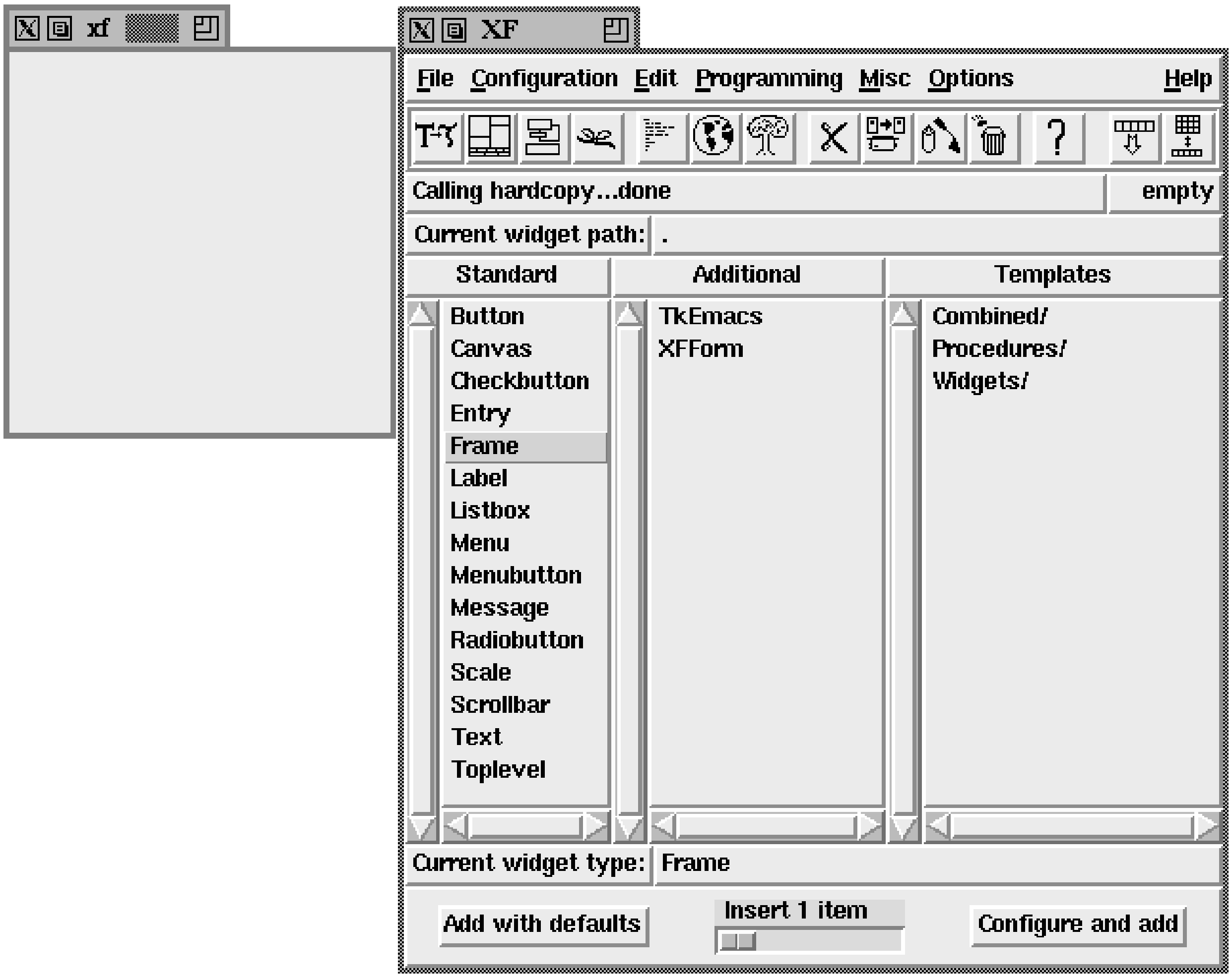


Figure 4.1: The screen after the start of XF

while the main *Tk* window cannot be destroyed. So it makes sense to put dialog elements that have to be displayed permanently into the main *Tk* window, and dialog elements that are used only temporarily into toplevel windows. The toplevel windows and the main *Tk* window contain the widgets that form the interface of the application. Widgets can contain other widgets forming the widget tree. Higher-level widgets are used as containers to layout the leaf widgets that implement the functionality.

4.1.2 Inserting widgets into the widget tree

The building of an application begins with the inserting of the widgets. If there exists a rough idea of the layout, the widgetstructure can already represent the layout of the application. This means that frames are used to structure the widgets.

Widgets are inserted into the application by pressing one of the two buttons in the main window of *XF*. It is also possible to double click on the list item in the main *XF* window representing a widget class. A new widget is inserted into the current widget path(current widget). This path can be set by double clicking the middle mouse button on the widget that is to become the new current widget, or by specifying it with the menubuttons showing the current widget path below the status line. The dots in this pathname contain menus that show all children of the widget on the left of the dot.

The scale at the bottom of the main window can be used to insert a number of widgets at once. Inserting a widget with the left button (`Add with defaults`) produces a widget with default settings and a default widget name. Inserting a widget with the right button (`Configure and add`) pops up a dialog, in which certain parameters and the widget name can be specified. By clicking on the button (`OK`), the widget is inserted.

4.1.3 Layouting widgets

When all widgets are inserted into the widget tree, respectively the widgets that should be inserted at this time, the layouting starts. Typically, the basic layout structure of an application is defined by using *frame* widgets. Widgets that are grouped together are inserted into the same frame.

The widgets are placed in their parents with the geometry managers. *Tk* provides two geometry managers, the *packer* and the *placer*. Which one is used depends on the needed result, and the preference of the user. It is possible, but not recommended, to combine the two layouting methods in the same toplevel window. Both geometry managers ignore the widgets managed with the other geometry manager. So it can happen that widgets accidentally overlap, or the size of a widget is not computed correctly. The combination of both methods should only be used by the experienced user, as it is necessary that both concepts are really understood in their effects. The draft of John Ousterhout's[15] book explains the *Tk* geometry managers in detail.

The user can choose between two ways of layouting with *XF*. He can manipulate the widgets directly (usually when the widgets are placed), or he can use special dialogs where all parameters for the layouting can be set (the usual way for packed widgets). Direct manipulation is only possible, when the layout dialog (`Configuration | Layouting`) is activated. This dialog is intended to prevent unintentional widget layouting during the development and provides minimal access to some layout parameters.

Layouting with the placer

As explained above, there is a placing dialog, where widgets can be selected, placed and rearranged. This dialog is activated with the menu item (`Configuration | Placing`). The description of the placer given in the introduction to *Tk* explains the options that are available in the placing dialog.

When the layouting window is activated (or when layouting is always allowed), the user can manipulate widgets with the left mouse button (he has to press the Modifier1 key together with the mouse button). Selecting the widget in the center allows the moving of that widget. If the widget is grabbed at the border, the widget can be resized, where different points at the border allow sizing in certain directions.

Layouting with the packer

As explained above, there is a packing dialog, where widgets can be selected, packed and rearranged. This dialog is activated with the menu item (`Configuration | Packing`). The description of the packer given in the introduction to *Tk* explains the options that are available in the packing dialog.

It is also possible to directly pack widgets when the layout dialog is activated (or when layouting is always allowed). Selecting a widget by pressing the the left mouse button (he as to be pressed together with the Modifier1 key), moves the widget to the border of the parent that is the closest for the mouse pointer. In the layout window some parameters like fill, expand etc. can be set.

When layouting with the packer, it is important to group widgets with frame widgets. The user can use the frames as row-column widgets. The children of one frame are usually packed to one side of the parent.

4.1.4 Configuring widgets

The next step when building an application with *XF* is to set the widget parameters. Parameters that can be configured for widgets are i.e. the foreground color, or the text font. To change the parameters of a widget, the menu item (`Configuration | Parameters`) can be selected. It is also possible to double click with the right mouse button at the widget to be configured. This activates the parameters setting dialog for the currently selected widget, or the double clicked widget.

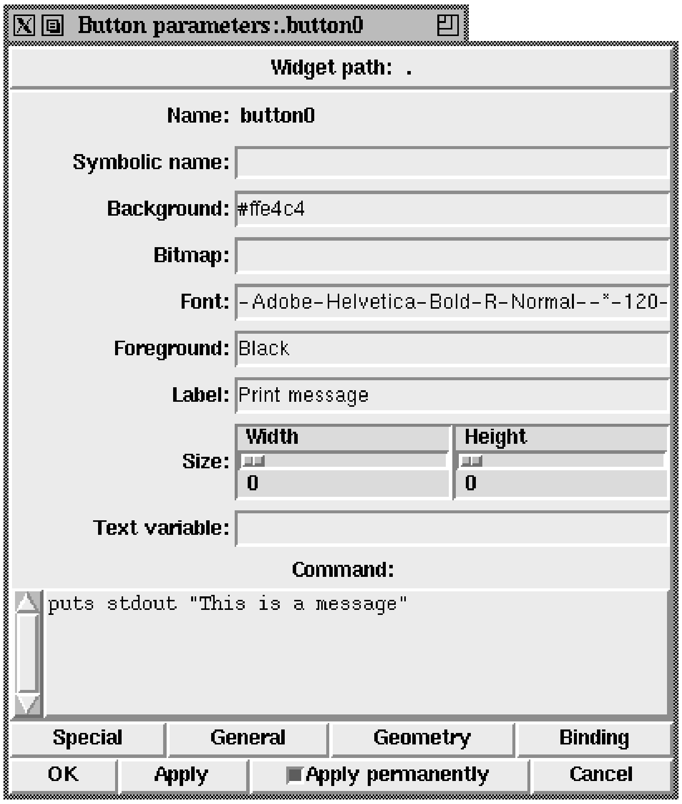


Figure 4.2: A standard widget parameter box

The parameter dialog box (an example is shown in figure 4.2) allows the setting of different parameters (resources) for the widgets. For some of these resources special dialogs allow an interactive selection of possible values, by double clicking the right mouse button in the value field inside the parameter setting dialog. A very important parameter is the *command* parameter. This parameter allows it to bind functionality (*Tcl* commands) to a widget (i.e. a button, or a menu item).

Each widget class has its own parameter dialog. For some widgets (like canvas or menu) there exist special dialog boxes that provide support for sophisticated features of the widget class (like drawing graphical objects in a canvas widget, or building a menu). For all parameters that are not covered with these dialog boxes, a general parameter dialog provides access to all resources that can be modified for a widget (see figure 4.3).

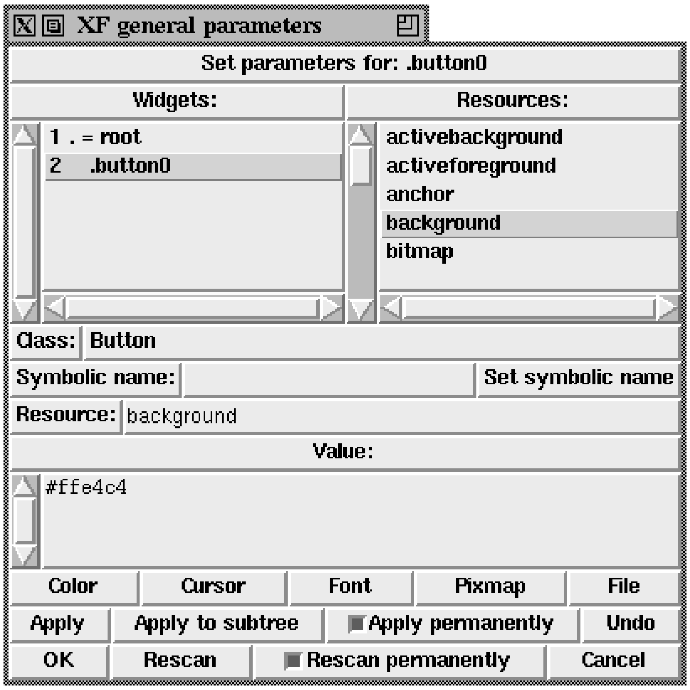


Figure 4.3: The general widget parameter box

Very often, certain widget parameters are identical for a great number of widgets. In this case, the parameter setting for widget groups can be used (see figure 4.4). This dialog allows it to select groups of widgets, and to set parameters for all these widgets at once. The basic handling is like for the general parameter dialog.

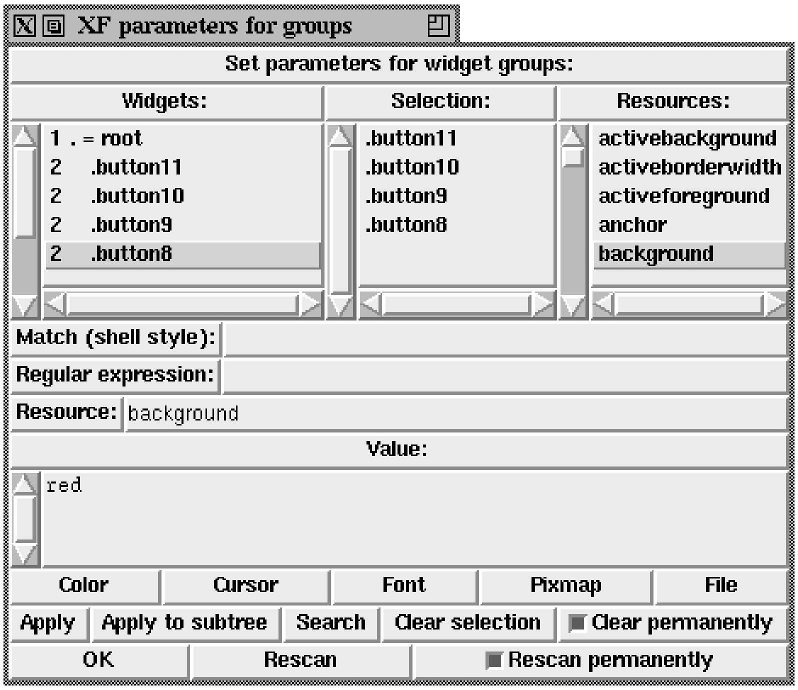


Figure 4.4: The widget parameter box for groups

4.1.5 Procedures

When the interface for the application is finished, the functionality must be implemented. The functionality is usually implemented with *Tcl/Tk* procedures that are called by buttons, bindings etc.. Here the user's *Tcl/Tk* experience is needed (respectively: here he gains *Tcl/Tk* experience).

The implementation of the functionality is supported by *XF* with special dialogs. These dialogs are activated with the menu items (`Programming | Procedures`) and (`Programming | Commands`). They activate dialog windows, where procedures and commands can be created, modified, deleted etc. (see figure 4.5).

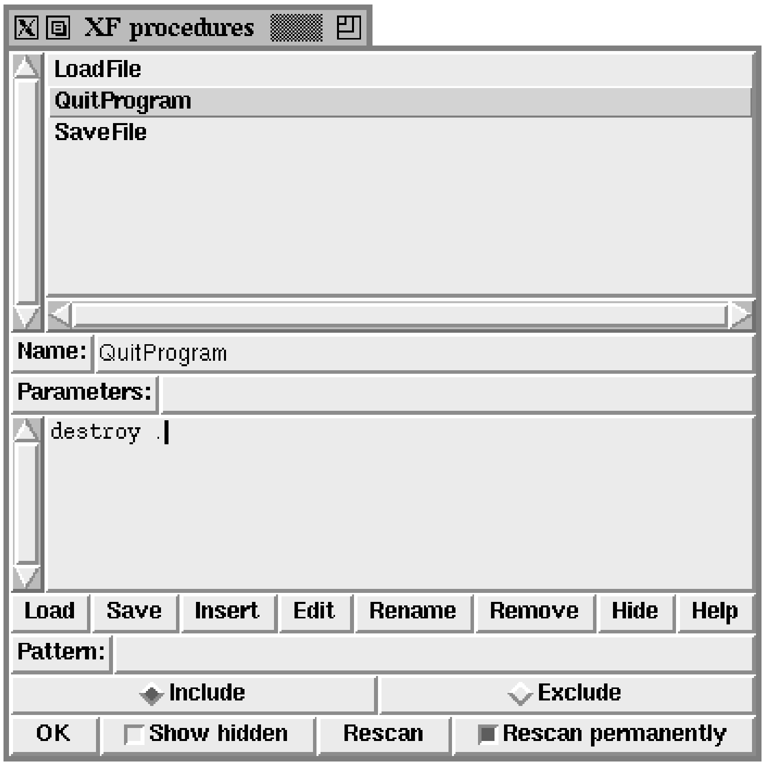


Figure 4.5: The procedure dialog

Besides the procedures that are written by the user, there exist various special procedures, that are used by *XF*. The procedure dialog only gives access to procedures that the user is allowed to change. The two most important procedures are *StartupSrc* and *EndSrc*. To modify these procedures, the menu items (`Programming | Startup source`) and (`Programming | End source`) are used.

If the application needs control when the application program is started (i.e. for parsing the argument list or for setting the contents of a list at startup), the code is added to the startup procedure. This code is executed as first source of the complete application.

If the application needs control after the toplevel windows (containing the dialog components) have been displayed (i.e. to initialize widgets with certain values), the code is added to the end procedures.

4.2 Advanced features

4.2.1 Templates

XF provides the concept of *templates*. Templates are files which contain *Tcl/Tk* code defining a widget structure and/or procedures (stored as *Tcl/Tk* code). A template can be loaded by the user, adding this widget structure and/or functionality to the application program. The inserted template behaves the same way as any other code written by the user. The user can change it, configure the inserted widgets etc..

The available templates are shown in the right list of the main *XF* window. By double clicking on a list item, the selected template is inserted. If the contents of a template have changed (i.e. when a new *XF* distribution is published), the template can be reloaded by inserting it again. This is only necessary when the old template was buggy, or the new template provides new functionality.

XF comes with a set of templates. There exist three main groups of templates named: *Combined*, *Procedures* and *Widgets*. The user can save widget structures and procedures as new templates. If a widget structure is temporarily stored to the cut buffer (with the cut/copy functionality), this widget structure can be saved with the menu item (`Edit | Save Template (cut buffer)`). It is also possible to interactively select procedures and a widget path to be stored to a template in the module structure dialog (`Misc | Modules`).

To reduce the size of the application program, it is possible to use a special type of templates. They are called autoloadable templates. This type of templates resides in a directory (./autoProcedures), where a tclIndex file can be found as well. All procedures defined in the templates are listed in the tclIndex file, and can be loaded automatically (using the *Tcl* auto load feature). The procedures are not saved when the program is saved. Instead the user has to add the pathname where the templates reside to the environment variable *XF_LOAD_PATH*. This variable contains a list of directory names separated by ":", where *XF* can find modules that are part of the application program.

4.2.2 Toplevel windows

Toplevel windows (including the main *Tk* window ".") contain the widgets that form the interface. The main *Tk* window "." is the root of the widget tree. An application usually contains various dialog components implementing different aspects of the program. They have to be displayed depending on the current status of the program. This makes it important to be able to hide/show toplevel windows.

The main *Tk* window "." can be hidden with the command: "*wm withdraw .*". To display the window, the command: "*wm deiconify .*" is called. This also works for the other toplevel windows, but *XF* provides an additional way to show/hide the toplevel windows. Each window can be displayed with the automatically created procedure ShowWindow.<toplevelName>. To remove a window, the procedure DestroyWindow.<toplevelName> is called.

When the code for the application is saved, the current display status of the toplevel windows is saved. This means that when the application is started, the toplevel windows that were displayed when the program was saved are displayed, and the toplevel windows that where hidden when the program was saved are not displayed. To change the display status of a window when developing with *XF*, the menu attached to the label (`Current widget path:`) in the main *XF* window can be used.

4.2.3 Source modules

An *XF* generated program can be packed into one file, containing the complete code. This is usually done for code that will be distributed. It makes the calling easier, and reduces the number of files to be installed. To create such a file, the menu item (`File | Save as...`) is activated.

During the development, it is usually better to have small modules that can be handled independently. For example it is a good idea to put the external (*XF* created) code into a separate module. The code for the widget creation should be put into a separate module, preferrably one module for each toplevel. Finally the procedures loaded from templates should also be stored into modules named after the templates. This makes it easier to locate and modify different pieces of code when working with an editor.

Another aspect of modularization is the working in groups. If a program is built by a group of developers, each developer can put his code into one or more separate modules. These modules are stored locally by each developer, giving him full control over the modules. Modules handled by the other developers can be retrieved by *XF* at loading time. To access the distributed modules, it is necessary to set the environment variable *XF_LOAD_PATH*. This variable contains a list of directory names separated by ":". Here, *XF* tries to find modules that are part of the application. *XF* looks for plain files, and for archives using the ShapeTools.

By specifying which modules should be saved (with the module structure dialog), only the modules that are accessed by the developer are updated. Each developer can publish a module when it is ready, by moving it to a public directory, or by making a check-in into the global archive using ShapeTools. The global program structure should be maintained by one specific administrator, who has control over the main module.

To distribute the toplevel windows and the procedures into different modules, there exists a special dialog box. This dialog allows the user to split the contents of his application (toplevels and procedures) into readable pieces (see figure 4.6).

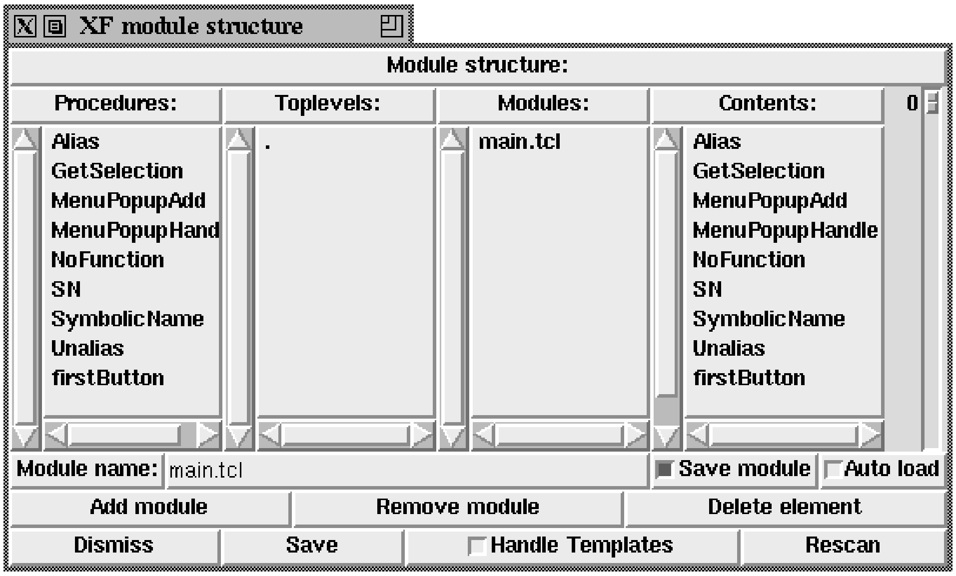


Figure 4.6: The module structure dialog

New modules are added by typing the new module name, and pressing the button (`Insert module`). To add an element to a module, the user clicks on the appropriate item in the left list. To remove elements from a module, the user double clicks on the item in the right list. There is always one module that has the name of the application. It contains all toplevels and procedures that were not placed in another module.

A module can be made an auto load module with the checkbutton labelled (`Auto load`), right beside the module name. This means the needed code is generated that loads the module when a procedure from that module is called.

To support the development in groups, it is possible to specify which modules should actually be saved. This is done with the checkbutton labelled (`Save module`), right beside the module name. This information can be saved to a user specific file.

By selecting the button (`Handle Templates`) this dialog is switched to the template mode. Here the user can select exactly one widget path and an unlimited number of procedures to be saved to a template. To save this template the (`Save`) button is pressed. This save button can also be used to save the complete program (when the template mode is not activated).

4.2.4 Levels for procedures and bindings

To make the handling of bindings and procedures easy to survey, they can be assigned to a level. This means that the first line of the *Tcl/Tk* command that is bound to an event or a procedure name begins with the comment "# xf ignore me <level>". Such a binding or procedure is handled differently than other bindings and procedures. They can be hidden in the *XF* dialogs, and they can be ignored when the source is saved.

The user can assign 9 levels to bindings and procedures. Each level can be turned on and off separately (both for displaying and saving). Some levels are used by *XF* already, but the remaining levels can be used to categorize procedures and event bindings. For event bindings, only the level 9 is used by *XF*. This level means that the bindings are totally ignored. The following table shows the usage of the levels for procedures.

Level	Purpose
1	Not used by *XF*.
2	Not used by *XF*.
3	Not used by *XF*.
4	Procedures that are used to implement the *XF* alias feature get this level. They should be saved, but it is not necessary to display them.
5	The main template procedures (those which are called by the user) have this level.
6	The supporting template procedures (those which are not called by the user) have this level. They should not be displayed.
7	Procedures that are used by the *XF* generated code get this level. They should be saved, but it is not necessary to display them.
8	*Tk* procedures that are to be saved get this level. Right now no procedure has this level.
9	Procedures that have this level are totally ignored by *XF*.

Chapter 5

The Implementation of XF

The startup of *XF* is shown in figure 5.1. First, the *XF* configuration file is loaded. Afterwards, the application default file is loaded, and parsed. Now, the application source is loaded. After that, the local startup file is loaded, where the user can add functionality to *XF* . Then the main *XF* window is initialized, and the application defaults are applied to the widgets. Finally, *XF* gives control to the *Tk* main loop. Inside this loop, the X events are processed (i.e. to activate the various *XF* dialogs).

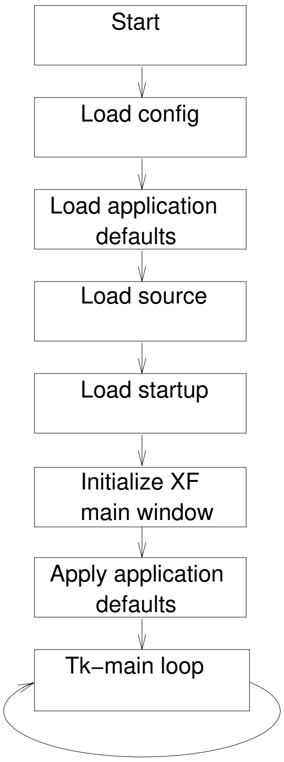


Figure 5.1: XF startup

Procedures that are needed by *XF* are loaded with the auto load feature of *Tcl*. This means that they are loaded on demand.

5.1 The generated code

The source that is created by *XF* has a general layout, and contains a certain set of functionality. The produced code can be saved into one file, or be splitted into several modules. In both cases the general layout and functionality is the same.

Apart from the procedures that are written by the user, *XF* saves a number of procedures that are used to create the widget structures, initialize the program or support the special *XF* bindings. The creation of most of this code can be turned off and on explicitly (`Options | Source options`).

1. Module inclusion code.
   If the source has been split into modules, the main file of the application begins with code that initializes the module load path and parses the commandline options.

2. Toplevel creation code.
   The procedures "ShowWindow...", "DestroyWindow..." etc. are created automatically by *XF* , to display and hide the various toplevel windows of the application. There can also be procedures named "StartupSrc...", "MiddleSrc..." and "EndSrc...". They are called by the corresponding window creation procedure, and can be used to initialize the widgets that are inside the displayed toplevel window.

3. User defined procedures.
   If the source is structured into modules, the procedures that have been written by the developer are all stored in this section. The order can be changed, and the procedures can be distributed to different modules.

4. Internal procedures.
   This section contains procedures that *XF* creates to produce a running application. The creation of most of these procedures can be turned off and on explicitly (`Options | Source options`). The remaining *XF* internal procedures are all assigned to the level 7 (see chapter "Using XF"). If the saving of level 7 is disabled, these procedures are not saved. If the source is not structured into modules, the user defined and the internal procedures are mixed, as *XF* stores all procedures in alphabetical order.

   (a) XFLocalIncludeModule
   This procedure is called with the name of the module to be loaded. The procedure scanns through the directories specified with the environment variable XF_LAOD_PATH, and tries to load this module. In addition to that archives are searched for the modules. The user can use the environment variable XF_VERSION_SHOW to specify the retrieve command (see the description of version control).

   (b) XFLocalParseAppDefs, XFLocalLoadAppDefs and XFLocalSetAppDefs
   These procedures provide full support for the X Resource mechanism. The procedure XFLocal-LoadAppDefs gets an application class name, and loads the resource stored under this name. The procedure XFLocalSetAppDefs applies the resources to the specified widget path.

   (c) SymbolicName and SN
   These two procedures implement the symbolic name handling. The developer can assign a name to a widget path name. This makes the access to this widget easier. Calling these procedures with the symbolic name, returns the concrete widget path name.

   (d) Alias and Unalias
   These two procedures implement the alias handling. Aliases can be used to access procedures and widgets under a different name. The difference from symbolic names is that each aliased procedure creates an additional procedure, making this approach a little space consuming. The alias procedure gets two parameters, first the new alias name, and second the procedure that is aliased. Unalias gets one parameter, specifiying the alias name to be removed.

   (e) GetSelection
   This procedure is used to implement a safe X selection retrieval for the text and entry widget. If the bindings do not contain this procedure, it may be deleted.

(f) MenuPopupAdd and MenuPopupHandle
The procedure "MenuPopupAdd" sets the appropriate bindings to pop up a menu. The popup menu is attatched to the widget specified with the first parameter. The second parameter specifies the mouse button that activates the popup menu (1, 2 or 3). The next paramter specifies the widget name of the menu that is to be popped up. The following parameter is optional, and can contain a valid event modifier. The last parameter can contain a canvas tag or id. The menu is attatched to this tag or id.

(g) NoFunction
This procedure does nothing. It can be called with any number of parameters.

5. StartupSrc and EndSrc.
These two procedures contain source that is executed at the startup of the application program. "StartupSrc" is executed as first code in the application, and "EndSrc" is executed after all toplevel windows have been displayed. The user must add initialization code to one of these procedures.

6. Invocation of StartupSrc.
The startup code is evaluated.

7. Initialization of variables.
The global variables are initialized with the procedure "InitGlobals", or directly inside the code. The developer should initialize important variables directly in his initialization code for his application. The values assigned in the "InitGlobals" procedure represent the values when the application was saved.

8. Showing the toplevel windows.
The toplevel windows that should be displayed are created by invoking the appropriate "ShowWindow" procedure.

9. Loading of bindings.
The user can specify a *Tcl/Tk* file containing bindings for his application. This file can be specified with the environment variable XF_BIND_FILE, or the commandline option "-xfbindfile" when the source contains the commandline parsing code.

10. Application default handling.
The application default file is loaded, and the settings are applied to the displayed widgets.

11. Invocation of EndSrc.
The end code is evaluated.

5.2 XF constraints

5.2.1 Restrictions

There are some restrictions when using *XF*. Code that is written with *XF*, or should be imported to *XF* should consider the following points:

- No variable names starting with xf,
- no window names starting with .xf,
- no procedure names starting with XF,
- no procedure names starting with ShowWindow,
- no procedure names starting with DestroyWindow,
- no procedure names starting with StartupSrc,
- no procedure names starting with MiddleSrc,

- no procedure names starting with EndSrc,
- toplevels may only be children of ".".

These restrictions should not affect the normal working with *Tcl/Tk*. They are needed for design or performance reasons.

5.2.2 Pitfalls

The following list contains interesting hints, that are not really restrictions, but have to be handled carefully.

- Using {} and "".
  Tcl commands bound to events or resources can be surrounded with {} or "". The use of {} is much safer, than the use of "". The code surrounded by "" is parsed and substitutions are performed, when the binding is added or the resource is set. This may lead to unwanted results. It is better to use global variables to pass parameters into commands bound to events or resources.
- Setting auto_path.
  If the loaded source sets the auto_path variable, it should check if it is started inside of *XF* . Manipulating the auto_path variable can make *XF* inoperable.

5.3 Extending XF

5.3.1 Supporting new widgets

The several widgets are supported via widget class specific files, that are located in the "additionals" directory. To add support for a new widget, the user has to write such a file. The name of the file is the class name of the widget class to be supported. The user should take a look at the files that have been defined already (i.e. elements/Button) to get an impression of this kind of support files.

In general, it is up to the user what he does in this file. There have to be some procedures that follow certain naming conventions. Besides, the user can implement the support as he wants to. Of course, the existing support files use many functions provided by *XF* to make the writing of the widget support files easier.

XFAdd.<WidgetClass>

Each support file must contain a procedure named "XFAdd.<WidgetClass>". All exported procedures in the support file end with the widget class name. This procedure inserts a widget of that class to the application. It gets three paramters. The first parameter can be ignored (set to ""). The second paramter is an optional widget name. This is not the complete widget path, only the preferred name of the widget at the insertation level. The last parameter is set to "add" or "config". "Add" means that the widget is inserted with default parameters, and "config" means that the widget will be configured by the user.

The procedure has to guarantee that the widget name is unique. To create a unique widget name, the procedure "XFMiscGetUniqueName" can be used. After inserting the widget into the current widget path (xfStatus(path)), the new widget must be placed, using the procedure "XFMiscPositionWidget". This procedure gets the complete widget path name as parameter. The procedure "XFMiscBindWidgetTree" also gets the widget path name, as this is required to set the *XF* internal bindings for that widget. Finally the procedure "XFEditSetPath" should be called, to update the *XF* internal lists. This procedure gets the current widget path as parameter.

XFAddTmp.<WidgetClass>

The optional procedure XFAddTmp.<WidgetClass> is used to create a temporary widget of that class. This can be useful when the widget creation command is not the same as the widget class name.

XFConfig.<WidgetClass>

The procedures "XFConfig.<WidgetClass>0-5" call the various parameter setting dialogs. The number indicates the type of the dialog. It is possible to specify higher numbers than 5. If a procedure representing a dialog between 0 and 3 does not exist, the default *XF* dialog is used.

Nr.	Purpose
0	Activates the packing dialog for this widget.
1	Activates the placing dialog for this widget.
2	Activates the default geometry handling dialog for this widget.
3	Activates the binding dialog for this widget.
4	Activates the default parameter dialog for this widget. This dialog should cover the most important resources of the widget.
5	Activates the special parameter dialog for this widget. This dialog is used to implement special features, like drawing for canvas widgets.

The procedures that implement the parameter setting dialog can use various functions that *XF* provides to make the writing of widget dialog easier. The procedures "XFTmpltToplevel" and "XFElementInit" create a standard parameter setting formular. The procedures get several parameters that are explained in the source code. For various resource types, there are procedures that create appropriate dialog elements (i.e. "XFElementColor").

To enable the undo feature, the procedure "XFElementSave" is called, getting the widget name of the configured widget, the class of that widget, and a list of resource names.

XFSaveWidget.<WidgetClass>

To allow a special handling for the saving of the widgets, the procedure "XFSaveWidget.<WidgetClass>" can be defined. This procedure gets a file descriptor, and the name of the widget to be saveed. This procedure has the responsibility to completely write the code that creates the widget.

XFSaveSpecial.<WidgetClass>

To just save additional code for the widget after the widget creation code has actually been saved, the procedure "XFSaveSpecial.<WidgetClass>" can be defined. This procedure gets the widget path name, and returns a string that should be saved, containing addtional widget creation code.

5.3.2 The procedure XFExternalInitProc

Procedures that begin with this name are evaluated when *XF* is started. This allows it to run initialization code that should only be executed when *XF* is running.

5.3.3 XF startup file

XF allows the specification of a startup file via the commandline option -xfstartup. The default name for this file is ".xf_init". This file is evaluated when *XF* is started. Here addiditional code can be merged adding new functionality to *XF*.

5.3.4 Adding procedures named XFProc

The best integration of new functionality into *XF* is done when the user writes a procedure the name of which begins with an "XFProc". By adding this procedure to the tclIndex file, the procedure can now be called from the menubar or the iconbar.

Chapter 6

Conclusion

6.1 Epilogue

A short look on *Tcl/Tk* is worth the effort, no matter if *XF* will be used or not. The boost in productivity is enormous. The first version of *XF* (which was my first *Tcl/Tk* program) was written in about 2-3 months. This version already contained almost all the basic functionality that is still part of *XF*.

Users reported that developing with *XF* tore down the wall between them and the *X window system*^{TM}. It enabled them to develop complex user interfaces in a few days, without knowing anything about the internals of the *X window system*^{TM}. Of course, a major reason for this is the easy handling of *Tcl/Tk*, but *XF* makes it even easier to work with *Tcl/Tk*. It is possible to switch between hand coding and developing with the user interface builder at any time. Experienced users who implement major parts of their applications by hand can still use *XF* for certain tasks.

There are a few restrictions when developing with *XF*. They should not affect the development. It's just that certain names should not be used for variables, procedures and widgets. And there are some conventions where directly inserted code has to be placed. Besides, it should be possible to write any application that can be written directly with *Tcl/Tk* under *XF*. The user can always switch between directly changing the application, and using *XF* to modify the application.

There is a growing number of *XF* users, and there are already commercial users of *Tcl/Tk* and *XF*. The reports of these users indicate that the concept and the implementation of *XF* are a good working base for the design and implementation of graphical interfaces, although there are some problems. The most important problem is that, as *XF* and the application that is built are sharing the same interpreter, changes to the application causing errors may also cause problems with *XF* itself. The cause for this sensitivity is just the same flexibility that allows the implementation of *XF*. Without this flexibility *XF* would not be possible.

The current restriction of *Tcl/Tk* to the $UNIX^{TM}$ platform may be overcome in the future. In this case, *XF* would be available on alternative platforms (e.g. *Windows* running under $MS - DOS^{TM}$[1]).

6.2 Missing features

There are some missing features in *XF*. Future releases will hopefully contain:

- undo/redo.
  Allowing an undo/redo in an program like *XF* is almost a "must". This is hard to do, as it can make the program slower, and the basic design does not support this.

- configuration via *send*.
  Right now, the application that should be modified with *XF* must be loaded together with *XF*. This was a design decision. It will be changed to allow the parallel manipulation of many applications with

[1] DOS is a registered trademark of Microsoft Corporation

one *XF*. This gets more and more important with the increasing use of *send* to connect different *Tcl/Tk* applications.

- support for specific interface styles.
  XF makes no assumptions (and provides no special support) for certain styles of interfaces. One interface style that should definitely be supported more is the design of forms. This will include an automatic layout, a validation check for entered data and an automatic connection between fields and forms.

- drag&drop.
  The ability to drag&drop widgets and groups of widgets will be added in a future release. This will be used to rearrange the widget layout, and to define connections between widgets (i.e. connecting a scrollbar to a listbox). The problem is the drag&drop between different applications/toplevels, and the general visualization of this drag&drop. Right now, the *Tcl/Tk* distribution does not provide support for this, like a general drag&drop protocol.

- default parameter settings.
  There should be a way to define default parameter settings for widgets. Changes to a default parameter setting should be automatically applied to all widgets that where configured with this default parameter setting,

- support for debugging.
  Right now debugging is not supported. There exists a small extension to *Tcl* that allows the debugging of *Tcl* programs. It should be examined if this feature can be used and embedded into *XF*.

- support for other languages & widget sets.
  XF was originally designed to support the development of *Tcl/Tk* applications. Of course, this will always be the major aim of *XF*, as *Tcl/Tk* is used to implement *XF*. But it should be possible to create C code for parts of the application code. It should also be possible to use *XF* as a sort of abstract interface builder that is able to create code for other widget sets than *Tk* (i.e. $Motif^{TM}$).

XF is still in development, and new features are permanently added. If a user has specific wishes or finds bugs, he can contact me to get this into the next release. With this paper, the internal interface is documented, and I hope that users will contribute new functionality.

Comments are welcome....

Appendix A

External Tools

Whenever it is possible and suitable, *XF* uses external tools to implement functionality. This means that certain tasks in the development are performed with an application especially designed for this purpose. This reduces the size and complexity of *XF*, and gives the user the chance to use commonly used tools inside of *XF*.

XF uses the following external programs:

- edge,
- emacs,
- shape,
- tkemacs,
- vi,
- xfappdef,
- xfhardcopy,
- xfhelp,
- xfpixmap,
- xftutorial.

A.1 Edge

Edge[14] is a program that allows the layouting and displaying of graphs. *XF* uses this program as an additional feature to represent the widget structure as a tree. It is not possible to use the displayed widget tree interactively. If this feature is not used (wanted), edge is not needed.

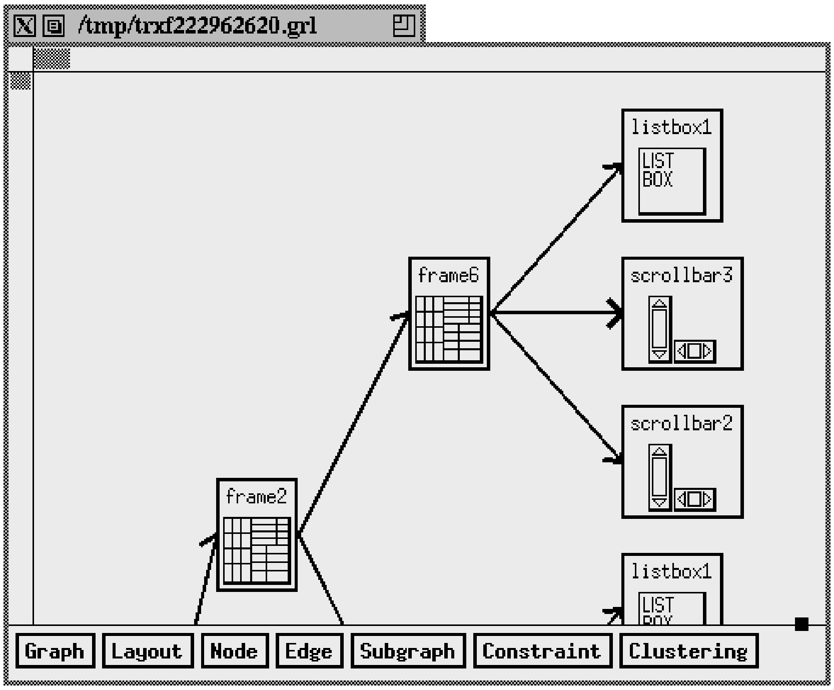


Figure A.1: The edge program

A.2 Editors

When editing sources every user has his own preferences which editor to use. *XF* does not force the user to use one specific editor. To select the editor, chose the menu item (`Options | General options`). The entry `Editor` allows you to specify a shell command that activates the editor. If this entry is empty the *Tk* internal text widget is used. When calling the editor *XF* must pass a file name to the editor. The editor command must contain the string "$xfFileName" at the position where the filename is to be replaced.

To select the (optional) tkEmacs widget as editor, the menu item (`Options | Interpreter options`) is chosen. After selecting the checkbutton (`Interpreter has the tkEmacs widget`), the emacs widget described below is used.

A.2.1 TkEmacs

If the tkEmacs widget is used as editor by selecting the checkbutton (`Interpreter has the tkEmacs widget`) in the (`Options | Interpreter Options`) dialog, the user gets complete access to emacs. The emacs is displayed in a *Tk* window, which means the user can work with emacs without leaving *XF*. The widget is only used in the procedure editing dialogs (`Programming | Procedures`) and (`Programming | Commands`). The reason is that each occurrence of this widget creates a separate emacs. This could lead into system overload.

A.2.2 Emacs

If emacs is specified as the editor by setting the entry `Editor` in the (`Options | General`) dialog to "emacs $xfFileName", an external emacs is started in a separate window. This external editor is only used in the procedure editing dialogs (`Programming | Procedures`) and (`Programming | Commands`), to prevent system overload. The entered code is sent to *XF* by terminating the editor. If the code is not correct (*Tcl* syntax), the editor is automatically restarted. If the user cannot find the error, the code should be saved to an external file. Then, the buffer can be cleared, and emacs can be terminated. An empty file is assumed to be correct.

A.2.3 vi

To use vi as editor under *XF*, the string "xterm -e vi $xfFileName" is entered as editor command. Beside the different command string, everything is the same as when emacs is used as external editor.

A.3 ShapeTools

ShapeTools is a toolkit for software configuration management [9], [8], [12], [13], [11]. *XF* uses parts of this package for version control. If the *ShapeTools* are not installed, the functionality of *XF* is restricted in some way, but the program is still working.

A.3.1 Saving procedures

XF allows saving selected procedures into a shape archive. This makes the procedures publically available, and allows the access to the development history of this procedure. This feature is accessed from the *Programming* dialog window. The buttons (`Save`) and (`Load`) are used to save and load the procedures. Saved procedures are stored with version numbers, showing the development of this procedure. If ShapeTools are not installed, the procedures are saved as plain files, without any additional information. In this case only one version of the procedure exists.

A.3.2 Retrieving modules

When *XF* is used in a developer group, writing on different parts of the same program, shared access and version control get important. *XF* tries to support the distributed development process.

The *ShapeTools* concept of distributed development is based upon the sharing of one global source archive. All pieces of code that are intended to be publically available are checked-in into this database.

XF supports this public access. When the application is started and loads a code module, *XF* tries to locate the module by scanning through the path names listed in the environment variable *XF_LOAD_PATH*. This variable contains a list of path names separated by ":" where modules for this application can be found. If the code module is not found, *XF* retrieves the last checked-in version of the module from the *ShapeTools* archive.

A.4 xfappdef

XFappdef allows to interactively manipulate an X Resource file. It is possible to add and remove resource name specifiers. There are menus, where all known program classes are listed, and menus showing a set of commonly used resource names. For certain resources, there exist special dialogs, where the user can interactively select/specify the value for a resource. The program has dialogs for Font, Color, Cursor File and Pixmap selection. This program is part of the *XF* distribution and is used to manipulate the *XF* resources, and the resources for the program that is build with *XF*.

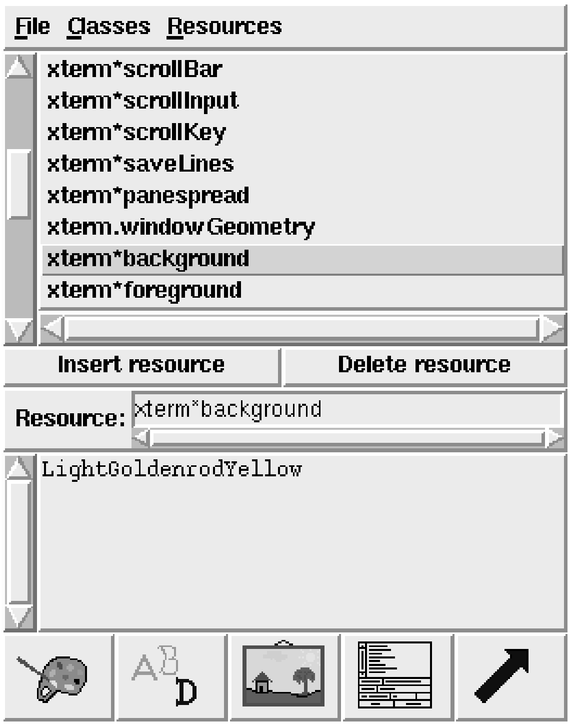


Figure A.2: The xfappdef program

The upper list contains all resource specifications that are given in the application resource file. The two buttons below this list control the insertion and deletion of resource specifiers. The entry contains the currently selected resource specifier, or the name of a resource specifier that should be inserted or deleted. The text field contains the value of the currently selected resource specifier, or the new value to be inserted. The buttons at the bottom activate special dialogs that support the setting of certain resource types.

To load, merge and save the resource file, the menubutton (`File`) is used. The menubutton (`Classes`) provides access to all application class names that are known. By selecting the menubutton (`Resources`), a selection of commonly used resource names can be accessed. Selecting an application class name or a resource name automatically inserts this value into the entry field that shows the resource name specifier.

A.5 xfhardcopy

XFhardcopy allows to interactively select *Tk* applications and path names inside these applications, for dumping to a file. The program uses different external programs to implement the hardcopy commands (xgrabsc 2.1, xwd), and it provides access to the *Tk* postscript command for canvas widgets (this command is new in *Tk* 3.0). If the hardcopy commands that come with the distribution do not work or if there exist more appropriate programs, the hardcopy commands can be adapted by the user. This program is part of the *XF* distribution, and is used to make hardcopies from the program that is built with *XF*.

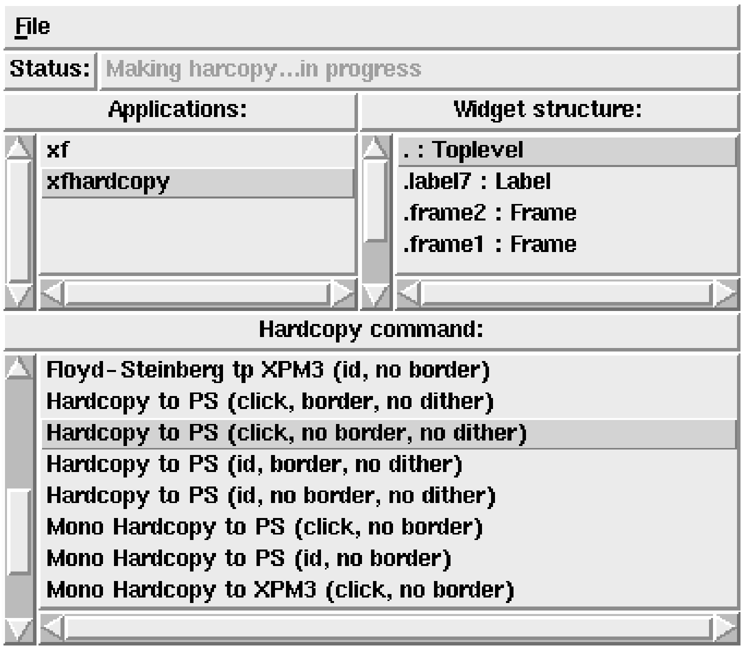


Figure A.3: The xfhardcopy program

The Tk application which should be hardcopied can be selected from the upper left list, and a widget path inside of this application from the upper right list. To make a hardcopy of the selected widget tree (application), one of the hardcopy commands from the bottom list is selected with a double click. By default the hardcopy is written into the current directory under the name xfHardCopy. To specify a different output file name, the hardcopy is made with the menu item named (`File | Hardcopy to...`).

A.5.1 Hardcopy commands

To modify the currently selected hardcopy command the menu item (`File | Modify hardcopy command`) is activated. The following parameters can occur in the hardcopy commands and are substituted when the hardcopy command is activated:

Variable name	Contents
height	The height of the selected widget
id	The X window id of the selected widget
outputFile	The name of the specified output file
rootx	The absolute x position of the widget
rooty	The absolute y position of the widget
widget	The *Tk* widget name of the selected widget
width	The width of the selected widget
x	The relative x position of the widget
y	The relative y position of the widget

A.6 xfhelp

XFhelp is a general help program based upon *Tcl/Tk* . Different programs can use this program to provide help for themselves. The user can browse through the help pages, specify book marks, append notes etc.. Once xfhelp is started, all additional calls of xfhelp use the existing xfhelp, and just change the currently displayed contents. This program is part of the *XF* distribution, and is used to provide help for the user.

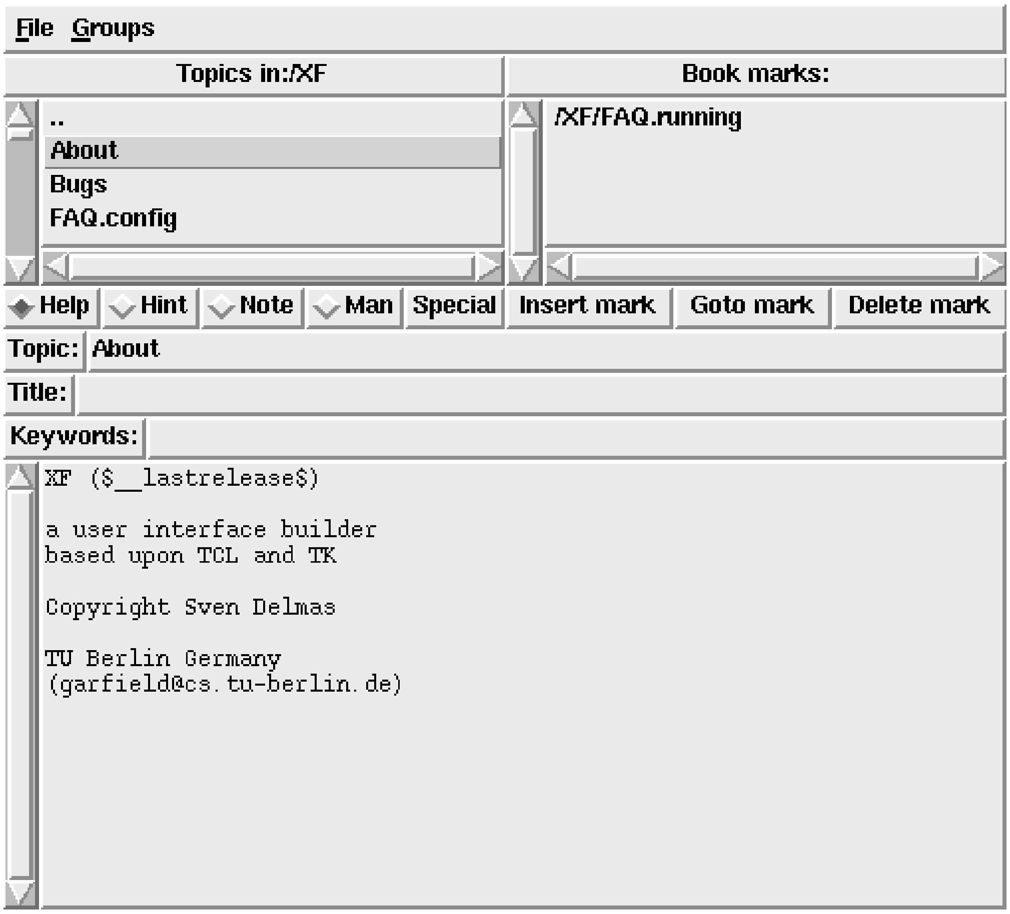


Figure A.4: The xfhelp program

The menubutton (`File`) only contains a quit button. The menubutton (`Groups`) allows to jump directly between the help pages for the different programs. The upper left list contains the help pages at the current level. A help page is displayed by clicking on the list item. The upper right list contains the book marks. Below the two lists, the currently selected help page is displayed. This help page contains the name of the topic, the title, the list of keywords, and the help text itself. The displayed help depends on the radiobuttons under the left listbox. It is possible to display Help pages, Hint pages, user writeable Note pages, Manual pages and a special Help page.

A book mark can be inserted by selecting the item in the left list, and pressing the button (`Insert mark`). To delete a book mark, the book mark is selected, and the button (`Delete mark`) is pressed. To jump to the book mark, double click the list item, or select the (`Goto mark`) button.

A.6.1 Help pages

The help pages can be distributed over different directory trees. This allows storing global help information in a global directory, while the user notes are stored in a local directory. New subtopics are created by creating a directory. New help pages can be added by creating a file with a .H extension in the help directory. Besides the main Help pages, additional help files can be created. They have the same name, but a different extension.

Hints have the extension .I, and notes have the extension .N. Special help information consists of a Tcl script ending with a .S. Special help files allow calling external programs, like ghostscript. To display a postscript file, the *Tcl* command DisplayPostscript is called. This is a built-in that checks if there exists an environment variable named XF_HELP_PS_CMD. If this variable exists, this is the command that is executed, with the file name as parameter. Otherwise, ghostscript is started. The variable "runPath" contains the absolute path name of the help page, and should precede the filename that is passed on to the command.

Help pages can start with two special information lines. One line can begin with '###Title '. This is the title for this help page. The second line can begin with '###Keywords '. This is a list of keywords attached to this help page. The following text is the help text that is displayed in the help area.

A.7 xfpixmap

Xfpixmap allows drawing pixmaps based upon the Xpm 3 format[7]. To use the program, the wish that is used must contain the TkPixmap patch. This patch implements the pinfo command. The functionality of xfpixmap is restricted to a basic functionality, and the performance is not very good especially for big pictures (big == >40x40). This program is part of the *XF* distribution, and is used to manipulate pixmaps.

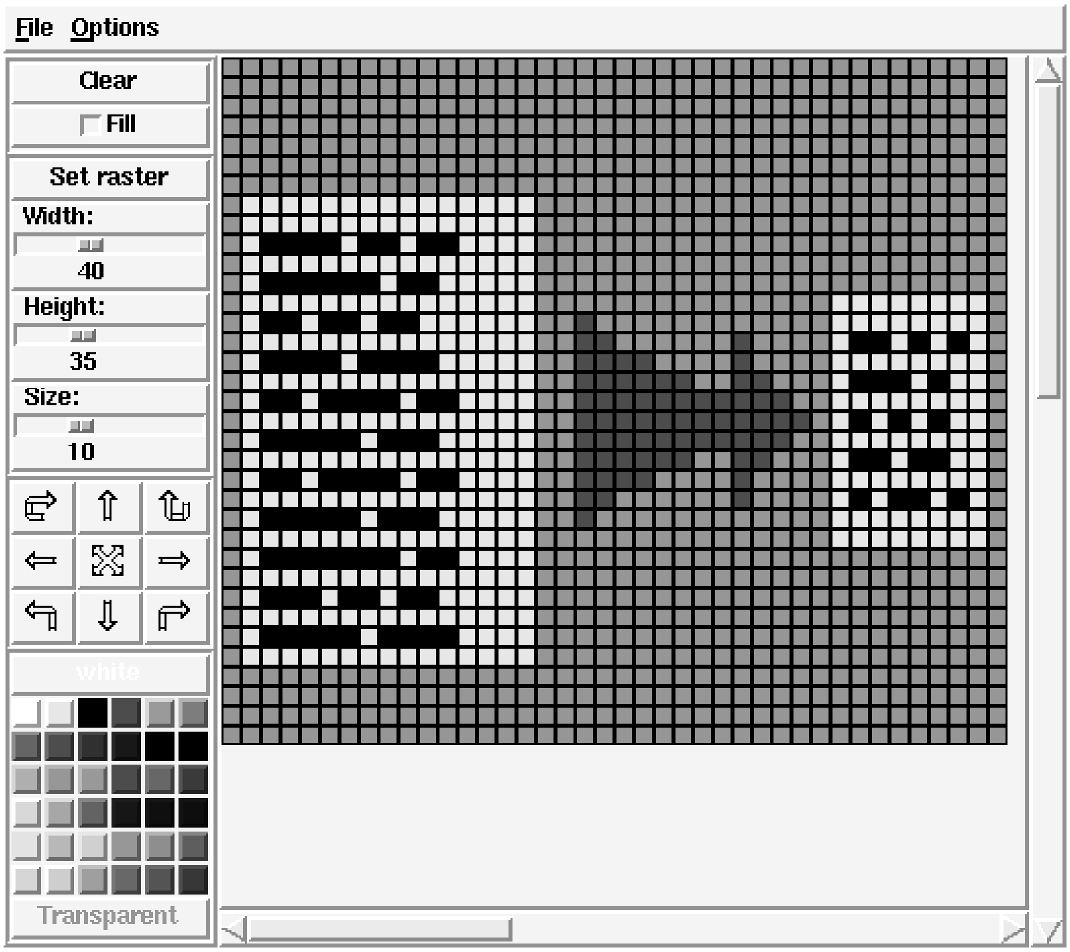


Figure A.5: The xfpixmap program

To set a color, the left mouse button is clicked at the pixel cell. The middle mouse button is used to set the transparent (background) color, and the right mouse button is used to set the drawing color to the color of the pixel cell under the mouse pointer. The current drawing color can be selected by pressing at the color button in the color array.

To change the color value for a color in the palette, this color is made the active color. Then the button that contains the name of the current color right over the palette is pressed. The new color can now be selected in the color selection box. By pressing at the button that contains the current transparent color, a new transparent color is selected.

The size of the picture can be changed with the scales at the left side. When a new width, height and pixel cell size have been selected, the button (`Set raster`) is pressed. The array of nine arrows is used to manipulate the drawing space in various ways. The buttons (`Fill`) and (`Clear`) do what they are supposed to do.

A.8 xftutorial

XFtutorial allows to interactively introduce a user to the usage of a *Tcl/Tk* program. The developer writes a tutorial script, leading the user through the functionality of the program. This program is part of the *XF* distribution, and is used to introduce the user into the handling of *XF*.

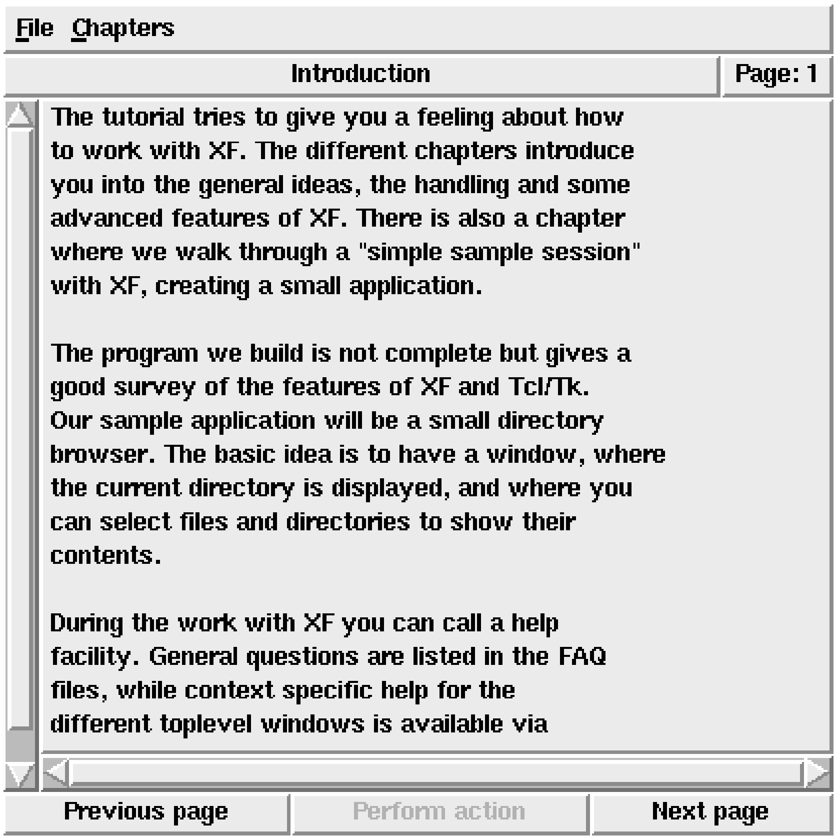


Figure A.6: The xftutorial program

The menubutton (`File`) contains an item to print the current help page, and to quit the program. The menubutton (`Chapters`) contains all chapters that are available. The text area displays the current help text. Below this area, there are three buttons that control the paging of the tutorial. Tutorial pages can have actions attached to them, which are performed when the next page is selected, or the Perform action button is pressed.

A.8.1 The script files

The script files must contain input following certain rules. One of the script files must define a global list named chapterList. This list contains a list of lists. The first element of the lists is the title of the chapter, followed by the underline position of the menu item, and the script name of this chapter. To insert a separator, an empty list is inserted.

Each script file must define two variables first. The first variable (*<chapter>Last*) specifies the last accessible page number. The second variable (*<chapter>LastSectionDone*) is an internal counter initialized with -1.

Each page is represented by three variables. The first variable (*<chapter>Name<pagenumber>*) contains the name of the page. The second variable (*<chapter>Text<pagenumber>*) contains the text to display. And the third variable (*<chapter>Command<pagenumber>*) contains the command to be executed. This command is sent to the application that uses the tutorial.

Appendix B

XF User's Guide

The functionality of *XF* is mainly accessed via the menubar and/or the iconbar of the main *XF* window. The items in the menubar and the iconbar usually call one of the procedures that *XF* offers for external access. These procedures are named in a special way. They begin with *XFProc*, followed by the real name of the procedure. The following sections describe the procedures that are available. They are grouped by functionality as it is represented by the menubuttons in the main *XF* window. Some procedures are not used in the default menubar, or are placed in another menu. The menu structure may be adapted by the user, so the structure of this chapter does not have to match the concrete menu structure.

It is possible to add new functionality to *XF* by writing a module that contains procedures following this naming convention. This module should be added to the directory where the *XF* source is located. To make the procedures accessible, the tclIndex file in this directory must contain the procedure names followed by the new module name. The procedures can be attached to menu items in the menubar configuration, or to icons in the iconbar configuration.

B.1 Main

B.1.1 XFProcMain

This procedure is automatically called when *XF* is started. The procedure pops up the main *XF* window. This window provides access to the functionality.

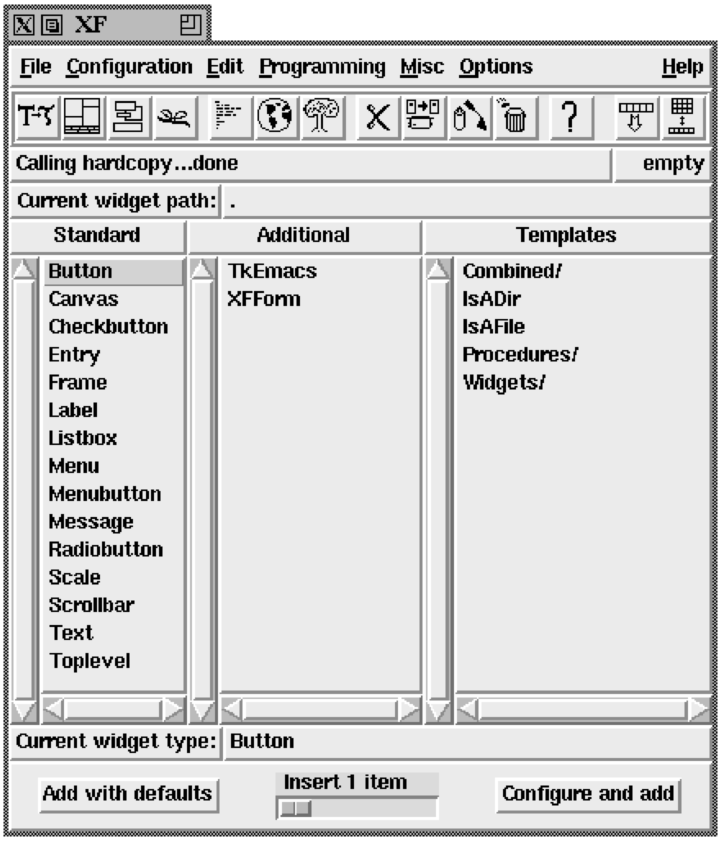


Figure B.1: The procedure XFProcMain

The main window is structured in several parts. At the top, there is a menubar and an iconbar. Both call the procedures that implement the various dialogs. They can be configured by the user.

Below the iconbar, a status line shows the status of the program, and a small label shows the status of the cut buffer.

The label (`Current widget path:`) has a menu attached to it, that contains a list of all toplevels. This menu is used to show/display the toplevels. The widget path right beside this label has menus attached to the dots, containing all children of the widget left from the dot. The user can navigate through the widget tree with these menus.

The three lists that occupy most of the space of the main window, contain the class names (and template names) that can be inserted into the application. A double click with the left mouse button inserts the widget with default parameters. A double click with the right mouse button inserts the widget after calling a parameter dialog. The two buttons at the bottom do the same as the double clicking. With the slider, the number of inserted widgets can be selected.

The left list contains the widgets that are part of the standard *Tk* distribution. The middle list contains additional widgets that have been added to the interpreter, and the right list contains the templates. Templates are complex widget structures and procedures that can be inserted and used as if the user wrote them himself.

B.2 File

B.2.1 XFProcFileEnterTCL (Enter TCL code)

Calling this procedure pops up a dialog box, where the user can enter *Tcl/Tk* code that is evaluated if he presses the button (`Send`) or the button (`Send + Clear`). The button (`Clear`) clears the text area. To remove the dialog box, the button (`Dismiss`) is pressed.

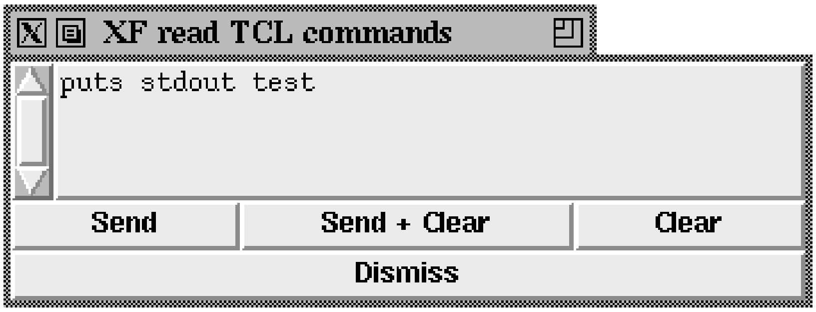


Figure B.2: The procedure XFProcFileEnterTCL

B.2.2 XFProcFileInsert (Insert...)

Calling this procedure pops up a standard file selector box (described in the templates section under FSBox). Here, the user can select a *Tcl/Tk* file that will be merged into the currently edited application. The code that is merged should not collide with the already built application. This is very important for the widget names, which are to be inserted. This procedure gets no parameters.

B.2.3 XFProcFileLoad (Load...)

Calling this procedure pops up a standard file selector box (described in the templates section under FSBox). Here, the user can select a *Tcl/Tk* file that will be loaded into the currently edited application. Before the code is loaded, the currently edited application is deleted form the interpreter. The loaded code should be careful with changes to the autoload path. If the autoload path is redefined, it is important that the *XF* source directory (*xfPath(src)*) is part of the new autoload path. This procedure gets no parameters.

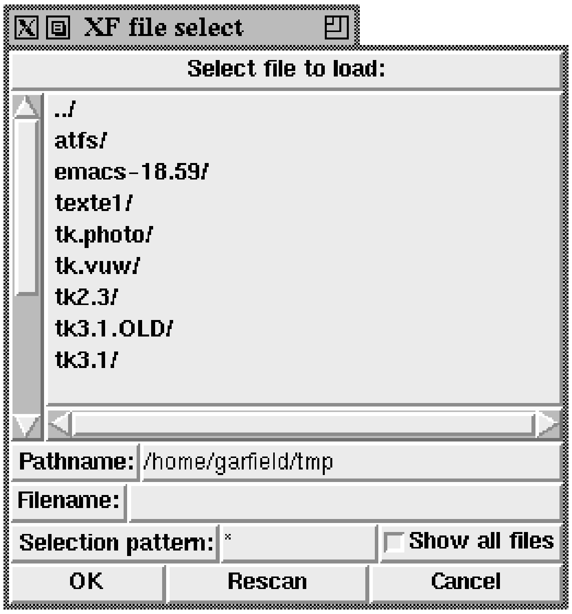


Figure B.3: The procedure XFProcFileLoad

B.2.4 XFProcFileNew (New)

Calling this procedure pops up a yes/no box asking the user if he really wants to remove all existing definitions. If the user answers yes, the current application is removed from the interpreter. This procedure gets no parameters.

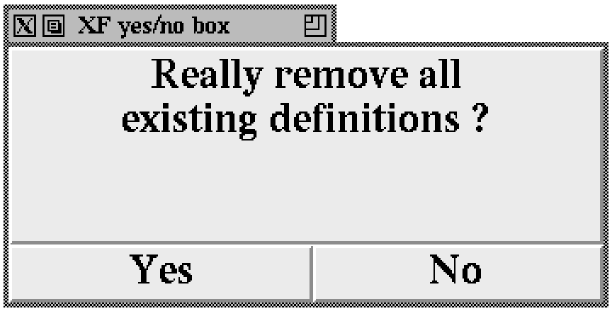


Figure B.4: The procedure XFProcFileNew

B.2.5 XFProcFileQuit (Quit)

Calling this procedure pops up a yes/no box asking the user if he really wants to quit the application. If he answers yes, the application is terminated. This procedure gets one optional parameter. This parameter is the value that is passed on to the *exit* call.

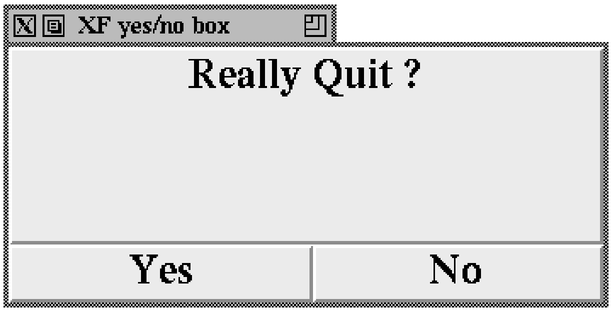


Figure B.5: The procedure XFProcFileQuit

B.2.6 XFProcFileSave (Save)

Calling this procedure saves the current application. The output is divided into the specified modules (see XFProcModule). This procedure gets no parameters.

B.2.7 XFProcFileSaveAs (Save as...)

Calling this procedure pops up a standard file selector box (described in the templates section under FSBox). Here, the user can select a filename that is used for the output. The complete application is saved to this file. Saving an application in this form does not change the specified module structure. If an application is loaded and saved in modules, the structure will be the same as before. This procedure gets no parameters.

B.3 Configuration

B.3.1 XFProcConfAddCurrentItem

This procedure inserts the currently selected type of item. The item type is selected in the main *XF* window from one of the three lists. The procedure takes one parameter that can be "add" or "config". This parameter specifies the way that the new element is inserted. "Adding" means that the inserted widget is created with default parameters, "configuring" means that the widget is configured by the user first and then inserted.

B.3.2 XFProcConfBinding (Binding)

This procedure activates the binding dialog for the currently selected widget or the widget that was passed on as first parameter. The binding dialog allows the setting of bindings for this specific widget.

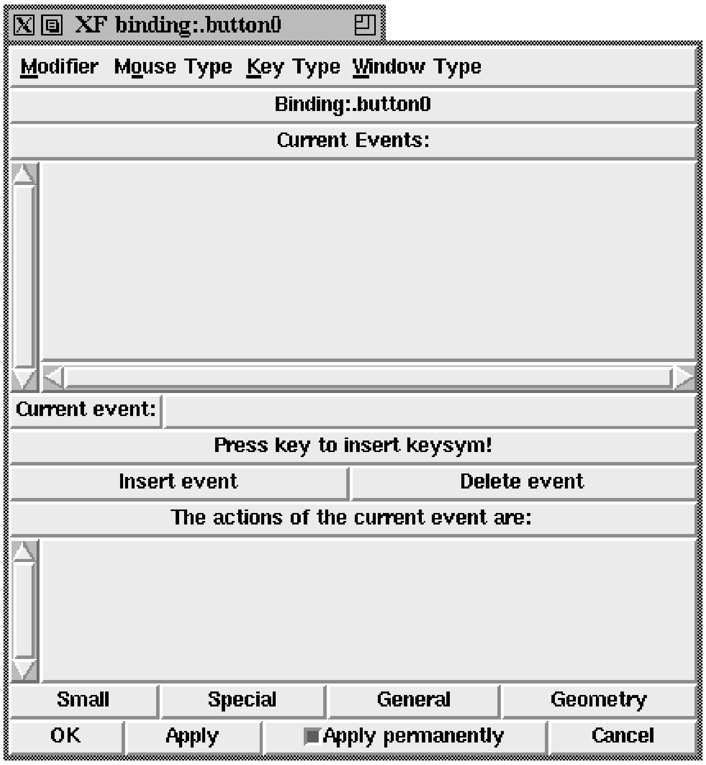


Figure B.6: The procedure XFProcConfBinding

The menubar at the top contains various event patterns that are inserted into the event string when the menu item is selected. Below the menubar the currently modified widget is displayed. Clicking on the name flashes the widget in the application.

Below the widget name, a list shows all events that are defined for this widget. It is possible to hide events (i.e. all Tk events are hidden). To show/hide an event, the first line of the command that is bound to the event must contain the string "# xf ignore me <level>", where level is a number from 0 to 9. Each level can be turned on or of separately (both for displaying and saving). To display a level, the appropriate checkbutton in the (`Options | General Options`) dialog is toggled.

The entry below the event listbox contains the currently selected event, or the new event composed with the menubar or by hand. To insert or delete the event, the two buttons (`Insert event`) and (`Delete event`) are used. The text widget at the bottom shows the *Tcl/Tk* command that is bound to the currently selected event, or the new command that should be inserted.

The buttons at the bottom allow the calling of other dialog boxes for the widget configuration. There are also buttons to apply the changes (if they are not applied permanently), and to terminate the dialog.

B.3.3 XFProcConfBindingAll (Binding for all widgets)

This procedure pops up the standard binding dialog. There are no parameters passed on to this procedure. The difference to a widget specific binding configuration is that the bindings modified with this dialog are applied to all widgets.

B.3.4 XFProcConfBindingClass (Binding for selected Class)

This procedure pops up the standard binding dialog. There is one optional parameter, specifying the widget class which is to be manipulated. The difference to a widget specific binding configuration is that the bindings modified with this dialog are applied to all widgets of the currently selected class. Class bindings are not saved automatically. Instead, the developer can save the current class bindings into a general binding file. This file can be set with an *XF* option or a commandline switch. If there exist special class bindings, this file should be part of the distribution. If this is too complicated, there exists an option in the (`Options | Source options`) dialog to embed the class bindings into the created code.

B.3.5 XFProcConfConfigure

This procedure is usually not called directly. It activates the configuration dialog for the currently selected widget, or the widget that was passed on as first parameter. The second parameter selects the dialog box to be displayed. This parameter is a number. 0 calls the packing dialog, 1 the placing dialog, 2 the default geometry dialog, 3 the binding dialog, 4 the default parameter dialog, and 5 the special parameter dialog. Each number matches a dialog, specified in a widget specific dialog file (from the directory ./elements or ./additionals). The next parameter should be an empty string. After that follows the widget class that is configured, and the type of the configuration (add or config). "Add" means that a new widget is created with default parameters, and "config" means that the parameter setting dialog is called. For the standard calls of this procedure there are wrappers that make the calling easier.

B.3.6 XFProcConfGeometryDefault

This procedure activates the default geometry handling dialog for the currently selected widget, or the widget that was passed as first parameter. The default geometry handler can be specified in the dialog box (`Options | General Options`).

B.3.7 XFProcConfInsertTemplate

This procedure inserts the specified template into the application. The template name passed on as parameter refers to the currently displayed template directory. If the name is a directory, the current template directory is changed to this directory.

B.3.8 XFProcConfInsertWidgetDefault

This procedure gets a widget class name as parameter, and inserts a new widget of this class into the currently selected widget. The widget is inserted with default parameters.

B.3.9 XFProcConfInsertWidgetConfig

This procedure gets a widget class name as parameter, and inserts a new widget of this class into the currently selected widget. Before the widget is actually inserted, a parameter dialog box is popped up, where the user can configure the widget (including the widget name).

B.3.10 XFProcConfLayout (Layout)

This procedure pops up the layout dialog box. This dialog box provides access to the interactive (direct) placing and packing of widgets. Usually, direct manipulation of the widgets is only allowed when this dialog box is displayed. This can be changed in the options dialog (`Options | General Options`).

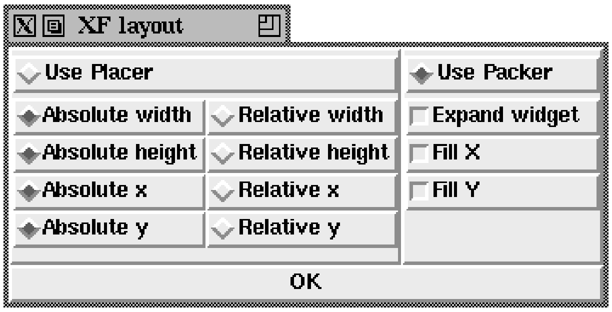


Figure B.7: The procedure XFProcConfLayout

The left side of the dialog box contains placer options. Here, the user can select if the geometry is set in absolute or relative values. The position of a widget can be set by pressing the left mouse button together with Modifier1. If the button is pressed on the border of the widget, the widget can be resized. If a parent contains no placed children, the first time a widget is placed in that parent, a small dialog box warns the user, and gives him the choice to place the parent, too, to keep the size of the parent widget or to abort the placing.

The right side of the dialog box gives access to a number of packer options. These options are applied to a packed widget when the left mouse button is pressed together with Modifier1 in the border region of the widget. To move a widget to a specific border of the parent, the widget is selected with the left mouse button together with Modifier1. Then, the mouse is moved to the border.

B.3.11 XFProcConfPacking (Packing)

This procedure activates the packing dialog for the currently selected widget, or the widget that was passed on as first parameter. With this dialog box, it is possible to change the packing of the complete widget tree. This means that the user just has to call this dialog box once to layout the complete widget tree.

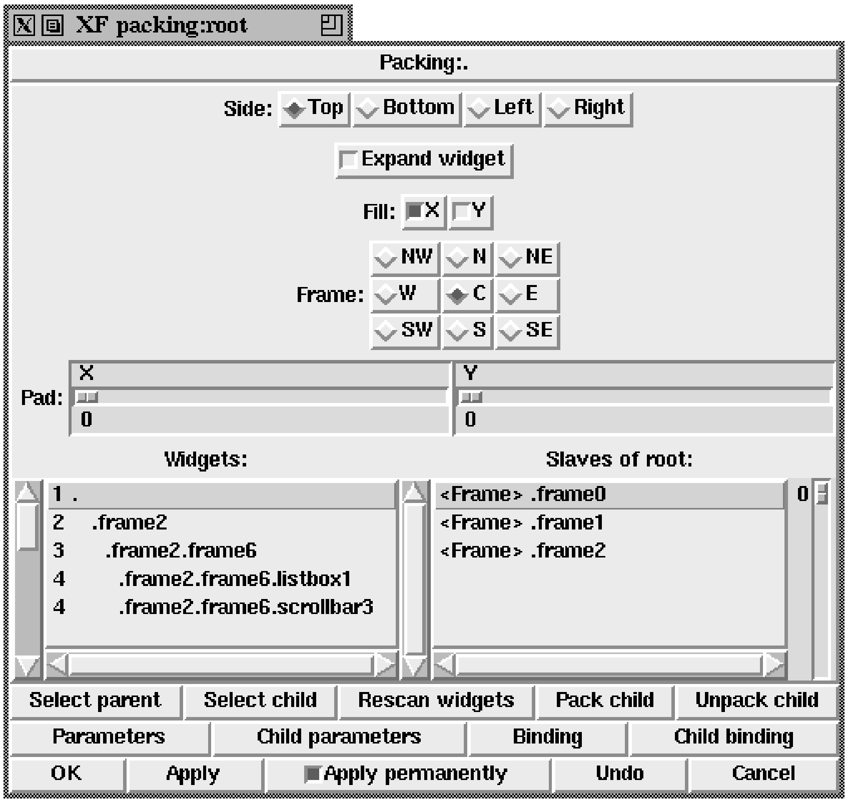


Figure B.8: The procedure XFProcConfPacking

The top area contains the available packer options. The options show the current setting of the child selected in the right list at the bottom. Selecting another child in that list updates the settings. The left list contains the widget tree. By double clicking at a widget in that list, this widget is made the current master. The right list contains the packed children of this master.

The buttons below the listboxes allow the navigation in the widget tree. Selecting the parent means that the parent of the current master is made the new master. Selecting a child means that the currently selected child in the right list is made the new master. As there exist two independent geometry manager, it can happen that a child of a widget should be managed with the other geometry manager. This is done with the two buttons (`Pack child`) and (`Unpack child`).

The remaining buttons at the bottom give access to the other widget specific dialogs. There is always one button that activates the parameter box for the current master, and one button that activates it for the current child. There are also buttons that apply the changes and terminate the dialog box.

B.3.12 XFProcConfParametersDefault (Parameters)

This procedure activates the default parameter setting dialog for the currently selected widget, or the widget that was passed on as first parameter. This dialog is usually the most important dialog.

B.3.13 XFProcConfParametersGeneral (Parameters (general))

This procedure gives access to the general parameter dialog. *XF* supports a subset of widget resources with *XF* specific dialog boxes. This only covers the most frequently used resources. Special resources can be accessed with this dialog box.

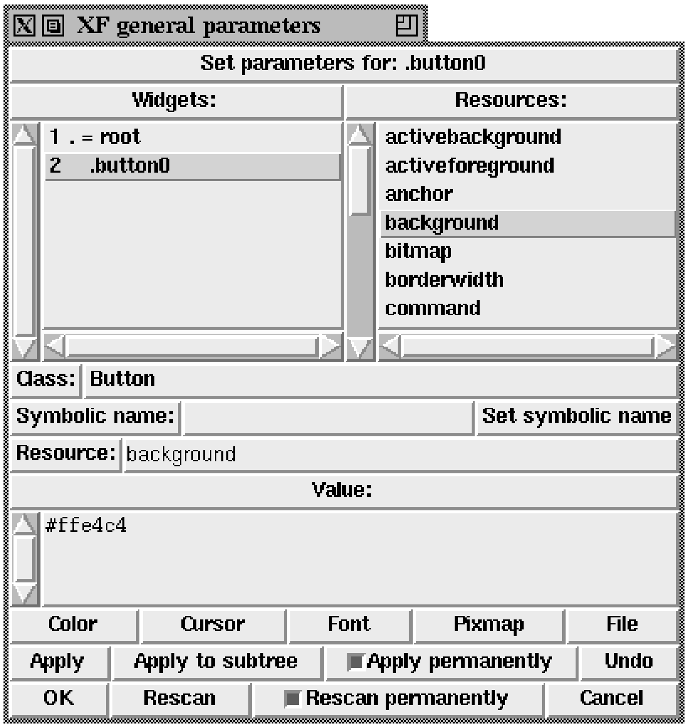


Figure B.9: The procedure XFProcConfParametersGeneral

The upper left list shows all widgets in the application. To change the current widget, the user clicks on the widget name. The right list shows the names of all available resources for the current widget.

Below these lists, the class of the selected widget is displayed. The symbolic name is shown as well and can be manipulated. The resource field contains the name of the currently selected resource, and the text field at the bottom shows the value of the resource.

A number of buttons at the bottom provides access to some dialog boxes, where values for standard resources (like colors) can be interactively selected. The remaining buttons at the bottom are used for setting the resources for the current widget (or for all descendants of the current widget), and for terminating the dialog box.

B.3.14 XFProcConfParametersGroups (Parameters (widget groups))

This procedure calls the dialog box for parameter setting for groups of widgets. Very often, parameters (like foreground) have to be set for a great number of widgets. Instead of calling the parameter setting dialog for each widget separately, this dialog allows the interactive selection of widgets, and the setting of parameters for these widgets.

The upper left list contains all widgets in the application. A widget is added to the selection by clicking on its name. A selected widget is displayed in the middle list, where it can be removed with a click. The right list shows a list of resources. A resource name can be selected by clicking on the name.

Below the lists, two lines allow the selection of widgets via shell style expressions, or regular expressions. The next entry contains the resource name. The name can be entered by clicking on a list item in the upper right list, or by typing it by hand. The text widget at the bottom contains the value for the resource.

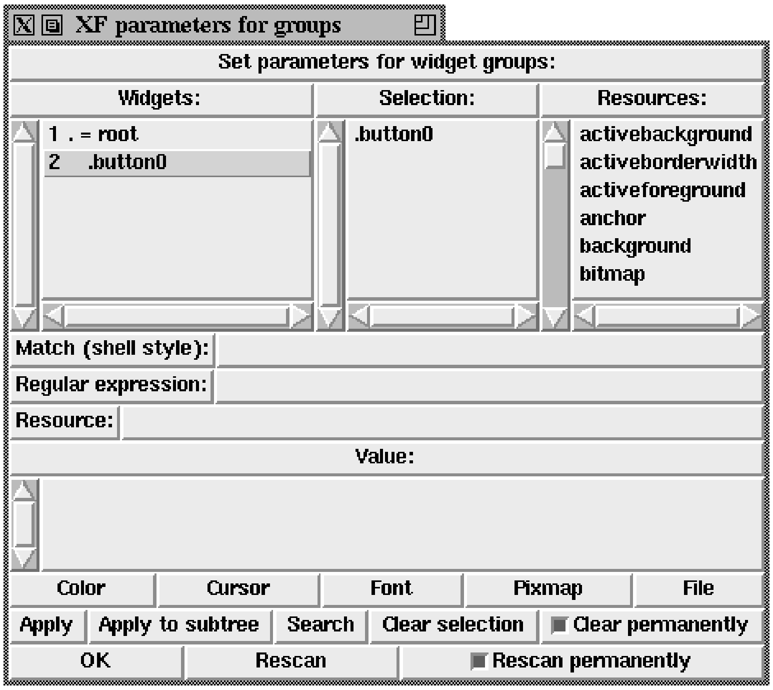


Figure B.10: The procedure XFProcConfParametersGroups

A number of buttons at the bottom provides access to some dialog boxes, where values for standard resources (like colors) can be interactively selected. The remaining buttons at the bottom are used to set the resources for the selected widgets (or for all descendants of the selected widgets), to clear the selection etc..

B.3.15 XFProcConfParametersSmall (Parameters (small))

This procedure activates the small parameter setting dialog for the currently selected widget, or the widget that was passed on as first parameter. This dialog is usually the most important dialog, and contains the most important resources that can be changed for a widget. If a resource is not accessible with this dialog, the general parameter dialog must be used.

B.3.16 XFProcConfParametersSpecial (Parameters (special))

This procedure activates the special parameter dialog for the currently selected widget, or the widget that was passed as first parameter. A special parameter dialog is a complex interface for special features of a widget class. I.e. the handling of menu items and canvas items is done in a special parameter dialog. If a widget has no special dialog box, nothing happens when this procedure is called.

B.3.17 XFProcConfPlacing (Placing)

This procedure activates the placing dialog for the currently selected widget, or the widget that was passed on as first parameter. With this dialog box, it is possible to change the placing of the complete widget tree. This means that the user only has to call this dialog box once to layout the complete widget tree.

The top area contains the available placer options. The options show the current setting of the child selected in the right list at the bottom. Selecting another child in that list updates the settings. The left list contains the widget tree. By double clicking at a widget in that list, this widget is made the current master. The right list contains the children placed (managed) by this master.

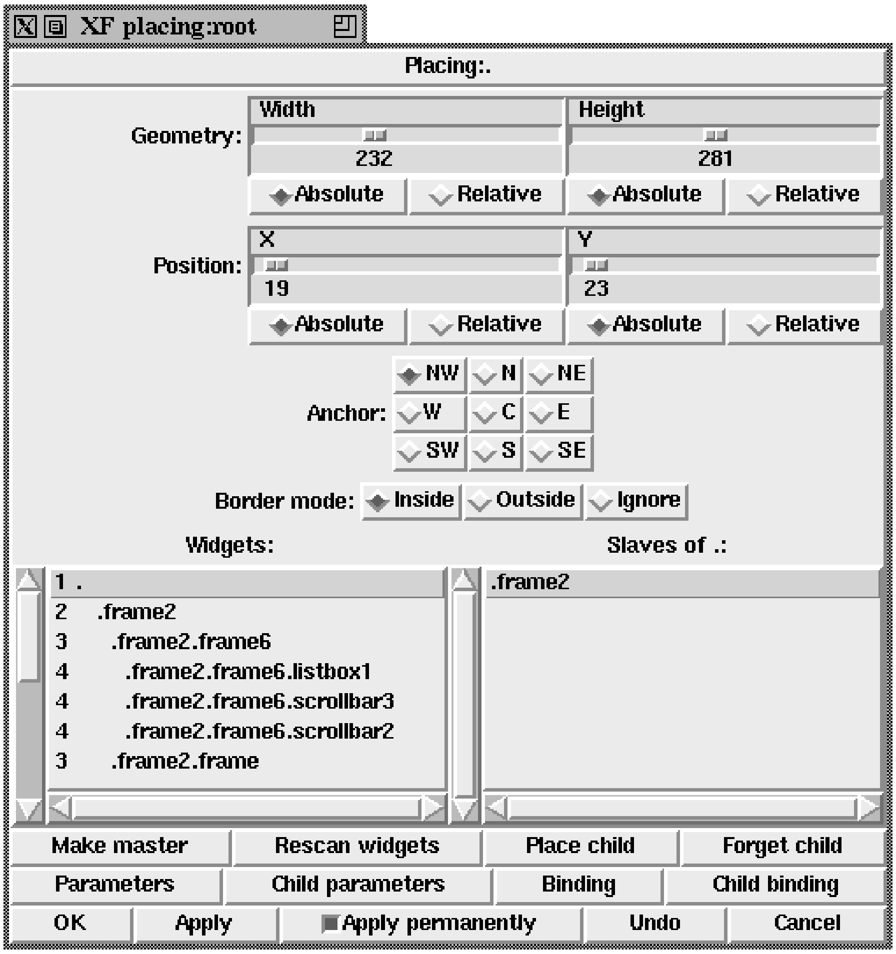


Figure B.11: The procedure XFProcConfPlacing

As there exist two independent geometry managers, it can happen that a child of a widget should be managed with the other geometry manager. This is done with the two buttons (`Place child`) and (`Forget child`).

The remaining buttons at the bottom give access to the other widget specific dialogs. There is always one button that activates the parameter box for the current master, and one button that activates it for the current child. There are also buttons that apply the changes and terminate the dialog box.

B.4 Edit

B.4.1 XFProcEditClearCut (Clear Cutbuffer)

Calling this procedure clears the current cutbuffer. The widget structure stored in the cutbuffer is lost. If the contents of the cutbuffer were displayed in a dialog box, this dialog box is removed. This procedure takes no arguments.

B.4.2 XFProcEditCopy (Copy)

Calling this procedure copies the specified widget or the current widget (if no widget was specified) to the cutbuffer. The widget structure remains in the application, and the cutbuffer contains a copy of this widget structure. This procedure takes one optional argument. The argument specifies the widget path to be copied. If no widget path is specified, the current widget path is used.

If widgets inside of the copied widget structure are used by other widgets, these commands have to be adapted when the widget tree is pasted.

B.4.3 XFProcEditCut (Cut)

Calling this procedure cuts the specified widget or the current widget (if no widget was specified) to the cutbuffer. The widget structure is removed from the application, and the cutbuffer contains the widget structure. This procedure takes one optional argument. The argument specifies the widget path to be cut. If no widget path is specified, the current widget path is used.

If widgets inside of the cut widget structure are used by other widgets, these commands have to be adapted when the widget tree is pasted.

B.4.4 XFProcEditDelete (Delete)

Calling this procedure deletes the specified widget or the current widget (if no widget was specified). The widget structure is removed from the application. This procedure takes one optional argument. The argument specifies the widget path to be deleteed. If no widget path is specified, the current widget path is used.

B.4.5 XFProcEditLoadCut (Load Cutbuffer)

Calling this procedure pops up a standard file selector box (described in the templates section under FSBox). Here, the user can select a cutbuffer file that was saved using the procedure XFProcEditSaveCut. The file to be loaded must be a cutbuffer file, in order to be accessible for pasting. This procedure takes no arguments.

B.4.6 XFProcEditLoadTemplate

Calling this procedure pops up a standard file selector box (described in the templates section under FSBox). Here the user can select a template file that was saved using the procedure XFProcEditSaveCutAsTemplate. This procedure takes no arguments.

B.4.7 XFProcEditMakeAProc (Make a procedure)

Calling this procedure creates a procedure from the specified widget or the current widget (if no widget was specified). The name of the new procedure is V<WidgetClass><pathName>. The widget structure remains in the application. Making a widget structure a procedure allows it to dynamically reproduce complex widget structures. This can be useful for applications where several toplevels contain the same sub widget structure. If no widget path is specified, the current widget path is used.

The created procedure is called with one parameter to create a new instance of the widget tree. This parameter specifies the parent name of the widget. The programmer has to guarantee that the parent will not contain the same widget structure more than once.

The call of the procedure can have additional parameters which are used to configure the created widget structure. Calling the procedure with a parent name that already contains the widget structure allows it to reconfigure the widget tree. The parameters are the usual resource pairs. First the resource name, and then the new value. The resource is set for all widgets that support this resource. If a resource is to be set for one specific subwidget, the complete pathname is specified as a parameter, followed by the usual resourcename/resourcevalue pair. There are three special configuration parameters that are only used when the widget structure is created. They are named -startupSrc, -middleSrc and -endSrc. Each parameter gets one argument. The startupSrc is evaluated before any widget is created. The middleSrc is evaluated when all widgets are created, but before they are mapped. The endSrc is evaluated before the procedure is finished.

B.4.8 XFProcEditPaste (Paste)

Calling this procedure inserts the current cutbuffer to the specified widget or the current widget (if no widget was specified). The widget structure remains in the cutbuffer. This procedure takes one optional argument. The argument specifies the widget path where the cutbuffer is to be inserted. If no widget path is specified, the current widget path is used.

B.4.9 XFProcEditSaveCut (Save Cutbuffer)

Calling this procedure pops up a standard file selector box (described in the templates section under FSBox). Here, the user can select a filename that is used to save the cutbuffer. Later, this cutbuffer file can be loaded using the procedure XFProcEditLoadCut. This procedure takes no arguments.

B.4.10 XFProcEditSaveCutAsTemplate (Save Template (cut buffer))

Calling this procedure pops up a standard file selector box (described in the templates section under FSBox). Here, the user can select a filename that is used to save the cutbuffer. The cutbuffer is saved as a template, which means that the filename has the extension .t, and the file should be located in one of the template directories. This procedure takes one argument. The value should always be "cb".

B.4.11 XFProcEditShowCut (Show Cutbuffer)

Calling this procedure pops up a dialog box containing the current cutbuffer. This can be in textual representation, or as a widget structure. If the parameter of the procedure is "tree", the cutbuffer is displayed as a widget tree:

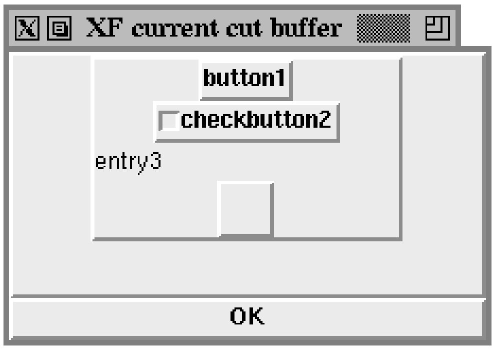


Figure B.12: The procedure XFProcEditShowCut (tree)

If the parameter is "script", the *Tcl/Tk* script is displayed:

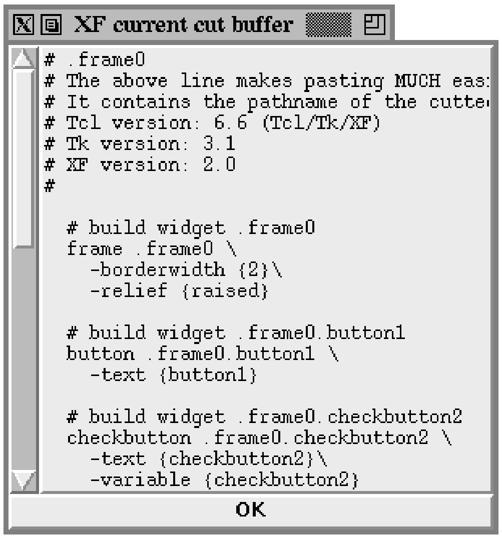


Figure B.13: The procedure XFProcEditShowCut (script)

B.5 Programming

B.5.1 XFProcProgCommands (Commands)

This procedure pops up the command handling dialog. This dialog provides access to the user defined procedures and the *Tcl/Tk* commands.

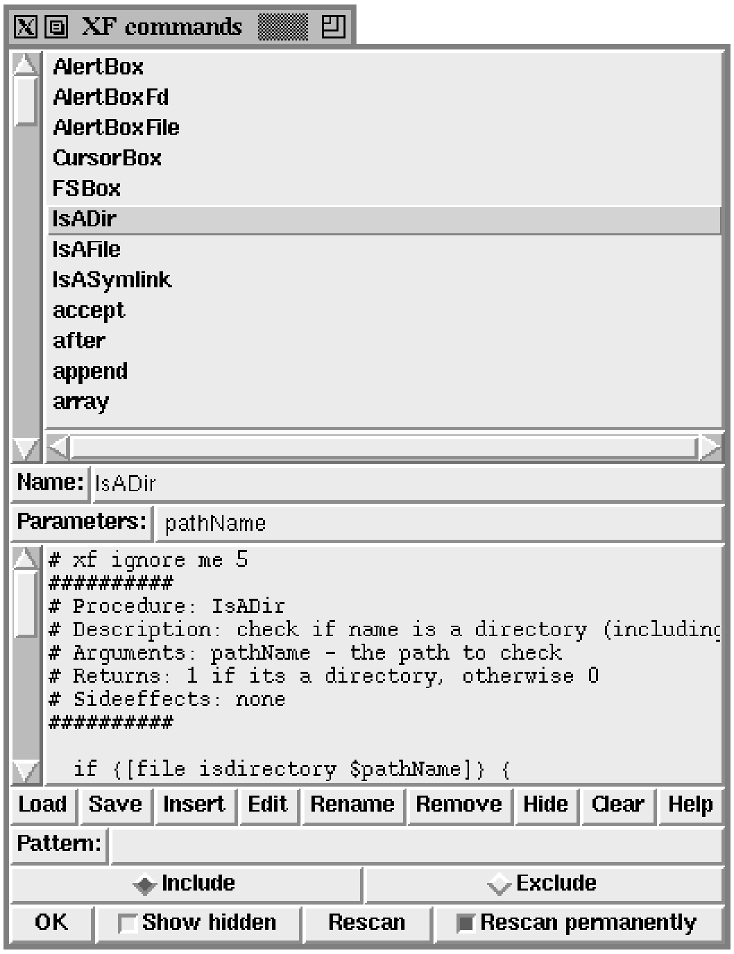


Figure B.14: The procedure XFProcProgCommands

The upper list shows all commands. By clicking on a name, the name, the arguments and the body are inserted into the three fields below the list.

To restrict the displayed commands to a subset, a pattern can be specified. This pattern can be used to include matching commands, or to exclude them.

The buttons at the bottom of the window control the changing of the commands. The (`Insert`) button creates a new procedure with the current name, arguments and body. The (`Edit`) button updates the current procedure. The (`Rename`) button pops up a dialog box where the new name can be entered. The (`Remove`) button removes the current procedure. The (`Hide`) button allows hiding a procedure. A hidden procedure no longer exists as a procedure. To unhide a procedure the user switches to the hidden procedures and presses the (`Unhide`) button. The (`Clear`) button clears the text fields, and the (`Help`) button calls the help program for the current procedure. The remaining buttons at the bottom control the rescan of the variables, and allow the termination of the dialog.

The (`Save`) button allows the saving of procedures to a ShapeTools archive or a plain file. These procedures can later be reused in other programs. The save dialog allows the definition of a message that is attached to the saved file.

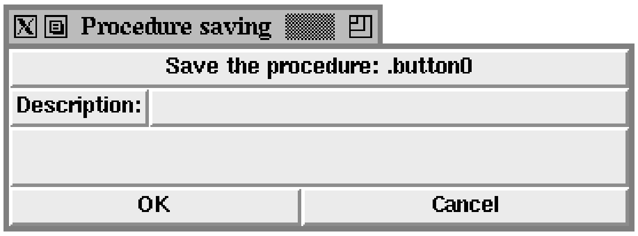


Figure B.15: The procedure XFProcProgCommands (saving)

The (`Load`) button allows the loading of externally saved procedures. The procedure can be loaded or displayed.

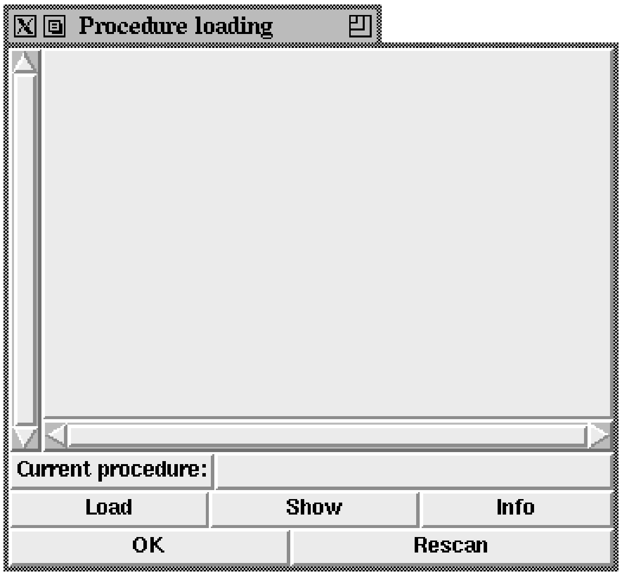


Figure B.16: The procedure XFProcProgCommands (loading)

B.5.2 XFProcProgEditScript (Edit script)

This procedure saves the current application, and displays it in a text box. The user can view and edit the resulting code. When he confirms the changes, the application is reloaded.

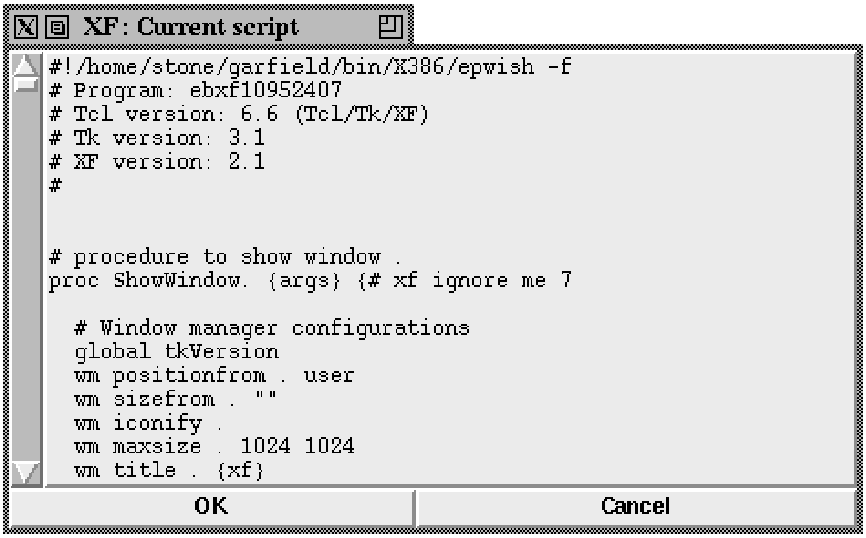


Figure B.17: The procedure XFProcProgEditScript

B.5.3 XFProcProgEndSrc (End source)

This procedure activates the procedure handling dialog with "EndSrc" as procedure name. This procedure is evaluated before the control is passed on to the *Tk* main loop. It is the last code that is executed at the startup.

B.5.4 XFProcProgErrors (Error status)

This procedure pops up a dialog box, where the last error that occurred during the execution of *XF* is displayed. Most errors that are displayed are "correct", which means that it is ok that they occur.

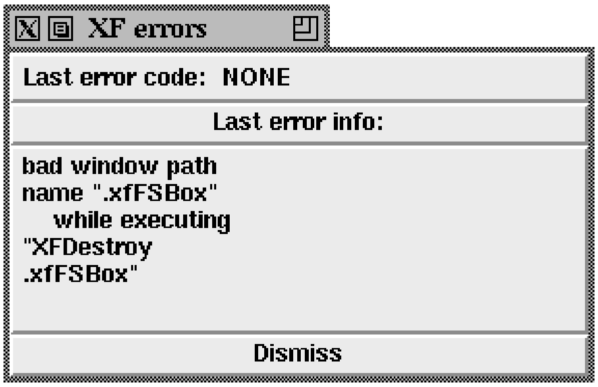


Figure B.18: The procedure XFProcProgErros

B.5.5 XFProcProgGlobals (Global variables)

This procedure pops up the globals handling dialog. This dialog provides access to the global variables.

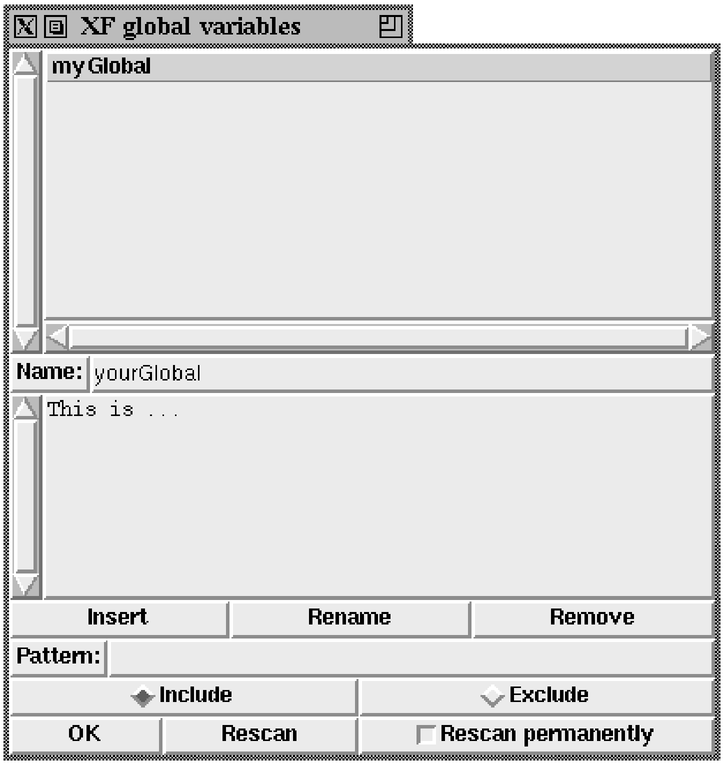


Figure B.19: The procedure XFProcProgGlobals

The upper list shows all global variables. By clicking on a name, the name and the current value are inserted into the two fields below the list.

To restrict the displayed variables to a subset, a pattern can be specified. This pattern can be used to include matching variables, or to exclude them.

The buttons at the bottom of the dialog control the changing of the global variables. The insert button creates a global variable with the current name containing the current value. The rename button pops up a dialog box where the new name can be entered. The remove button removes the current variable, and the clear button clears the text fields. The remaining buttons at the bottom control the rescan of the variables, and allow the termination of the dialog.

B.5.6 XFProcProgProcs (Procedures)

This procedure pops up the procedure handling dialog. This dialog provides access to the user defined procedures. The handling is the same as for the command handling dialog, except that this dialog does not display commands.

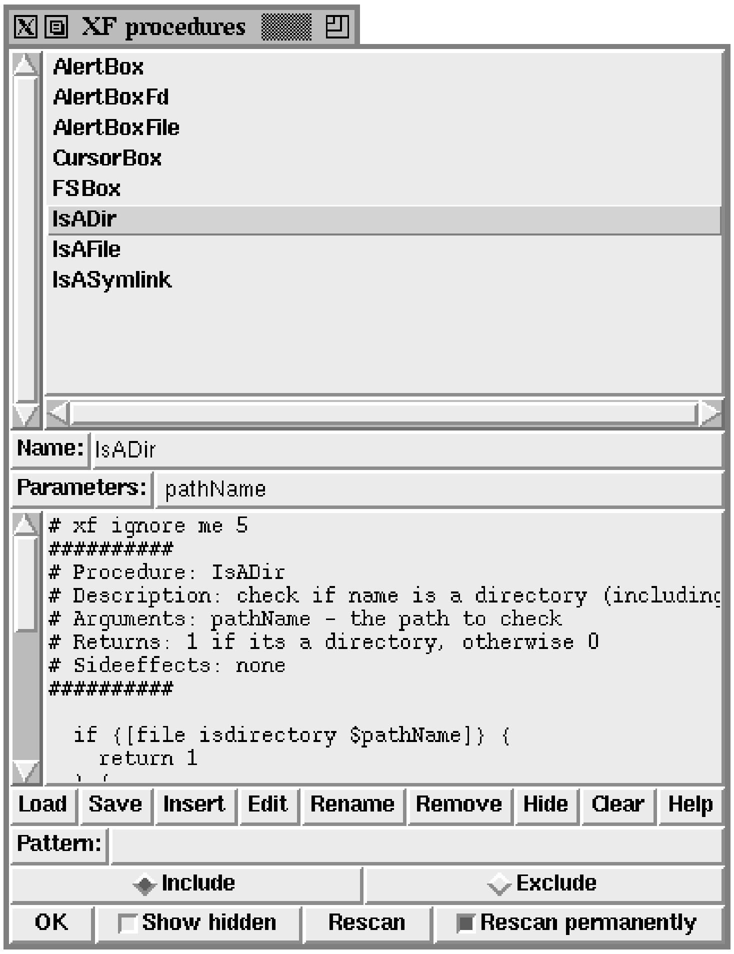


Figure B.20: The procedure XFProcProgProcs

B.5.7 XFProcProgShowScript (Show script)

This procedure saves the current application, and displays it in a text box. The user can view the resulting code.

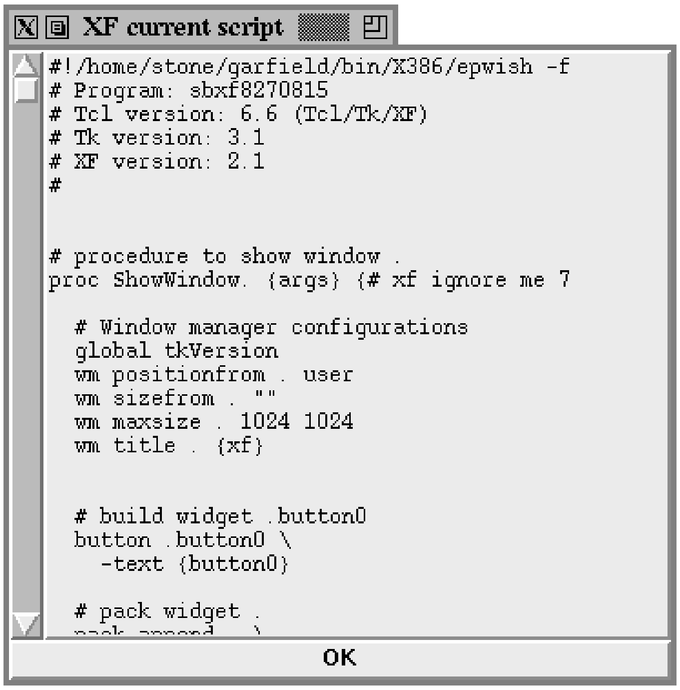


Figure B.21: The procedure XFProcProgShowScript

B.5.8 XFProcProgStartupSrc (Startup source)

This procedure activates the procedure handling dialog with "StartupSrc" as procedure name. This procedure is evaluated as first code in the application. It can be used to parse the commandline etc..

B.5.9 XFProcProgWidgetTree (Widget tree)

This procedure activates the widget tree dialog. Here, the widget tree is displayed as a simple graph. The user can restrict the displayed widget tree to a subtree.

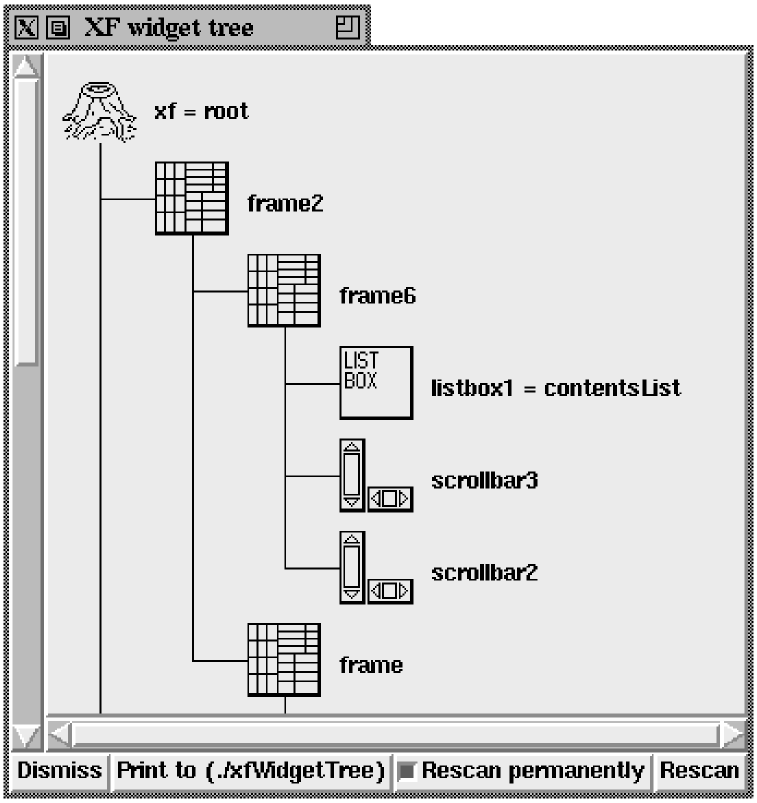


Figure B.22: The procedure XFProcProgWidgetTree

The items representing the various widgets have a popup menu attatched to the left mouse button. Here, all widget specific dialogs can be activated. With this menu, the user can also restrict the displayed widgets to a subtree, and remove the restriction. It is possible to set the current widget with the standard binding (usually with a doubleclick with the middle mouse button). The button (`Print to (./xfWidgetTree)`) writes a postscript hardcopy of the displayed widget tree to the file ./xfWidgetTree.

B.6 Misc

B.6.1 XFProcMiscAliases (Aliases)

This procedure gives access to the alias feature of *XF*. It is possible to define procedure names that can be used instead of existing procedure names. This is very useful for complex widget names which can now be abbreviated to a short precise name.

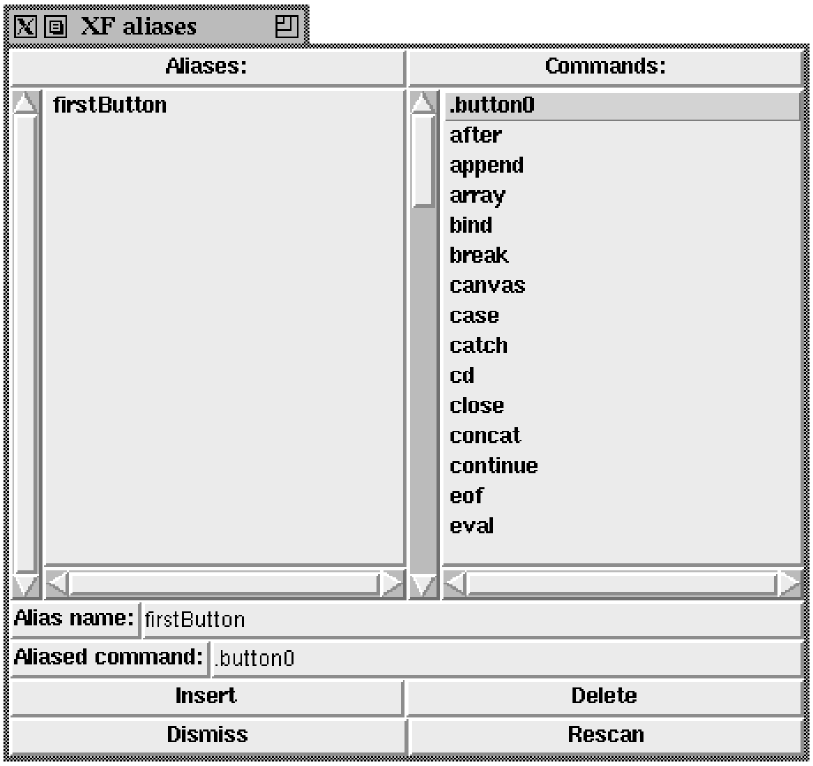


Figure B.23: The procedure XFProcMiscAliases

The left list contains the known aliases, and the right list the known procedures and commands. To insert a new alias, the new name is typed into the entry labeled (`Alias name:`). By clicking on a procedure or command name from the right list, this name is inserted into the entry below the alias name. The two buttons (`Insert`) and (`Delete`) control the insertion and deletion of the aliases.

B.6.2 XFProcMiscAppDefaults (Application defaults)

This procedure calls the external program *xfappdef*. The procedure gets an application class name specifying the resource file to edit. A description of the program can be found in the appropriate part of this documentation.

B.6.3 XFProcMiscEdge (Widget tree (edge))

This procedure dumps the current widget tree as an edge grl file, and automatically starts edge. Edge is a program that can display graphs.

B.6.4 XFProcMiscHardcopy (Hardcopies)

This procedure calls the external program *xfhardcopy*. A description of the program can be found in the appropriate part of this documentation.

B.6.5 XFProcMiscModules (Module structure)

Calling this procedure pops up the module structure dialog box. This procedure gets no parameters. An *XF* generated program can be packed into one file, containing the complete code. It is also possible to split the code into several modules. This is done in this dialog box.

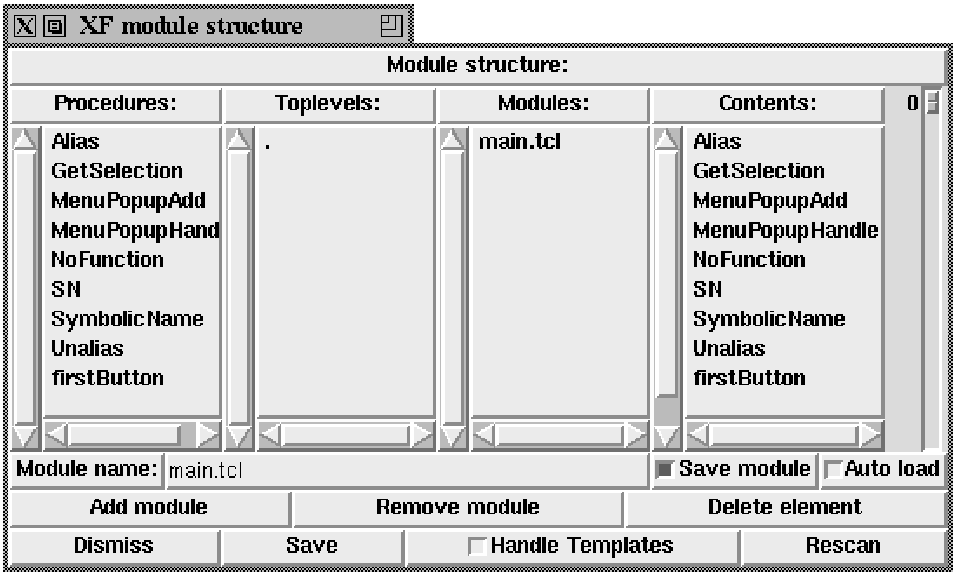


Figure B.24: The procedure XFProcMiscModules

To add a new module, the new module name is entered, and the (`Insert module`) button is pressed. The current module is selected from the module list. The contents of that module are displayed in the right list. The order of the module contents can be changed by selecting a name, and changing its position with the slider right beside the list.

To add new elements (procedures and toplevels) to a module, a name is selected in one of the left lists. To remove an element from a module, the element is selected in the right list and the (`Delete element`) button is pressed. To remove a module, the module is selected in the left list, and the (`Delete module`) button is pressed.

If the contents of a module should be auto loadable, the checkbutton (`Auto load`) right beside the module name is toggled. This means that a tclIndex file is created, and the needed code for auto loading is created.

To restrict the saving to a subset of the modules, the checkbutton (`Save module`) right beside the module name can be toggled. If the checkbutton is deselected for a module, this module will not be saved. The current selection of modules to be saved can be stored to a local file named .xf-save-modules with the procedure XFProcOptionsSaveModules.

This dialog is also used to create templates. By clicking the (`Handle Templates`) button, the selected procedures and the widget path are written to a template that is created when the (`Save`) button is pressed.

B.6.6 XFProcMiscPixmaps (Pixmaps)

This procedure activates a dialog box in which the user can select pixmaps to be preloaded. Preloading means that the generated source code contains the bitmaps/pixmaps in string form. To do this, the TkPixmap extension *pinfo* is required.

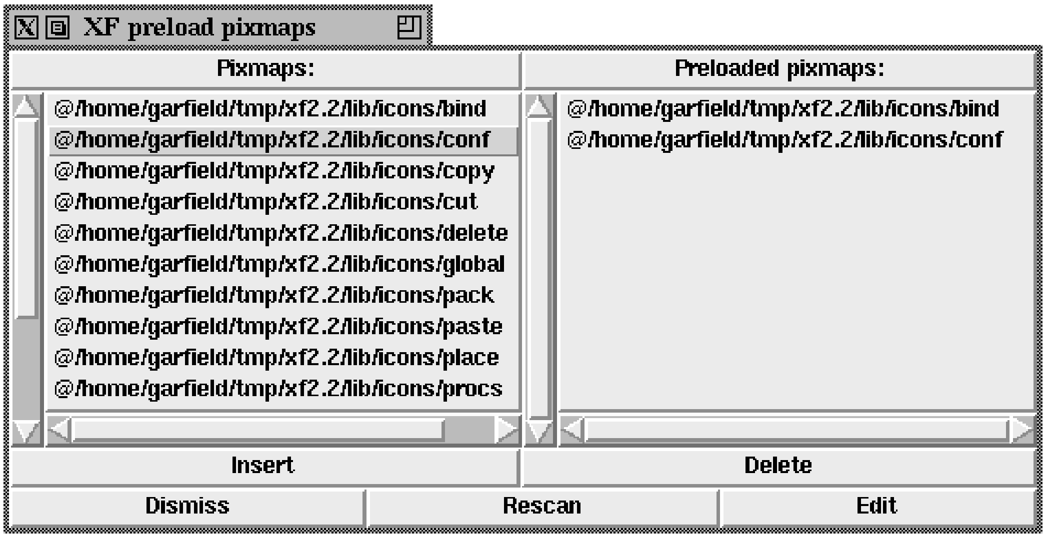


Figure B.25: The procedure XFProcMiscPixmaps

The left list contains all bitmaps/pixmaps that are currently loaded into *Tk*. By double clicking on a name, this name is inserted into the right list. This means that this bitmap/pixmap will be preloaded. To remove a selected name from the right list, double click on the name. With the (`Edit`) button, an external bitmap editor can be called, to modify the bitmap/pixmap.

B.6.7 XFProcMiscSaveEdge (Dump tree (/tmpPath/....grl))

This procedure dumps the current widget tree as an edge grl file. Edge is a program that can display graphs. The output file is written to the temporary directory, and is named <ProgramName>.grl.

B.6.8 XFProcMiscTestProgram (Test program)

This procedure saves the current application, and starts an extra wish, where the user can test the application.

B.7 Options

B.7.1 XFProcOptionsBindings (Bindings)

This dialog box provides access to the bindings that are used by *XF*. Most bindings in *XF* can be adapted by the user, to allow *XF* to work with different window manager configurations.

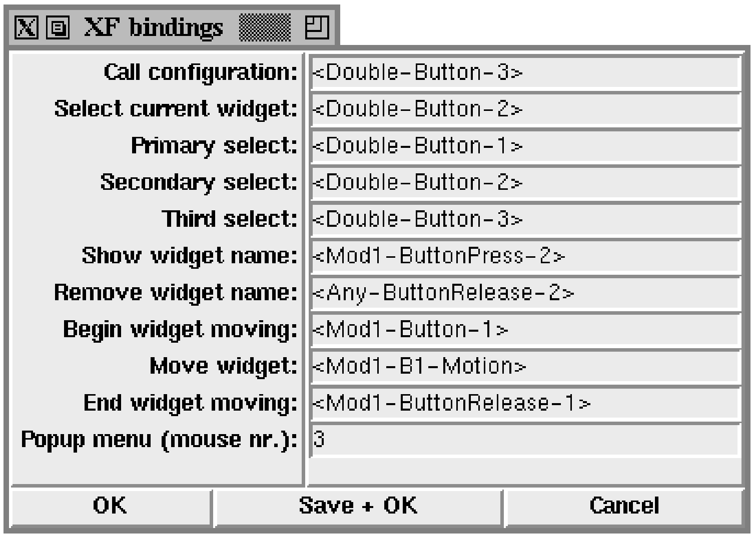


Figure B.26: The procedure XFProcOptionsBindings

The single options have the following meanings (some option names may be abbreviated):

Option name	Purpose
Call configuration	This is the binding to activate parameter setting. This event works for each widget in the application, and also for some parameter setting fields in the *XF* parameter dialogs.
Select current widget	This event is used to make one widget in the application the current widget.
Primary select	This is the general (preferred) selection event.
Secondary select	This is the alternative for the primary select. This is only used when the primary select is already used (almost never required).
Third select	This is the alternative for the primary and secondary select.
Show widget name	This event allows it to display the widget name of the widget under the mouse pointer in a dialog box (the name is also inserted into the cutbuffer, so it can be pasted).
Remove widget name	This event must correspond to the "Show widget name" event. This event removes the dialog box showing the widget name.
Begin widget moving	This event starts the interactive placing/sizing of a widget. It must correspond to the other moving/sizing events.

Option name	Purpose
Move widget	This event is the moving event that is used to update the widget position during the moving. It must correspond to the other moving/sizing events.
End widget moving	This event ends the interactive placing/sizing of a widget. It must correspond to the other moving/sizing events.
Popup menu (mouse nr.)	This is the number of the mouse button that should be used to display a popup menu. Popup menus are available in the widget tree.

B.7.2 XFProcOptionsGeneral (General options)

This dialog box provides access to the general *XF* options.

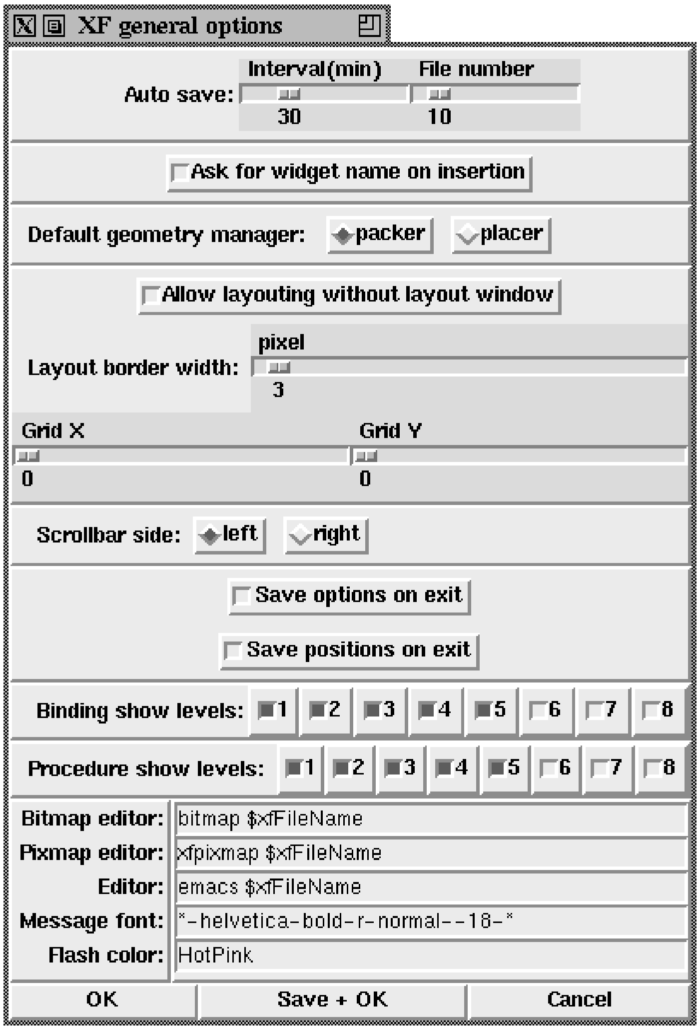


Figure B.27: The procedure XFProcOptionsGeneral

The single options have the following meanings (some option names may be abbreviated):

Option name	Purpose
Auto save	The interval slider specifies the interval between two auto saves. The file number slider specifies the number of backup files to be created. The backup files are created in the temporary directory, and they start with an "as".
Ask for widget name...	This checkbutton activates a dialog box, where the user can enter a widget name before a widget is inserted.
Default geometry manager	Depending on these buttons, new widgets are inserted using the packer or the placer.
Allow layouting without...	If this checkbutton is true direct layouting of the widgets is only allowed when the layout dialog box is popped up (to prevent erroneous geometry changes). Otherwise the layouting is always possible.
Default geometry manager	Depending on these buttons, new widgets are displayed, using the packer or the placer.
Layout border width	With this slider, the sizing border of widgets can be specified. This border is used to size widgets, while the remaining inner area is used to move the widget.
GridX/GridY	With these sliders, a grid can be defined for widgets that are layouted with the placer.
Scrollbar side	Depending on these buttons, scrollbars are displayed left from the controlled widgets or right.
Save options on exit	If this checkbutton is true, the current *XF* options are saved when the program is stopped.
Save positions on exit	If this checkbutton is true, the current *XF* window positions are saved when the program is stopped.
Binding show levels	These checkbuttons specify which levels of bindings are displayed in the binding dialog. The level of a binding is specified by the string "# xf ignore me <level>" at the beginning of the *Tcl/Tk* command.
Procedure show levels	These checkbuttons specify which levels of procedures are displayed in the procedure dialogs. The level of a procedure is specified by the string "# xf ignore me <level>" at the beginning of the *Tcl/Tk* command.
Bitmap editor	This entry contains the command that is invoked to start an external bitmap editor. The editor command must contain the string $xfFileName at the position where the filename which is to be edited should be substituted.
Pixmap editor	This entry contains the command that is invoked to start an external pixmap editor. The editor command must contain the string $xfFileName at the position where the filename which is to be edited should be substituted.
Editor	This entry contains the command that is invoked to start an external editor. The editor command must contain the string $xfFileName at the position where the filename which is to be edited should be substituted.

Option name	Purpose
Message font	This font is used in *XF* message boxes. All other dialogs are using the default font.
Flash color	This color is used to highlight the selected widget.

B.7.3 XFProcOptionsIconBar (Iconbar configuration)

This procedure activates the iconbar configuration. This dialog is explained in the chapter about the templates.

B.7.4 XFProcOptionsInterpreter (Interpreter options)

This dialog box provides access to the interpreter settings that are used by *XF*.

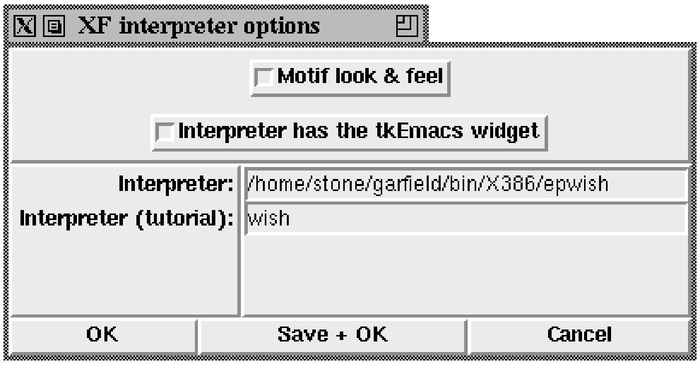


Figure B.28: The procedure XFProcOptionsInterpreter

The single options have the following meanings (some option names may be abbreviated):

Option name	Purpose
Motif look & feel	This checkbutton toggles the global variable tk_strictMotif which is used to make the behavior of *Tk* more motif-like.
Interpreter has tkEmacs	Only when this checkbutton is selected, the tkEmacs widget is used for editing *Tcl/Tk* source. Otherwise, an existing tkEmacs widget is ignored.
Interpreter	This is the name of the interpreter which is inserted at the beginning of the created code to allow the execution of this *Tcl/Tk* code.
Interpreter (tutorial)	This is the name of the interpreter that is used to run the tutorial. Usually, this is the standard wish. To allow the use of special extensions in future versions, this name is adaptable.

B.7.5 XFProcOptionsMenuBar (Menubar configuration)

This procedure activates the menubar configuration. This dialog is explained in the chapter about the templates.

B.7.6 XFProcOptionsPathFile (Path/file names)

This dialog box provides access to the path and file names that *XF* uses.

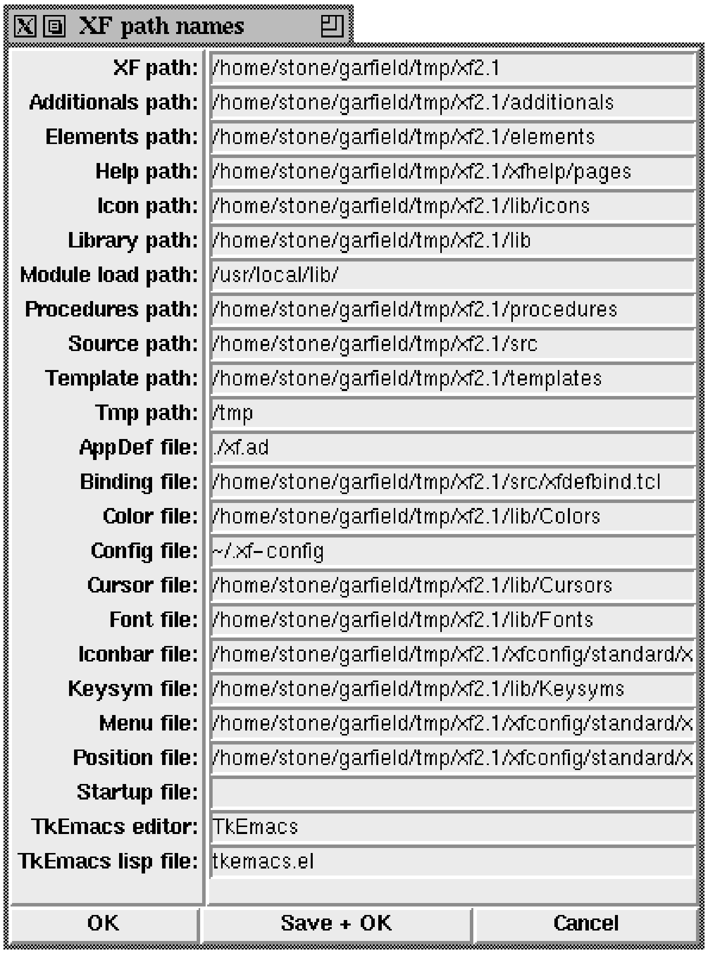


Figure B.29: The procedure XFProcOptionsPathFile

The single options have the following meanings (some option names may be abbreviated):

Option name	Purpose
XF path	This pathname is pointing at the root of the installed *XF* distribution.
Additionals path	This pathname is pointing at the directory where the sources for the support of additional widgets are located.
Elements path	This pathname is pointing at the directory where the sources for the support of the standard *Tk* widgets are located.
Help path	This is a list of pathnames separated by ":" containing the help pages for the help program.

Option name	Purpose
Icon path	This is a list of pathnames separated by ":" containing the icons for the iconbar.
Library path	This pathname points at the directory where the library files of *XF* are located.
Module load path	This is a list of pathnames separated by ":" pointing at directories where *XF* can find modules that should be loaded. If these directories contain tclIndex files, the auto loading facility of *Tcl* also uses this pathname.
Procedures path	This pathname is pointing at the directory where the *Tcl/Tk* procedures can be stored.
Source path	This pathname is pointing at the directory where the *XF* sources are located.
Template path	This is a list of pathnames separated by ":" pointing at directories where templates can be found and stored.
Tmp path	This pathname is pointing at the directory where *XF* can store temporary data. This includes the auto save files.
AppDef file	This filename specifies the application default file that *XF* should load at startup. This file can contain standard X resource specifications
Binding file	This filename specifies the file containing class bindings. These bindings can be changed and saved with *XF*. If the class bindings are significant for the application, they should be included directly in the application source with an option in (`Options` | `Source options`).
Color file	This filename specifies the file containing the colornames for the color selection box. This file is created automatically when *XF* is installed.
Config file	This filename specifies the configuration file for *XF*. This filename can be specified with a commandline option when *XF* is started (-xfconfig).
Cursor file	This filename specifies the file containing the cursornames for the cursor selection box. This file is created automatically when *XF* is installed.
Font file	This filename specifies the file containing the fontnames for the font selection box. This file is created automatically when *XF* is installed.
Iconbar file	This filename specifies the iconbar configuration file.
Keysym file	This filename specifies the file containing the keysymnames for the keysym selection box. This file is created automatically when *XF* is installed.
Menubar file	This filename specifies the menubar configuration file.
Position file	This filename specifies the window position file for *XF*. This file contains the window positions of the *XF* dialog boxes.

Option name	Purpose
Startup file	This filename specifies the startup file. This file is evaluated when *XF* is started. Here, the user can make local extensions to *XF*.
TkEmacs editor	This is the name of the emacs that is called by the tkEmacs widget. Usually, this value is not changed.
TkEmacs lisp file	This is the name of the emacs lisp code that is loaded by the tkEmacs widget. Usually, this value is not changed.

B.7.7 XFProcOptionsSaveClassBindings (Save class bindings)

This procedure saves the currently defined class bindings for the widgets to the bindings file. This file can be specified in the (`Options` | `Path/file names`) dialog.

B.7.8 XFProcOptionsSaveModuleList (Save module list)

This procedure saves the current selection of changeable modules to the local file ".xf-save-modules". Only modules that have been selected in the module structure dialog are saved when the application is saved.

B.7.9 XFProcOptionsSaveOptions (Save options)

This procedure explicitly saves the current options to the options file. This file can be specified in the (`Options` | `Path/Filenames`) dialog. The user can specify that the options should be automatically saved when he leaves *XF*.

B.7.10 XFProcOptionsSavePositions (Save window positions)

This option explicitly saves the current positions and sizes of the *XF* windows to the position file. This file can be specified in the (`Options` | `Path/Filenames`) dialog. The user can specify that the options should be automatically saved when he leaves *XF*.

B.7.11 XFProcOptionsSource (Source options)

This dialog box provides access to the source code generation options.

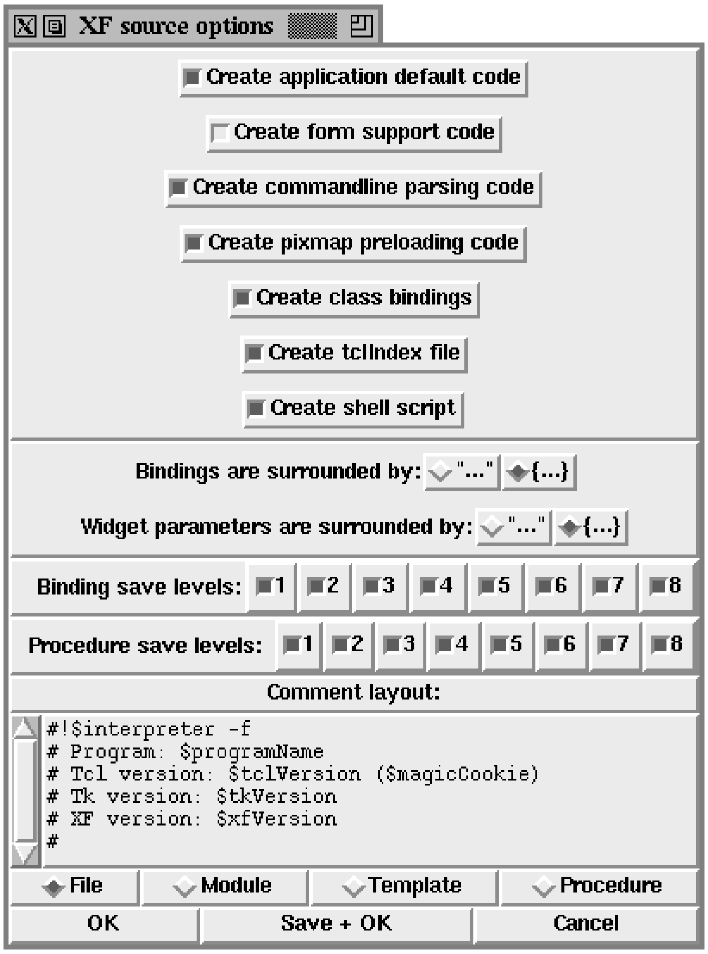


Figure B.30: The procedure XFProcOptionsSource

The single options have the following meanings (some option names may be abbreviated):

Option name	Purpose
Application default code	If this checkbutton is selected, *XF* will create code that allows the parsing of application default files. The code searches in the application default directories for a file matching the application name, and parses it.
Form support code	If this checkbutton is true, *XF* will create code that supports formulars. The code allows the automatic connection of text/entry widgets, and handles the geometry of these widgets.

Option name	Purpose
Commandline parsing code	If this checkbutton is true, *XF* will create code that parses the commandline options for some special *XF* extensions.
Pixmap preloading code	If this checkbutton is true, *XF* will create code that uses the TkPixmap extension *pinfo* to include the bitmaps/pixmaps that are used by the application into the code.
Class bindings	If this checkbutton is true, *XF* will include the class bindings into the created code.
Create tclIndex file	If this checkbutton is true, *XF* will create a tclIndex file for those modules that are specified to be auto loadable.
Create shell script	If this checkbutton is true, *XF* will create a shell script for calling the resulting application.
Bindings are surrounded...	Depending on these buttons, *XF* will enclose the *Tcl/Tk* commands bound to an event in {} or "". Please use {}, for the enclosing in "" may lead into trouble.
Procedures are surrounded...	Depending on these buttons, *XF* will enclose the *Tcl/Tk* commands bound to a resource (like the -command resource for buttons) in {} or "". Please use {}, for the enclosing in "" may lead into trouble.
Binding save levels	These checkbuttons specify which levels of bindings are saved. The level of a binding is specified by the string "# xf ignore me <level>" at the beginning of the *Tcl/Tk* command.
Procedure save levels	These checkbuttons specify which levels of the procedures are saved. The level of a procedure is specified by the string "# xf ignore me <level>" at the beginning of the *Tcl/Tk* command.
Comment layout	Depending on the radiobuttons below the text widget, the text widget allows the changing of comments that are inserted in the code by *XF*. These comments can contain several variables. These are: programName, moduleName, tclVersion, tkVersion, xfVersion, magicCookie and procedureName.

B.7.12 XFProcOptionsVersion (Version control options)

This dialog box provides access to the version control facilities that *XF* uses. It is possible to store and retrieve procedures and modules to/from ShapeTools archives. The commands need some parameters which are provided by *XF* as *Tcl* variables. The variable xfFileName contains the name of the object to be processed. The variable xfFileVersion contains the version number of the object to be processed. The variable xfMessage contains the message to attach to an object when it is saved.

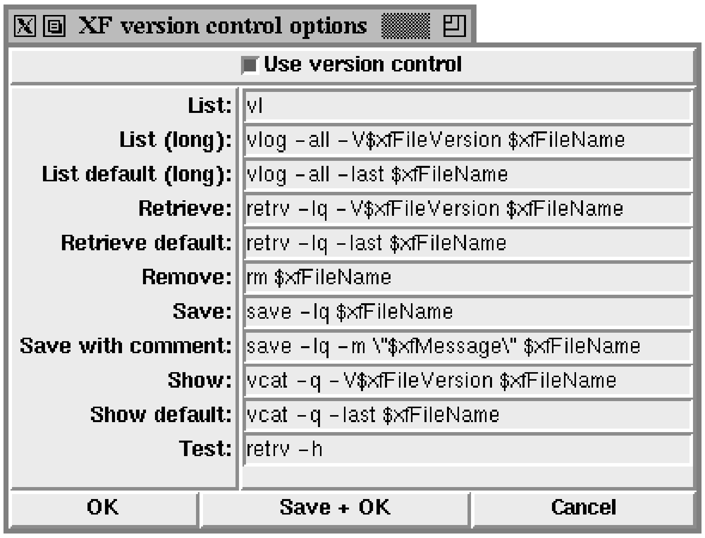


Figure B.31: The procedure XFProcOptionsVersion

The single options have the following meanings (some option names may be abbreviated):

Option name	Purpose
Use version control	This checkbutton allows it to disable the use of the version control system.
List	This command is executed to get a name list of all objects in the version system. Before this command is executed, *XF* changes into the correct directory.
List (long)	This command is executed to get a detailed information on one specific object in the version system. The object is identified with a version number. Before this command is executed, *XF* changes into the correct directory.
List default (long)	This command is executed to get a detailed information on one specific object in the version system. The object is the default object that is used when no explicit version number is given. Before this command is executed, *XF* changes into the correct directory.
Retrieve	This command is executed to retrieve one specific object from the version system. The object is identified with a version number. Before this command is executed, *XF* changes into the correct directory.
Retrieve default	This command is executed to retrieve one specific object from the version system. The object is the default object that is used when no explicit version number is given. Before this command is executed, *XF* changes into the correct directory.

Option name	Purpose
Remove	This command is executed to remove a retrieved object. Before this command is executed, *XF* changes into the correct directory.
Save	This command is executed to save an object into the version system. Before this command is executed, *XF* changes into the correct directory.
Save with comment	This command is executed to save an object into the version system. It also takes a message that is attached to that object. Before this command is executed, *XF* changes into the correct directory.
Show	This command is executed to show the contents of one specific object from the version system. The object is identified with a version number. Before this command is executed, *XF* changes into the correct directory.
Show default	This command is executed to show the contents of one specific object from the version system. The object is the default object that is used when no explicit version number is given. Before this command is executed, *XF* changes into the correct directory.
Test	This command is executed to check if the version control system is installed on the machine.

B.7.13 XFProcOptionsWindow (Window options)

This dialog box provides access to the window handling in *XF*. It is possible to control the appearance of the main window, and the positioning/sizing of the *XF* dialog boxes.

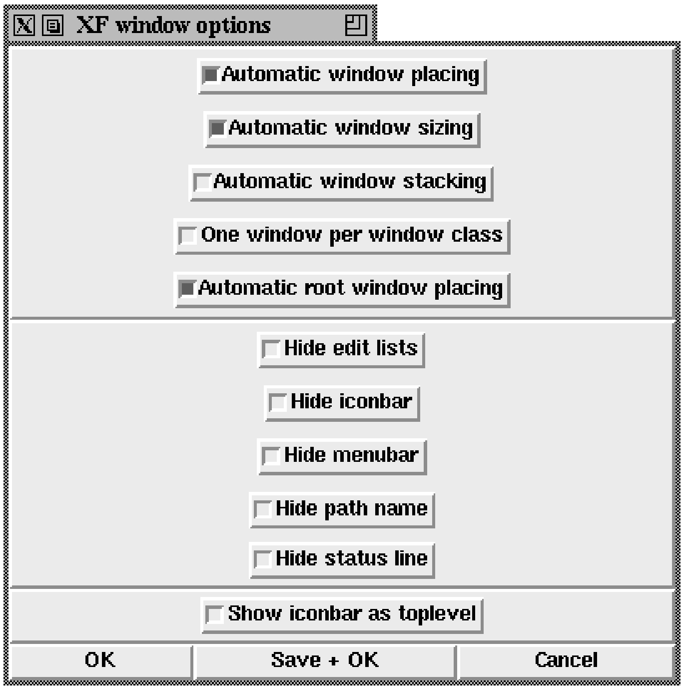


Figure B.32: The procedure XFProcOptionsWindow

The single options have the following meanings (some option names may be abbreviated):

Option name	Purpose
Automatic window placing	This checkbutton toggles the placing policy of *XF*. Automatic placing means, that the position of the dialog boxes is set by *XF* at startup. The changes that the user makes are stored.
Automatic window sizing	This checkbutton toggles the sizing policy of *XF*. Automatic sizing means, that the size of the dialog boxes is set by *XF* at startup. The changes that the user makes are stored.
Automatic window stacking	This checkbutton toggles the placing/sizing policy of *XF*. Automatic stacking means that the size and position of some dialog boxes are set to the size and position of a "leading" window. This can only be done for parameter dialogs.
One window per window class	This checkbutton toggles the dialog box creation policy of *XF*. If only one window per window class is allowed, *XF* will use an already existing toplevel of the same window class to display dialog boxes.

Option name	Purpose
Automatic root window placing	If this checkbutton is true, the main application window is placed to +0+0 on startup.
Hide edit lists	If this checkbutton is true, the main *XF* window does not contain the widget listboxes.
Hide iconbar	If this checkbutton is true, the main *XF* window does not contain the iconbar.
Hide menubar	If this checkbutton is true, the main *XF* window does not contain the menubar.
Hide path name	If this checkbutton is true, the main *XF* window does not contain the current widget path.
Hide status line	If this checkbutton is true, the main *XF* window does not contain the status line.
Show iconbar as toplevel	If this checkbutton is true, the iconbar of the main *XF* window is displayed as a separate toplevel at startup.

B.8 Help

B.8.1 XFProcHelpAbout (About)

This procedure pops up the *XF* about box. It gets no parameters.

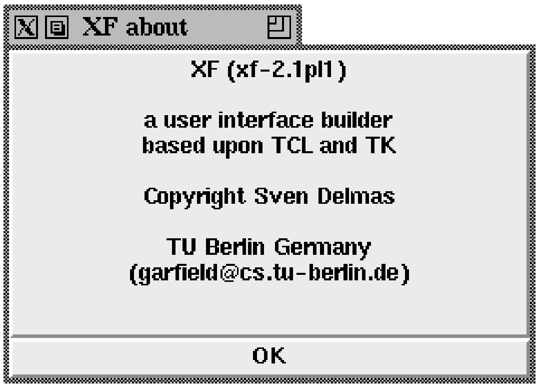


Figure B.33: The procedure XFProcHelpAbout

B.8.2 XFProcHelpHelp

This procedure calls the external program *xfhelp*. The procedure gets a list separated by spaces specifying the help page to be displayed. The help pages are structured in a directory tree. The program provides access to the *Tcl/Tk* manual pages, *XF* help pages and user changeable notes. A description of the program can be found in the appropriate part of this documentation.

B.8.3 XFProcHelpTutorial (Tutorial)

This procedure calls the external program *xftutorial*. The procedure gets no parameters. The tutorial program introduces the user into the usage of *XF*. An interactive example for a session leads him through the basic concepts of *XF*. A description of the program can be found in the appropriate part of this documentation.

Appendix C

Templates

XF provides the concept of *templates*. Templates are files that contain a widget structure and/or procedures. They can be loaded by the user, and add this widget structure and/or functionality to the program.

The *XF* distribution contains three main groups of templates. *Combined* templates contain only a combination of widgets that form a complex widget structure. The second group are *Procedures*. They implement functionality, i.e. a dialog box that can be popped up or general functions to handle lists. The third group of templates (*Widgets*) implements a sort of new widgets. This means that new complex widgets are built basing upon existing widgets.

The following chapter describes the templates that are part of the *XF* distribution. The user can define his own new templates.

C.1 Combined

C.1.1 CanvasLS, CanvasRS

A canvas widget surrounded by two scrollbars. The appropriate commands to enable scrolling are already set. CanvasLS and CanvasRS differ at the side where the vertical scrollbar is displayed.

Figure C.1: The template CanvasLS

C.1.2 EntryL, EntryLLS, EntryLS, EntryS

EntryL implements an entry widget with a label at the left side.

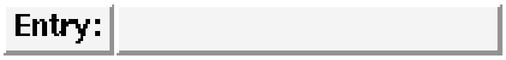


Figure C.2: The template EntryL

EntryLLS implements an entry widget with a label at the left side, and a horizontal scrollbar. The commands for scrolling are set.

Figure C.3: The template EntryLLS

EntryLS implements an entry widget with a label at the left side, and a horizontal scrollbar. The commands for scrolling are set.

Figure C.4: The template EntryLS

EntryS implements an entry widget with a horizontal scrollbar. The commands for scrolling are set.

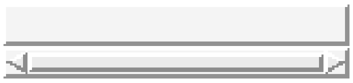


Figure C.5: The template EntryS

C.1.3 HypertextLS, HypertextRS

A hypertext widget surrounded by two scrollbars. The appropriate commands to enable scrolling are already set. HypertextLS and HypertextRS differ at the side where the vertical scrollbar is displayed.

Figure C.6: The template HypertextLS

C.1.4 ListboxLS, ListboxRS

A listbox widget surrounded by two scrollbars. The appropriate commands to enable scrolling are already set. ListboxLS and ListboxRS differ at the side where the vertical scrollbar is displayed.

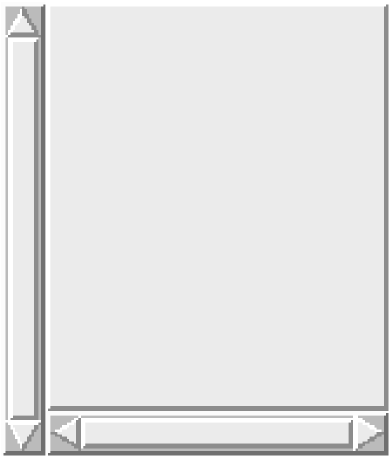


Figure C.7: The template ListboxLS

C.1.5 PhotoLS, PhotoRS

A photo widget surrounded by two scrollbars. The appropriate commands to enable scrolling are already set. PhotoLS and PhotoRS differ at the side where the vertical scrollbar is displayed.

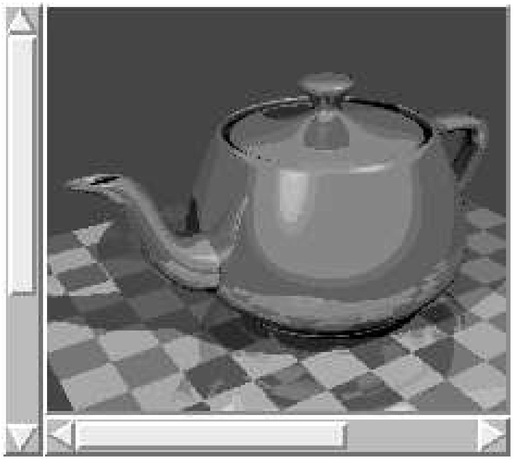


Figure C.8: The template PhotoLS

C.1.6 TextLS, TextRO, TextROLS, TextRORS, TextRS

A text widget with a vertical scrollbar. The appropriate commands to enable scrolling are already set. TextLS and TextRS differ at the side where the vertical scrollbar is displayed. TextRO, TextROLS and TextRORS insert text widgets that are disabled for user input.

Figure C.9: The template TextLS

C.1.7 TkEmacsLS, TkEmacsRS

A tkemacs widget with a vertical and horizontal scrollbar. The appropriate commands to enable scrolling are already set. TkEmacsLS and TkEmacsRS differ at the side where the vertical scrollbar is displayed.

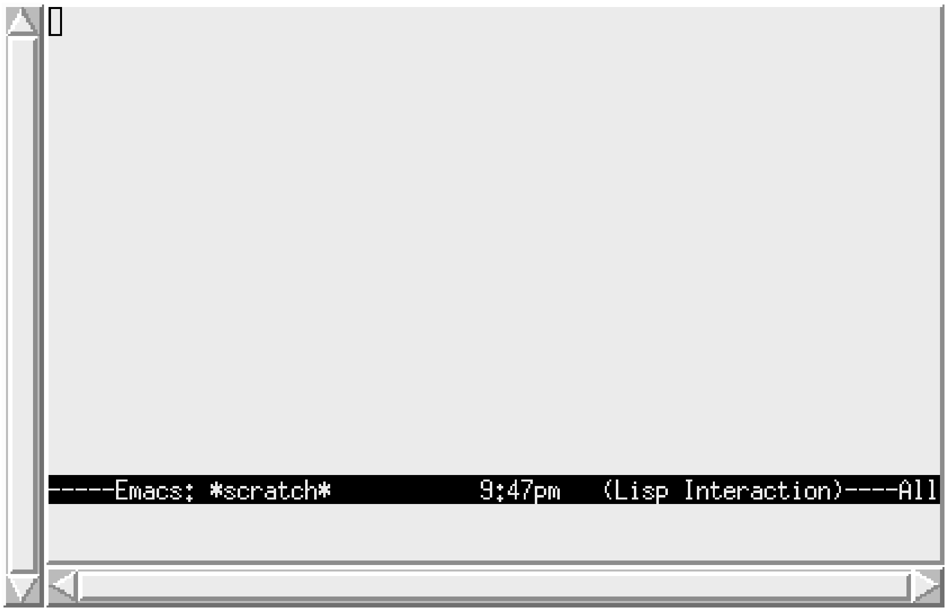


Figure C.10: The template TkEmacsLS

C.2 Procedures

C.2.1 AlertBox, AlertBoxFd, AlertBoxFile

This template defines three new procedures named AlertBox, AlertBoxFd and AlertBoxFile. Calling one of these procedures pops up an alert box. These boxes can be modal or not. If the dialog box is modal, the procedure returns the number of the pressed button. Otherwise the specified command is evaluated. The procedures get the following parameters:

Parameter name	Opt.	Purpose
alertBoxMessage	y	The message, file or file descriptor that is displayed.
alertBoxCommand	y	The command to execute when OK is pressed. The dialog box is not modal (non blocking) when this parameter is not an empty string.
alertBoxGeometry	y	This is the geometry of the dialog box.
alertBoxTitle	y	This is the title bar of the dialog box.
args	y	Any additional parameters are interpreted as a button label. The dialog box is modal (blocking), and the return value of the procedure is the number of the pressed button.

To configure the different aspects of the alert box, there exists a global array named alertBox. A default value of "-" means that the Tk default value is used. This array contains elements that control the alert box (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
after	0	Invokes the first button after n seconds. The dialog box is removed.
anchor	nw	The anchor of the message widget.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
justify	center	The justification of the message widget.
toplevelName	.alertBox	The toplevel name. This variable makes it possible to popup multiple dialog boxes at the same time.

A small example of an invocation may look like this:

```
set result [AlertBox ''This is an alert message!''  ''''\
     200x70 ''Dialog title'' OK Abort Cancel]
```

This would create the following dialog box:

Figure C.11: The template AlertBox

C.2.2 ClearList, ClearText

This template defines two procedures that clear the contents of a list/text widget. The procedures get the following parameters:

Parameter name	Opt.	Purpose
listWidget	n	The list/text widget that should be cleared

C.2.3 ColorBox

This template defines a new procedure named ColorBox. Calling this procedure pops up a dialog box to select a color. Colors can be entered by their name, selected from a list, defined in RGB values or as HSV values. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
colorBoxFileColor	y	The file containing a list of colors.
colorBoxMessage	y	The message to be displayed. If the parameter contains the patterns *foreground* or *background*, the appropriate resource is set in the demo widget, and in the target widget.
colorBoxEntryW	y	This is the entry widget where the selected color is to be inserted.
colorBoxTargetW	y	This is the widget that is configured. If this parameter is specified, the selected color is applied to the widget.

To configure the different aspects of the color box, there exists a global array named colorBox. A default value of "-" means that the Tk default value is used. This array contains elements that control the color box (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
palette	""	A list of color names.
scrollActiveForeground	-	The scrollbar active foreground color.
scrollBackground	-	The scrollbar background color.
scrollForeground	-	The scrollbar foreground color.
scrollSide	right	The side of the scrollbar.

A small example of an invocation may look like this:

```
ColorBox ''/usr/lib/X11/rgb.txt'' ''Background''
```

This would create the following dialog box:

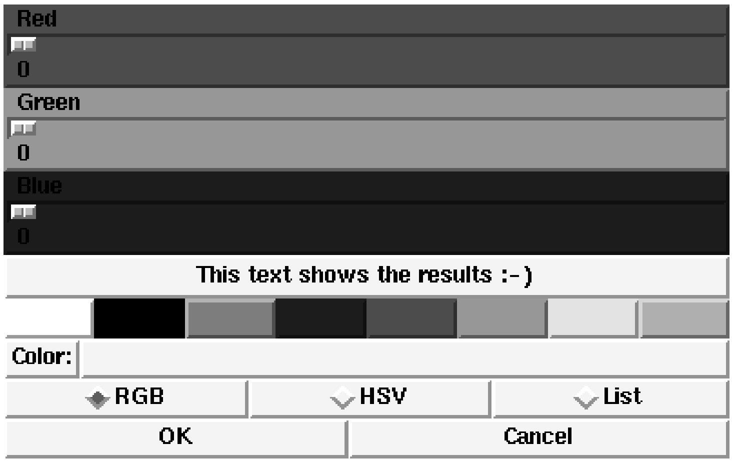


Figure C.12: The template ColorBox

C.2.4 CursorBox

This template defines a new procedure named CursorBox. Calling this procedure pops up a dialog box to select a cursor. The cursor can be selected from a list, or entered directly. The foreground and the background color can be selected. If the template *ColorBox* exists, a double click with the right mouse button activates the color selection box. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
cursorBoxFileCursor	y	The file containing a list of cursors.
cursorBoxFileColor	y	The file containing a list of colors.
cursorBoxMessage	y	The resource name that is configured.
cursorBoxEntryW	y	This is the entry widget where the selected cursor is inserted.
cursorBoxTargetW	y	This is the widget that is configured. If this parameter is specified, the selected cursor is applied to the widget immediately.

To configure the different aspects of the cursor box, there exists a global array named cursorBox. A default value of "-" means that the Tk default value is used. This array contains elements that control the cursor box (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
scrollActiveForeground	-	The scrollbar active foreground color.
scrollBackground	-	The scrollbar background color.
scrollForeground	-	The scrollbar foreground color.
scrollSide	right	The side of the scrollbar.

A small example of an invocation may look like this:

```
CursorBox ''/usr/local/lib/Cursors'' ''/usr/lib/X11/rgb.txt''
```

This would create the following dialog box:

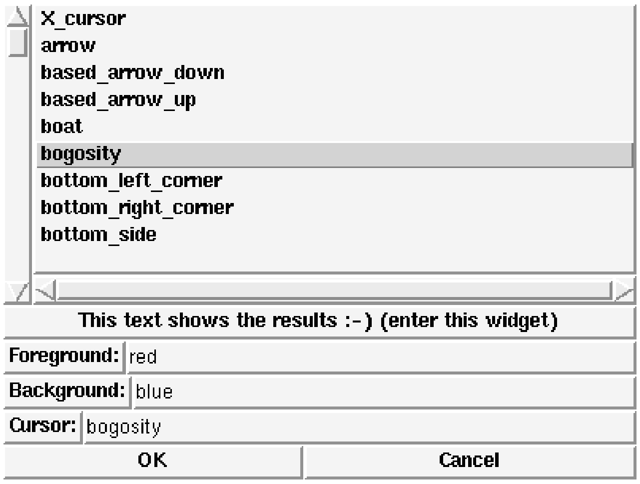


Figure C.13: The template CursorBox

C.2.5 FSBox

This template defines a new procedure named FSBox. Calling this procedure pops up a dialog box to select a file. The dialog box is either modal or non-modal. If the dialog box is modal, the procedure returns the selected file name. Otherwise, the specified Tcl command script is evaluated. Many features support the file selection. There exists a path history, available as a pull down menu (at the label left from the current path name). The label left from the selection pattern contains a pull down menu with all possible extensions. When typing path and file names by hand, the Tab key performs file name completion. The file selector box has a special mode for selecting bitmaps (pixmaps) where the currently selected picture is displayed in a display area. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
fsBoxMessage	y	The message to be displayed.
fsBoxFileName	y	This is a file name that is inserted in the file name selection field, as a default value
fsBoxActionOk	y	This is the Tcl script that is evaluated when the OK button is pressed. To access the selected file and path name, access the global variable fsBox described below. If no commands are specified, the dialog box is modal.
fsBoxActionCancel	y	This is the Tcl script that is evaluated when the Cancel button is pressed. To access the selected file and path name, access the global variable fsBox described below. If no commands are specified, the dialog box is modal.

To configure the different aspects of the file box, there exists a global array named fsBox. A default value of "-" means that the Tk default value is used. This array contains elements that control the file box (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
name	""	The name of the selected file.
path	""	The path name of the selected file.
pattern	""	The display selection pattern.
scrollActiveForeground	-	The scrollbar active foreground color.
scrollBackground	-	The scrollbar background color.
scrollForeground	-	The scrollbar foreground color.
scrollSide	right	The side of the scrollbar.
showPixmaps	0	If this variable is 1, the selected files are interpreted as picture files, and are displayed in an area right from the file list.

A small example of an invocation may look like this:

```
FSBox
```

This would create the following dialog box:

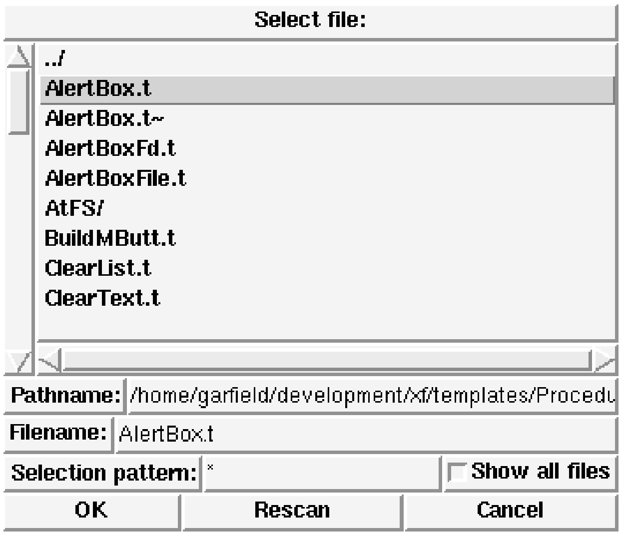


Figure C.14: The template FSBox

C.2.6 FdInList, FileInList, FdInText, FileInText

This template defines four procedures that put the contents of an open file descriptor or a file into a list widget or a text widget. The procedures get the following parameters:

Parameter name	Opt.	Purpose
listWidget	n	The list widget where the file contents are inserted.
fileInFile	y	The filename/filedescriptor that is to be inserted.

C.2.7 FontBox

This template defines a new procedure named FontBox. Calling this procedure pops up a dialog box to select a font. The font can be selected from a list, or the different style parameters can be combined from menus. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
fontBoxFileFont	y	The file containing a list of fonts.
fontBoxMessage	y	The resource name that is configured.
fontBoxEntryW	y	This is the entry widget where the selected font is inserted.
fontBoxTargetW	y	This is the widget that is configured. If this parameters is specified, the selected font is applied to the widget immediately.

To configure the different aspects of the font box, there exists a global array named fontBox. A default value of "-" means that the Tk default value is used. This array contains elements that control the font box (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
scrollActiveForeground	-	The scrollbar active foreground color.
scrollBackground	-	The scrollbar background color.
scrollForeground	-	The scrollbar foreground color.
scrollSide	right	The side of the scrollbar.

A small example of an invocation may look like this:

```
FontBox ''/usr/local/lib/Fonts''
```

This would create the following dialog box:

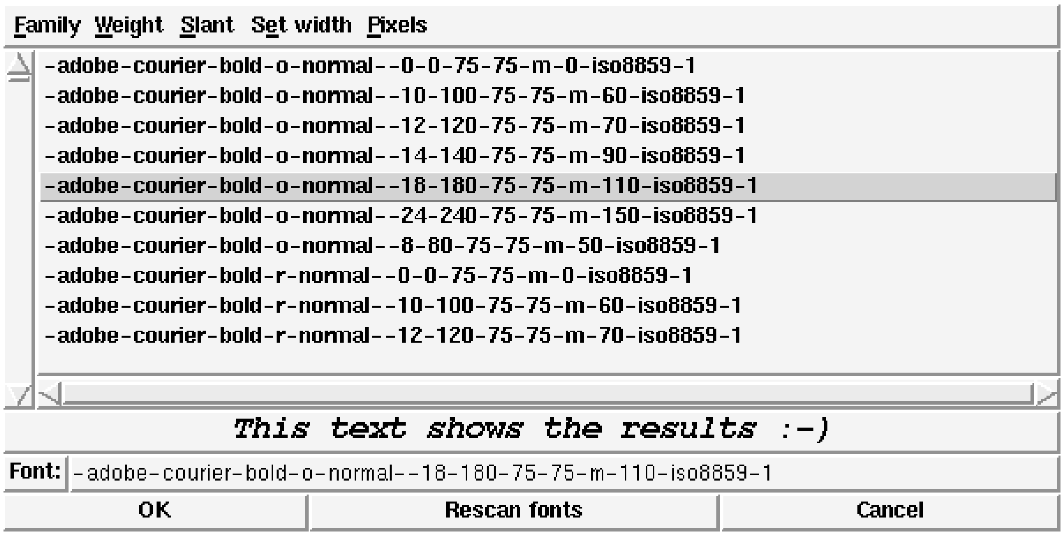


Figure C.15: The template FontBox

C.2.8 IconBar

This template defines a new feature that supports user changeable icon bars. The usage of this feature is more complex than it is for most other templates. An iconbar can be part of your toplevel, or it can use its own toplevel. To switch between both modes, each iconbar has an icon at the right side that toggles between both modes. The icon left from this toggle icon pages through the different iconbar lines. An iconbar contains lines that are separated by a separator. Your code must contain a frame definition. By calling the procedure *IconBarInit*, the iconbar is initialized. The procedure takes the following parameters:

Parameter name	Opt.	Purpose
iconBarUserFile	n	This file contains the user specific iconbar configuration. This file is written when the user presses the `Save` button.
iconBarFile	n	This file contains the fallback iconbar definition. This file is usually global for all users.
iconBarIcons	n	A list of path names separated by ":". In thse path names, the bitmaps for the icon bar can be found.

This will load and initialize the iconbar. To actually display the iconbar you have to call the procedure *IconBarShow*. This procedure creates the iconbar. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
iconBarName	n	The icon bar name. The name identifies a set of icons. An application can contain several iconbars, each under a unique name.
iconBarPath	y	The widget path name where the iconbar is located. If the path name is empty, a toplevel is created.
iconBarStatus	y	The status of the iconbar. An iconbar can have the status "child", which means, that it is inserted to the widget path defined by the previous parameter. The status "toplevel" means that the iconbar is displayed in a separate toplevel.

To remove a displayed iconbar, you can call the procedure *IconBarRemove*. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
iconBarName	n	The icon bar name. The name identifies a set of icons. An application can contain several iconbars, each under a unique name.
iconBarPath	y	The widget path name where the iconbar should be inserted. If the path name is not empty, the children of this widget are destroyed.

To modify an existing iconbar, the procedure *IconBarConf* is called. This pops up a dialog box where bitmaps can be combined with procedure calls, and added to the iconbar. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
iconBarName	n	The icon bar name. The name identifies a set of icons. An application can contain several iconbars, each under a unique name.
iconBarPath	y	The widget path name where the iconbar is located.
iconBarProcs	y	A list of procedure names that can be used in the iconbar. This does not restrict the usage of other procedures, but it gives an impression of the available functionality.

To configure the different aspects of the iconbar, there exists a global array named iconBar. A default value of "-" means that the Tk default value is used. This array contains elements that control the iconbar (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
background	-	The background color.
barBorder	2	The iconbar border width.
barIgnoreSep	0	The iconbar separators are ignored.
barRelief	sunken	The iconbar relief.
font	-	The font.
foreground	-	The foreground color.
iconBorder	2	The icon border width.
iconHeight	20	The icon height.
iconOffset	0	The icon offset.
iconRelief	2	The icon relief.
iconWidth	20	The icon width.
label	""	The label where the description is displayed.
scrollActiveForeground	-	The scrollbar active foreground color.
scrollBackground	-	The scrollbar background color.
scrollForeground	-	The scrollbar foreground color.
scrollSide	right	The side of the scrollbar.

A small example of an initialization may look like this:

```
frame .myIconBar
IconBarInit  /.local-iconbar /usr/local/lib/global-iconbar /usr/local/lib/icons
IconBarShow default .frame
```

This would create the following dialog box:

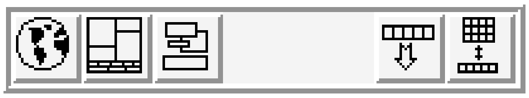


Figure C.16: The template IconBar (initialization)

A small example of an invocation of the configuration may look like this:

```
IconBarConf default .frame {Proc1 Proc2 Proc3}
```

This would create the following dialog box:

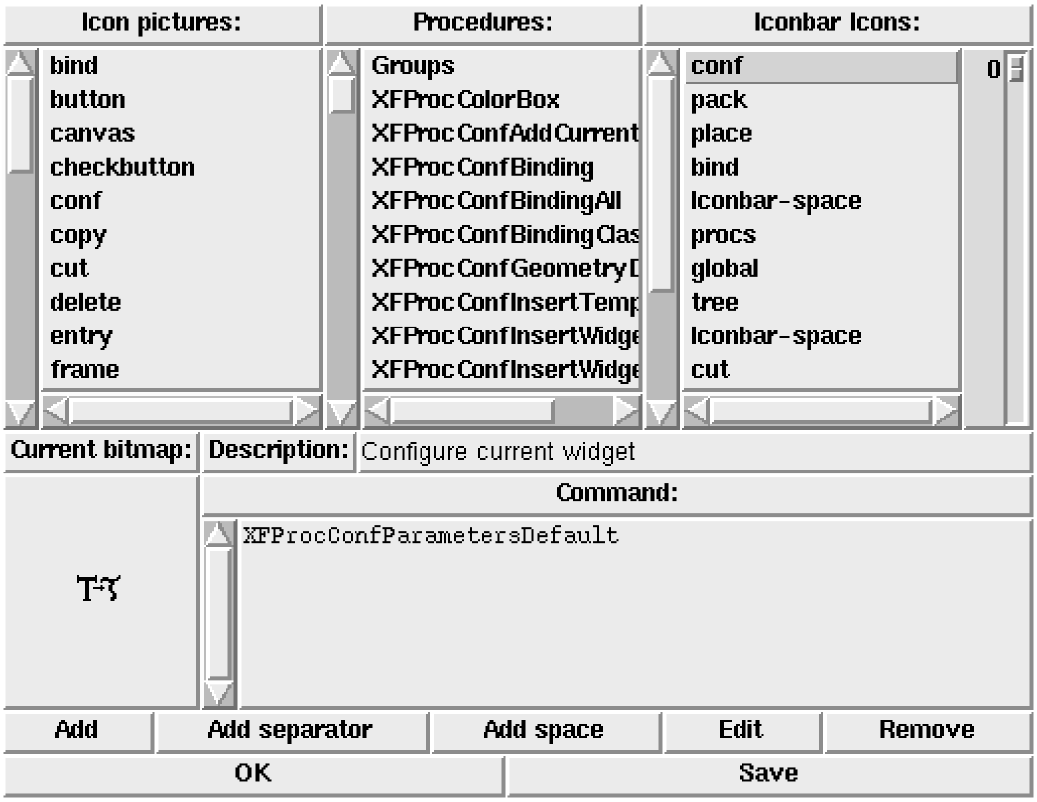


Figure C.17: The template IconBar (configuration)

C.2.9 InputBox

This template defines two new procedures named InputBoxOne and InputBoxMulti. Calling this procedures pops up a dialog box to make textual input. The procedure InputBoxOne allows one line of text, and the procedure InputBoxMulti allows several lines of text. The input boxes can be modal or not. If they are modal, the entered string is returned. Otherwise, the specified command is evaluated. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
inputBoxMessage	y	The message to be displayed.
inputBoxCommandOk	y	The Tcl script that is evaluated when the button named (`OK`) is pressed. To access the inserted text, use the variable inputBox(toplevelName,inputOne) or inputBox(toplevelName,inputMulti). If no commands are specified, the dialog box is modal.
inputBoxCommandCancel	y	The Tcl script that is evaluated when the button named (`Cancel`) is pressed. To access the inserted text, use the variable inputBox(toplevelName,inputOne) or inputBox(toplevelName,inputMulti). If no commands are specified, the dialog box is modal.
inputBoxGeometry	y	The geometry of the toplevel.
inputBoxTitle	y	The title of the toplevel.

To configure the different aspects of the input box, there exists a global array named inputBox. A default value of "-" means that the Tk default value is used. This array contains elements that control the input box (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
anchor	n	The anchor of the message widget.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
justify	center	The justification of the message widget.
scrollActiveForeground	-	The scrollbar active foreground color.
scrollBackground	-	The scrollbar background color.
scrollForeground	-	The scrollbar foreground color.
scrollSide	right	The side of the scrollbar.
toplevelName	.inputBox	The toplevel name. This variable makes it possible to popup multiple dialog boxes at the same time.
toplevelName,inputOne	""	"toplevelName" is replaced by the name of the toplevel. This variable contains the text of the one line input box.
toplevelName,inputMulti	""	"toplevelName" is replaced by the name of the toplevel. This variable contains the text of the multiple line input box.

A small example of an invocation may look like this:

```
InputBoxMulti
```

This would create the following dialog box:

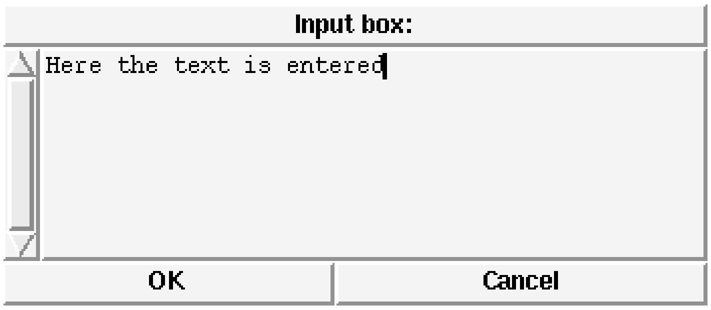


Figure C.18: The template InputBox

C.2.10 IsADir, IsAFile, IsASymlink

This template defines three procedures that check if the passed parameter specifies a valid directory, file or symbolic link. Symbolic links are resolved to the concrete pathname. The procedures get the following parameters:

Parameter name	Opt.	Purpose
pathName	n	The path/file name to check.

C.2.11 KeysymBox

This template defines a new procedure named KeysymBox. Calling this procedure pops up a dialog box to select a keysym. The keysym can be selected from a list, or keypress keysyms can be entered via an example area. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
keysymBoxFileKeysym	y	The file containing a list of keysyms.
keysymBoxMessage	y	The message to display.
keysymBoxEntryW	y	The entry widget where the selected keysym is inserted.

To configure the different aspects of the keysym box, there exists a global array named keysymBox. A default value of “-” means that the Tk default value is used. This array contains elements that control the keysym box (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
overwrite	0	New events are inserted into the entry widget, or overwrite the current event.
scrollActiveForeground	-	The scrollbar active foreground color.
scrollBackground	-	The scrollbar background color.
scrollForeground	-	The scrollbar foreground color.
scrollSide	right	The side of the scrollbar.

A small example of an invocation may look like this:

```
KeysymBox ''/usr/local/lib/Keysyms''
```

This would create the following dialog box:

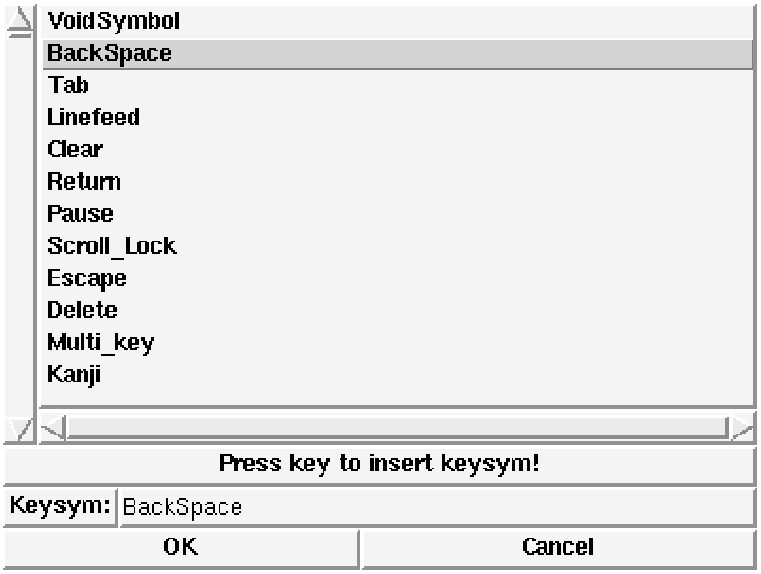


Figure C.19: The template KeysymBox

C.2.12 MakeMButton

This template creates a menubutton, with an automatically created menu attached. The type and contents of the menu are specified by parameters that are passed on to the procedure. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
widgetName	n	The name of the menubutton that is to be created.
buttonLabel	n	The label of the menubutton.
itemType	n	The type of the menu items that are created. Valid types are command, check and radio.
itemList	n	The list of menu item names that are to be created. If itemType is check or radio, and the itemFunctions are empty, these are also the names of the associated variable.
itemFunctions	y	This list contains one or more command/variable names. They are attached to the created menu items.

A small example of an invocation may look like this:

```
MakeMButton .mbutton Optionsmenu radio {Option1 Option2 Option3}
```

This would create the following dialog box:

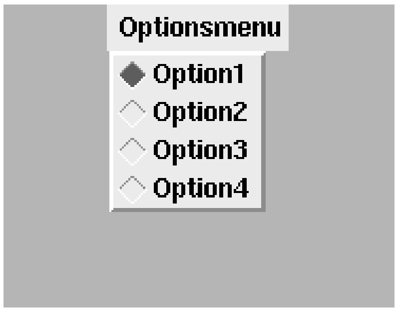


Figure C.20: The template MakeMButton

C.2.13 MenuBar

This template defines a new feature, that supports user changeable menu bars. The usage of this feature is more complex than it is for most other templates. Your code must contain a number of menubutton creations. These menubuttons need not be widget tree siblings, but is only possible to configure one set of sibling menubuttons at a time. After a set of menubuttons has been created, the procedure *MenuBarInit* is called. This procedure takes the following parameters:

Parameter name	Opt.	Purpose
menuBarUserFile	n	This file contains the user-specific menubar configuration. This file is written when the user presses the `Save` button.
menuBarFile	n	This file contains the fallback menubar definition. This file is usually global for all users.

This will load and initialize the menubar. To call the menubar configuration, the procedure *MenuBarConf* is called. This procedure gets the following parameter:

Parameter name	Opt.	Purpose
menuBarConfig	n	The widget path name, containing the menubuttons to be configured.

This pops up a dialog window in which all aspects of the menubar can be modified. Select the menubutton to configure from the upper right list. Setting the label to an empty string hides the menubutton. To configure a menu, the lower right list is used. A new menu is created when a menubutton is inserted that uses this menu. Select the menu to change, and press the `Modify menu` button. When all changes are done, the modified menubar should be saved to the local user-specific file by pressing the `Save` button.

To configure the different aspects of the menubar, there exists a global array named menuBar. A default value of "-" means that the Tk default value is used. This array contains elements that control the menubar (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
scrollActiveForeground	-	The scrollbar active foreground color.
scrollBackground	-	The scrollbar background color.
scrollForeground	-	The scrollbar foreground color.
scrollSide	right	The side of the scrollbar.

A small example of an initialization may look like this:

```
frame .myMenuBar
menubutton .myMenuBar.file -text {File}
menubutton .myMenuBar.misc -text {Misc}
menubutton .myMenuBar.help -text {Help}
MenuBarInit ~/.local-menubar /usr/local/lib/global-menubar
pack append .myMenuBar .myMenuBar.file {left} .myMenuBar.misc {left} .myMenuBar.help {right}
pack append .  .myMenuBar {top fill}
```

This would create the following dialog box:

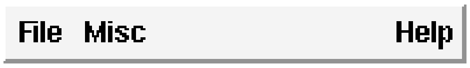


Figure C.21: The template MenuBar (initialization)

A small example of an invocation of the configuration may look like this:

```
MenuBarConf .menuBar
```

This would create the following dialog box:

Figure C.22: The template MenuBar (configuration)

C.2.14 ReadBox

This template defines a new procedure named ReadBox. Calling this procedure pops up a dialog box to enter and evaluate *Tcl/Tk* commands. If the template *AlertBox* is also included, error messages are displayed in a alert box. The procedure gets no parameters.

To configure the different aspects of the read box, there exists a global array named readBox. A default value of "-" means that the Tk default value is used. This array contains elements that control the read box (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
scrollActiveForeground	-	The scrollbar active foreground color.
scrollBackground	-	The scrollbar background color.
scrollForeground	-	The scrollbar foreground color.
scrollSide	right	The side of the scrollbar.

A small example of an invocation may look like this:

```
ReadBox
```

This would create the following dialog box:

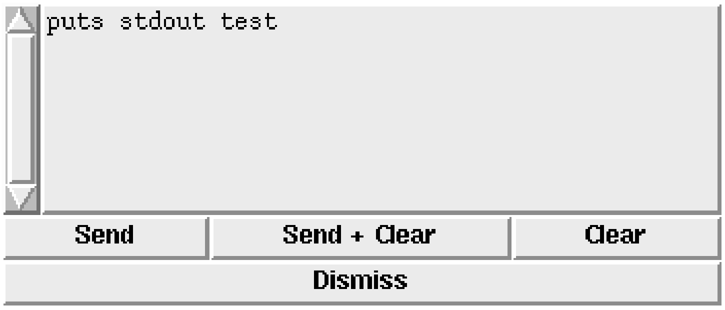


Figure C.23: The template ReadBox

C.2.15 TextBox, TextBoxFd, TextBoxFile

This template defines three new procedures named TextBox, TextBoxFd and TextBoxFile. Calling this procedures pops up a dialog box to display several lines of text in a text widget. These boxes can be modal or not. If the dialog box is modal, the procedure returns the number of the pressed button. Otherwise the specified command is evaluated. The displayed text, a filename or an open file descriptor can be passed on. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
textBoxMessage	y	The message, file or file descriptor that is displayed.
textBoxCommand	y	The command to be executed when OK is pressed. The dialog box is not modal (non blocking) when this parameter is not an empty string.
textBoxGeometry	y	This is the geometry of the dialog box.
textBoxTitle	y	This is the title bar of the dialog box.
args	y	Any additional parameters are interpreted as a button label. The dialog box is modal (blocking). The return value of TextBox is the number of the pressed button.

To configure the different aspects of the text box, there exists a global array named textBox. A default value of "-" means that the Tk default value is used. This array contains elements that control the text box (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
scrollActiveForeground	-	The scrollbar active foreground color.
scrollBackground	-	The scrollbar background color.
scrollForeground	-	The scrollbar foreground color.
scrollSide	right	The side of the scrollbar.
state	disabled	The state of the text widget. Disabled means, that no input from the user is allowed. Normal means that the user can type text.
toplevelName	.textBox	The toplevel name. This variable makes it possible to popup several dialog boxes at the same time.

A small example of an invocation may look like this:

```
TextBox ''Text message''
```

This would create the following dialog box:

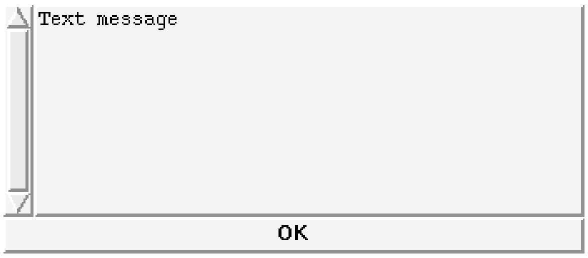


Figure C.24: The template TextBox

C.2.16 YesNoBox

This template defines a new procedure named YesNoBox. Calling this procedure pops up a dialog box with a question to be answered with yes or no. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
yesNoBoxMessage	y	The message to be displayed.
yesNoBoxGeometry	y	The geometry of the yes/no box.

To configure the different aspects of the yes/no box, there exists a global array named yesNoBox. A default value of "-" means that the Tk default value is used. This array contains elements that control the yes/no box (color, font etc.):

Array element	Default	Purpose
activeBackground	-	The active background color.
activeForeground	-	The active foreground color.
afterNo	0	Invokes the no button after n seconds. The dialog box is removed.
afterYes	0	Invokes the yes button after n seconds. The dialog box is removed.
anchor	nw	The anchor of the message widget.
background	-	The background color.
font	-	The font.
foreground	-	The foreground color.
justify	center	The justification of the message widget.

A small example of an invocation may look like this:

```
global yesNoBox
set yesNoBox(font) *times*24*
if {[YesNoBox ''Yes/no message''] == 0} {
     puts stdout ''yes''
}
```

This would create the following dialog box:

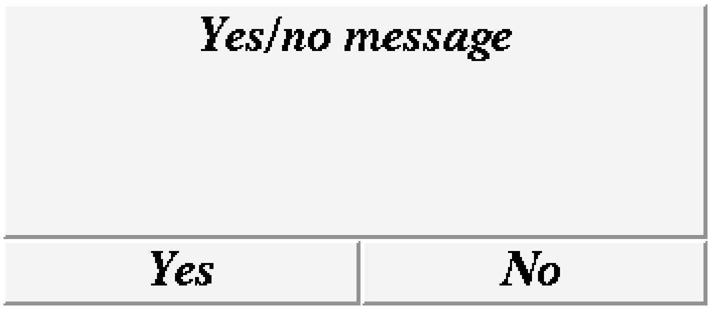


Figure C.25: The template YesNoBox

C.2.17 fileselect

This template defines a new procedure named fileselect. Calling this procedure pops up a dialog box to select a file. When the (`OK`) button is pressed, the procedure passed on as first parameter is evaluated. This procedure gets the filename as parameter. The procedure gets the following parameters:

Parameter name	Opt.	Purpose
cmd	y	This command is evaluated when the OK button is pressed
purpose	y	This is the message of the file selector box
w	y	This is the toplevel path name

A small example of an invocation may look like this:

```
fileselect
```

This would create the following dialog box:

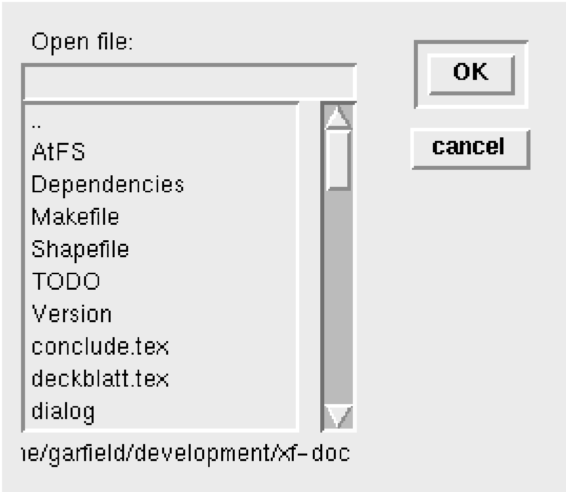


Figure C.26: The template fileselect

C.3 Widgets

C.3.1 MListbox

A listbox arranged in a way that looks a little bit like $Motif^{TM}$. The appropriate commands for scrolling are already set.

Figure C.27: The template MListbox

C.3.2 Menubar

A frame widget containing two menubuttons. One is named File, and is aligned to the left side. The second button is named Help, and is aligned to the right side.

Figure C.28: The template Menubar

C.3.3 OptionButtonE, OptionButtonL

This is a labeled entry/label that displays a value that can be changed by selecting the pull down menu bound to the button right from the entry/label.

Figure C.29: The template OptionButtonE

C.3.4 Popup1, Popup2, Popup3, PopupC-1, PopupM-1, PopupS-1

These templates create popup menus that are activated by the event that identifies the name of the template.

Figure C.30: The template Popup1

List of Figures

2.1 BYO design 12
2.2 XF design 13

3.1 Placed widgets 17
3.2 Packed widgets 18

4.1 The screen after the start of XF 21
4.2 A standard widget parameter box 23
4.3 The general widget parameter box 24
4.4 The widget parameter box for groups 24
4.5 The procedure dialog 25
4.6 The module structure dialog 27

5.1 XF startup 29

A.1 The edge program 37
A.2 The xfappdef program 40
A.3 The xfhardcopy program 41
A.4 The xfhelp program 42
A.5 The xfpixmap program 44
A.6 The xftutorial program 45

B.1 The procedure XFProcMain 47
B.2 The procedure XFProcFileEnterTCL 48
B.3 The procedure XFProcFileLoad 49
B.4 The procedure XFProcFileNew 49
B.5 The procedure XFProcFileQuit 50
B.6 The procedure XFProcConfBinding 51
B.7 The procedure XFProcConfLayout 53
B.8 The procedure XFProcConfPacking 54
B.9 The procedure XFProcConfParametersGeneral 55
B.10 The procedure XFProcConfParametersGroups 56
B.11 The procedure XFProcConfPlacing 57
B.12 The procedure XFProcEditShowCut (tree) 59
B.13 The procedure XFProcEditShowCut (script) 60
B.14 The procedure XFProcProgCommands 61
B.15 The procedure XFProcProgCommands (saving) 62
B.16 The procedure XFProcProgCommands (loading) 62
B.17 The procedure XFProcProgEditScript 63
B.18 The procedure XFProcProgErros 63
B.19 The procedure XFProcProgGlobals 64
B.20 The procedure XFProcProgProcs 65
B.21 The procedure XFProcProgShowScript 66

B.22 The procedure XFProcProgWidgetTree . . . 67
B.23 The procedure XFProcMiscAliases . . . 68
B.24 The procedure XFProcMiscModules . . . 69
B.25 The procedure XFProcMiscPixmaps . . . 70
B.26 The procedure XFProcOptionsBindings . . . 71
B.27 The procedure XFProcOptionsGeneral . . . 72
B.28 The procedure XFProcOptionsInterpreter . . . 74
B.29 The procedure XFProcOptionsPathFile . . . 75
B.30 The procedure XFProcOptionsSource . . . 78
B.31 The procedure XFProcOptionsVersion . . . 80
B.32 The procedure XFProcOptionsWindow . . . 82
B.33 The procedure XFProcHelpAbout . . . 84

C.1 The template CanvasLS . . . 86
C.2 The template EntryL . . . 87
C.3 The template EntryLLS . . . 87
C.4 The template EntryLS . . . 87
C.5 The template EntryS . . . 87
C.6 The template HypertextLS . . . 88
C.7 The template ListboxLS . . . 89
C.8 The template PhotoLS . . . 90
C.9 The template TextLS . . . 91
C.10 The template TkEmacsLS . . . 92
C.11 The template AlertBox . . . 93
C.12 The template ColorBox . . . 95
C.13 The template CursorBox . . . 96
C.14 The template FSBox . . . 98
C.15 The template FontBox . . . 100
C.16 The template IconBar (initialization) . . . 102
C.17 The template IconBar (configuration) . . . 103
C.18 The template InputBox . . . 105
C.19 The template KeysymBox . . . 107
C.20 The template MakeMButton . . . 108
C.21 The template MenuBar (initialization) . . . 110
C.22 The template MenuBar (configuration) . . . 110
C.23 The template ReadBox . . . 111
C.24 The template TextBox . . . 112
C.25 The template YesNoBox . . . 113
C.26 The template fileselect . . . 114
C.27 The template MListbox . . . 115
C.28 The template Menubar . . . 116
C.29 The template OptionButtonE . . . 117
C.30 The template Popup1 . . . 118

Bibliography

[1] Michel Beaudouin-Lafon. User Interface Support for the Integration of Software Tools: an Iconic Model of Interaction. *Sigplan Notices*, 24(2):143–152, November 1988.

[2] Michel Beaudouin-Lafon and Solange Karsenty. Iconic Shells for Multitasking Workstations. In *Proceedings of the ACM Symposium on Small Systems*. ACM Press, May 1988.

[3] Martin R. Cagan. The HP SoftBench Environment Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.

[4] D. Eckardt, W. Huebner, and G. Lux-Muelders. Konzeption der STONE-Benutzungsoberflaeche THESEUS++. STONE Technical Report ZGDV.006.1, Zentrum fuer Graphische Datenverarbeitung, Darmstadt, Germany, December 1989.

[5] Brian D. Fromme. HP Encapsulator: Bridging the Generation Gap. *Hewlett-Packard Journal*, 41(3):59–68, June 1990.

[6] Jonathan Grudin. The Case Against User Interface Consistency. *Communications of the ACM*, 32(1):1164–1173, October 1989.

[7] Arnaud Le Hors. XPM Manual. Technical report, Groupe Bull, Bull Research c/o INRIA, 1993.

[8] Andreas Lampen. Advancing Files to Attributed Software Objects. In *Proceedings of the Winter 1991 USENIX Conference*, pages 219–229, Berkeley (CA), USA, January 1991. USENIX Association.

[9] Andreas Lampen and Axel Mahler. An Object Base for Attributed Software Objects. In *Proceedings of the Autumn 1988 EUUG Conference*, pages 95–106, Lisbon, Portugal, October 1988. European Unix systems User Group.

[10] Mark A. Linton, Paul R. Calder, and John M. Vlissides. InterViews: A C++ Graphical Interface Toolkit. Technical report, Stanford University, Stanford (CA), USA, 1988.

[11] Axel Mahler. Organizing Tools in a Uniform Environment Framework. In *Proceedings of the Winter 91 USENIX Conference*, pages 231–242, Dallas (TX), USA, January 1991. USENIX Association.

[12] Axel Mahler and Andreas Lampen. An Integrated Toolset for Engineering Software Configurations. *SIGPLAN Software Engineering Notes*, 13(5):191–200, November 1988.

[13] Axel Mahler and Andreas Lampen. Integrating Configuration Management into a Generic Environment. In *Symposium on Practical Software Development Environments*, pages 229–237, Irvine (CA), USA, December 1990. ACM Press.

[14] F. Newbery-Paulisch and W. F. Tichy. EDGE: An Extendible Graph Editor. *Software-Practice and Experience*, 20, June 1990.

[15] John Ousterhout. The Tcl/Tk Book. unpublished draft.

[16] John Ousterhout. Tcl: An Embeddable Command Language. In *Proceedings of the Winter 1990 USENIX Conference*, pages 133–146, Berkeley (CA), USA, January 1990. USENIX Association.

[17] John Ousterhout. An X11 Toolkit Based on the Tcl Language. In *Proceedings of the Winter 1991 USENIX Conference*, pages 105–115, Berkeley (CA), USA, January 1991. USENIX Association.

[18] John Ousterhout. The Tcl Language and the Tk Toolkit. Tutorial, 7th Annual X Technical Conference, Boston (MA), USA, January 1993.

[19] Robert Scheifler and James Gettys. *X Window System (Second Edition)*. Digital Press, 1990.

[20] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley Publishing Company, reading, mass. edition, 1987.

[21] John M. Vlissides, Steve Tang, and Charles Brauer. Ibuild User's Guide. Technical report, Stanford University, Stanford (CA), USA, 1992.

Index

Configuration menu, 51–57
- XFProcConfAddCurrentItem, 51
- XFProcConfBinding, 51
- XFProcConfBindingAll, 52
- XFProcConfBindingClass, 52
- XFProcConfConfigure, 52
- XFProcConfGeometryDefault, 52
- XFProcConfInsertTemplate, 52
- XFProcConfInsertWidgetConfig, 52
- XFProcConfInsertWidgetDefault, 52
- XFProcConfLayout, 53
- XFProcConfPacking, 54
- XFProcConfParametersDefault, 54
- XFProcConfParametersGeneral, 55
- XFProcConfParametersGroups, 55
- XFProcConfParametersSmall, 56
- XFProcConfParametersSpecial, 56
- XFProcConfPlacing, 56

Edit menu, 58–60
- XFProcEditClearCut, 58
- XFProcEditCopy, 58
- XFProcEditCut, 58
- XFProcEditDelete, 58
- XFProcEditLoadCut, 58
- XFProcEditLoadTemplate, 58
- XFProcEditMakeAProc, 58
- XFProcEditPaste, 59
- XFProcEditSaveCut, 59
- XFProcEditSaveCutAsTemplate, 59
- XFProcEditShowCut, 59

External tools, 36–45
- Edge, 37
- ShapeTools, 39
- xfappdef, 40
- xfhardcopy, 41
- xfhelp, 42
- xfpixmap, 44
- xftutorial, 45

File menu, 48–50
- XFProcFileEnterTCL, 48
- XFProcFileInsert, 48
- XFProcFileLoad, 49
- XFProcFileNew, 49
- XFProcFileQuit, 50
- XFProcFileSave, 50
- XFProcFileSaveAs, 50

Help menu, 84
- XFProcHelpAbout, 84
- XFProcHelpHelp, 84
- XFProcHelpTutorial, 84

Interface builders, 9–11
- BYO, 11
- GINA, 11
- HP Interface Architect, 10
- Ibuild, 11
- NeXT Interface-Builder, 10
- SUIT, 11
- VUIT, 10

Main XF window, 47
- XFProcMain, 47

Misc menu, 68–70
- XFProcMiscAliases, 68
- XFProcMiscAppDefaults, 68
- XFProcMiscEdge, 68
- XFProcMiscHardcopy, 68
- XFProcMiscModules, 69
- XFProcMiscPixmaps, 70
- XFProcMiscSaveEdge, 70
- XFProcMiscTestProgram, 70

Options menu, 71–83
- XFProcOptionsBindings, 71
- XFProcOptionsGeneral, 72
- XFProcOptionsIconBar, 74
- XFProcOptionsInterpreter, 74
- XFProcOptionsMenuBar, 74
- XFProcOptionsPathFile, 75
- XFProcOptionsSaveClassBindings, 77
- XFProcOptionsSaveModuleList, 77
- XFProcOptionsSaveOptions, 77
- XFProcOptionsSavePositions, 77
- XFProcOptionsSource, 78
- XFProcOptionsVersion, 79
- XFProcOptionsWindow, 82

packer, 22

placer, 22
procedures, 25
  EndSrc, 25
  StartupSrc, 25
  XFExternalInitProc, 33
  XFProc..., 33
Programming menu, 61–67
  XFProcProgCommands, 61
  XFProcProgEditScript, 63
  XFProcProgEndSrc, 63
  XFProcProgErrors, 63
  XFProcProgGlobals, 64
  XFProcProgProcs, 65
  XFProcProgShowScript, 66
  XFProcProgStartupSrc, 66
  XFProcProgWidgetTree, 67

Tcl, 14–15
  commands, 15
  datatypes, 14
  quoting, 15
  syntax, 14
  variables, 14
Templates, 25, 85–118
  Combined, 86–92
    CanvasLS, 86
    CanvasRS, 86
    HypertextLS, 88
    HypertextRS, 88
    ListboxLS, 89
    ListboxRS, 89
    PhotoLS, 90
    PhotoRS, 90
    TextLS, 91
    TextRO, 91
    TextROLS, 91
    TextRORS, 91
    TextRS, 91
    TkEmacsLS, 92
    TkEmacsRS, 92
  Procedures, 93–114
    AlertBox, 93
    AlertBoxFd, 93
    AlertBoxFile, 93
    ClearList, 94
    ClearText, 94
    ColorBox, 95
    CursorBox, 96
    EntryL, 87
    EntryLLS, 87
    EntryLS, 87
    EntryS, 87
    FdInList, 99
    FdInText, 99
    FileInList, 99
    FileInText, 99
    fileselect, 114
    FontBox, 100
    FSBox, 97
    IconBar, 101
    IconBarConf, 101
    IconBarInit, 101
    IconBarRemove, 101
    IconBarShow, 101
    InputBoxMulti, 104
    InputBoxOne, 104
    IsADir, 106
    IsAFile, 106
    IsASymlink, 106
    KeysymBox, 107
    MakeMButton, 108
    MenuBar, 109
    MenuBarConf, 109
    MenuBarInit, 109
    ReadBox, 111
    TextBox, 112
    TextBoxFd, 112
    TextBoxFile, 112
    YesNoBox, 113
  Widgets, 115–118
    Menubar, 116
    MListbox, 115
    OptionButtonE, 117
    OptionButtonL, 117
    Popup1, 118
    Popup2, 118
    Popup3, 118
    PopupC-1, 118
    PopupM-1, 118
    PopupS-1, 118
Tk, 15–19
  configuring widgets, 16
  creating widgets, 15
  geometry handling (packer), 17
  geometry handling (placer), 16
  other Tk commands, 18
  widget classes, 15
  widget commands, 16
  widget path, 15
  widget tree, 15
  XF relevant commands, 18

widget
  config, 22
  current widget, 21
  current widget path, 21
  insert, 21
  layout, 22

XF
- alias, 30
- developing in groups, 26
- final product, 27
- levels for bindings, 27
- levels for procedures, 27
- Pitfalls, 32
- restrictions, 31
- source modules, 26
- startup file, 33
- supporting new widgets, 32
- symbolic name, 30
- toplevel windows, 26

XF main window, 20
XFExternalInitProc, 33
XFProcConfAddCurrentItem, 51
XFProcConfBinding, 51
XFProcConfBindingAll, 52
XFProcConfBindingClass, 52
XFProcConfConfigure, 52
XFProcConfGeometryDefault, 52
XFProcConfInsertTemplate, 52
XFProcConfInsertWidgetConfig, 52
XFProcConfInsertWidgetDefault, 52
XFProcConfLayout, 53
XFProcConfPacking, 54
XFProcConfParametersDefault, 54
XFProcConfParametersGeneral, 55
XFProcConfParametersGroups, 55
XFProcConfParametersSmall, 56
XFProcConfParametersSpecial, 56
XFProcConfPlacing, 56
XFProcEditClearCut, 58
XFProcEditCopy, 58
XFProcEditCut, 58
XFProcEditDelete, 58
XFProcEditLoadCut, 58
XFProcEditLoadTemplate, 58
XFProcEditMakeAProc, 58
XFProcEditPaste, 59
XFProcEditSaveCut, 59
XFProcEditSaveCutAsTemplate, 59
XFProcEditShowCut, 59
XFProcFileEnterTCL, 48
XFProcFileInsert, 48
XFProcFileLoad, 49
XFProcFileNew, 49
XFProcFileQuit, 50
XFProcFileSave, 50
XFProcFileSaveAs, 50
XFProcHelpAbout, 84
XFProcHelpHelp, 84
XFProcHelpTutorial, 84
XFProcMiscAliases, 68
XFProcMiscAppDefaults, 68
XFProcMiscEdge, 68
XFProcMiscHardcopy, 68
XFProcMiscModules, 69
XFProcMiscPixmaps, 70
XFProcMiscSaveEdge, 70
XFProcMiscTestProgram, 70
XFProcOptionsBindings, 71
XFProcOptionsGeneral, 72
XFProcOptionsIconBar, 74
XFProcOptionsInterpreter, 74
XFProcOptionsMenuBar, 74
XFProcOptionsPathFile, 75
XFProcOptionsSaveClassBindings, 77
XFProcOptionsSaveModuleList, 77
XFProcOptionsSaveOptions, 77
XFProcOptionsSavePositions, 77
XFProcOptionsSource, 78
XFProcOptionsVersion, 79
XFProcOptionsWindow, 82
XFProcProgCommands, 61
XFProcProgEditScript, 63
XFProcProgEndSrc, 63
XFProcProgErrors, 63
XFProcProgGlobals, 64
XFProcProgProcs, 65
XFProcProgShowScript, 66
XFProcProgStartupSrc, 66
XFProcProgWidgetTree, 67