OS/2 Objective C Libraries

Version 0.8

Application Programming Tools

February 1996

Thomas Baier
baier@ci.tuwien.ac.at

Abstract

This manual is the user's manual for the tools shipped with the Objective C PM and database class libraries. Here you can find information on the utility programs designed to support the application programmer in writing OS/2 PM programs and programs accessing DBase III database files.

If you are searching for specific information concerning

- *Installation* $\cdots$ Read the Installation Manual.
- *Basics of Application development* $\cdots$ Read the appropriate sections in the Tutorial. There you can find a gentle introduction into using this library package for developing OS/2 PM applications.
- *Classes and Methods provided by the library* $\cdots$ You can find special information about the provided classes and methods in the Reference Manual.
- A reference on the supported *Actions*, *Commands* and *Outlets* $\cdots$ see Appendix A on page 41 in this manual.
- *The Database Editor Utility*, the *Project Editor* or the *Interface Editor* $\cdots$ This information can be found in this manual.
- *Literature* $\cdots$ Look in the Literature section of the Tutorial.

Contents

List of Figures

Chapter 1

Introduction

In addition to the libraries for OS/2 PM and database programming, some tools are required to fully utilize these packages or to simplify application development.

This manual is an attempt to document the programs shipped together with the library package. These programs are *Project Editor*, *Interface Editor*, *Database Editor* and two auxiliary programs called *Console* and *DBView*.

To simplify the development process at this time, I recommend, you get some simple tools as listed below:

1. Some kind of `make` program. I recommend using *GNU make*. This program is free and quite powerful. You can get it on `hobbes.nmsu.edu` and many other locations in the Internet where OS/2 software is stored. This program can also be found on the Hobbes CD produced by Walnut Creek.

   A program called `make` is assumed to exist in some directory listed in your `PATH` statement. Project Editor will automatically start it when compiling and linking your applications.

2. A good programmer's editor. My personal preference is *GNU Emacs*. It can be found in the same places as shown above for *GNU make*. A special mode for editing Objective C programs—for editing interface files as well as for editing implementation files—is provided in `\usr\lib\site-lisp`.

3. If you are not content with the features the Interface Editor application provides, you should use a "normal" dialog editor. To simply create dialog windows by drag and drop, you should use a dialog editor which is capable of writing resource definition files (`.rc`) or binary resource files (`.res`). A quite good dialog editor can be found as part of the OS/2 developer's toolkit (IBM). Another possibility is to use *Guidelines*. When using `Guidelines`, you have to patch the resource definition files manually to use standard window classes (instead of `WC_GUI*` just use `WC_*`).

   Always keep in mind, that—although a "normal" dialog editor might provide more well-suited for creating your dialog objects at the moment, the Interface Editor program does provide some additional features as e.g. creating instances of your custom classes and connecting the objects. The Interface Editor will be extended in the future to be a full-featured dialog editor by its own.

4. A compiler capable of creating on-line help files (`.hlp`) and online books (`.inf`). Such a program is also part of the OS/2 developer's toolkit from IBM. This one converts files written in a special tag language (`ipf`) to online help or book files.

   You will only need such a program if you plan to provide on-line help with your programs or if you want to ship documentation in a format readable by the OS/2 program *View* (`.inf`).

1.1 Current Status of this Manual

You will already have noticed, that this manual is a bit small for a visual development suite as shipped together with the Objective C library package.

CAUTION!

This manual is currently "under construction". I will try to describe the development tools at least briefly, but do not expect every feature—or in some cases "bug"—to be documented clearly.

In most cases, the programs should be self-explanatory, and I will try to document most non-trivial program-functions here.

For an introduction into using the development tools, see the *Tutorial* document. There creating an application using the visual development tools *Project Editor* and *Interface Editor* is shown.

For now this documentation must be sufficient. If you have any questions, feel free to cantact me via E-Mail.

Chapter 2

Project Editor—Managing the Files in Your Project

The core of the development tools shipped together with the OS/2 Objective library package is a program called `Project Editor`. This application is used to manage the files your project consists of, e.g. the header files, your classes' implementation files and the make file.

This chapter will provide a short description of the program itself and will show what and how you can configure this application program to match your personal preferences.

2.1 Purpose of the Application

The Project Editor application was designed to simplify the task of file management while developing a project. On the one hand it will manage the header files, class implementation files and resource scripts for you; on the other hand you can *visually* design new Objective C classes used in your application program and generate a skeleton source code for these classes. Additionally, the "`Makefile`" is created automatically and the actions of compiling and testing your program can be handled from the Project Editor program itself.

To make use of the Project Editor you must create a so-called *project* when starting to develop an application. For every project a directory will be created on your disk containing all necessary files, e.g. the source code and the user interface definitions.

A project consists of

- a *project description* and the project's *name*,
- some settings required to *compile and link* the application program, e.g. the name of the executable file and the libraries your program makes use of,
- *user interface definition files* which you create visually using the Interface Editor program,
- *resource scripts* for the application resources not stored in user interface definition files as created by the Interface Editor.
- a *definition* of all custom classes[1] including their instance methods (*actions*) and instance variables (*outlets*), and the source code files,
- the Objective C *interface declarations* for your custom classes along with

[1] The term *custom class* refers to classes not provided by the library package

- the *class implementation* files.

Every other file your application program will require at run-time should also be put into the project directory.

The Project Editor is some kind of *control center* concerning the development process; you can invoke text-editor programs to edit the source code files, you can compile and even test your programs by simply choosing one of the items in the pull-down menus.

In addition to the files you edit using external programs, e.g. the interface or implementation files or the user interface definitions, the Project Editor will create two data files storing

- the definitions of the custom classes your application program makes use of. This data is stored in the file "`pb.cls`" in your project's root directory, and
- the global settings for the project itself. Data concerning the position and size of the project window, project settings as the name of the executable file and the libraries you wish to use are stored in the data file "`pb.prj`".

2.2 Starting the Project Editor Program

After you have started the Project Editor, a window alike the one depicted in Figure 2.1 will be shown.

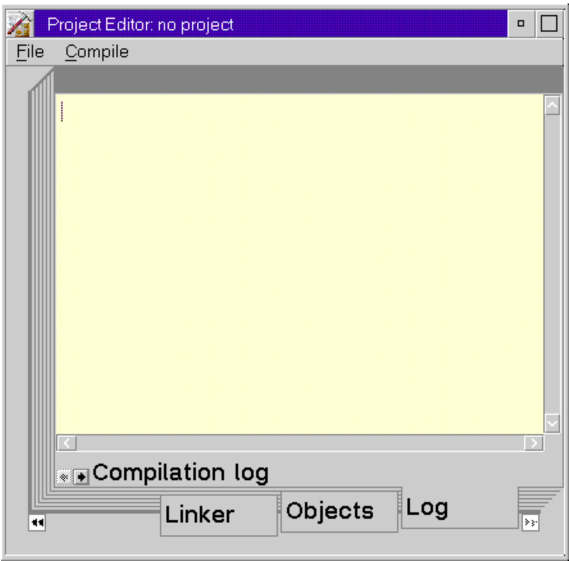

Figure 2.1: Log window of the Project Editor application as displayed on startup

The application itself simply consists of a notebook control and a menu bar with two pull-down menus *File* (see Figure 2.3) and *Compile* as shown in Figure 2.4.

2.3 Configuring the Project Editor for the First Time

Global configuration data for the Project Editor is stored in a binary configuration file named "`PB.INI`". It is located in the directory where the executable file "`pb.exe`" resides an will be created automatically when you start the application for the first time.

In this configuration file some global data will be stored. This includes

- The *size and position* of the main window of the Project Editor when started without a project already loaded and
- configuration data concerning the external programs used for editing the various files.

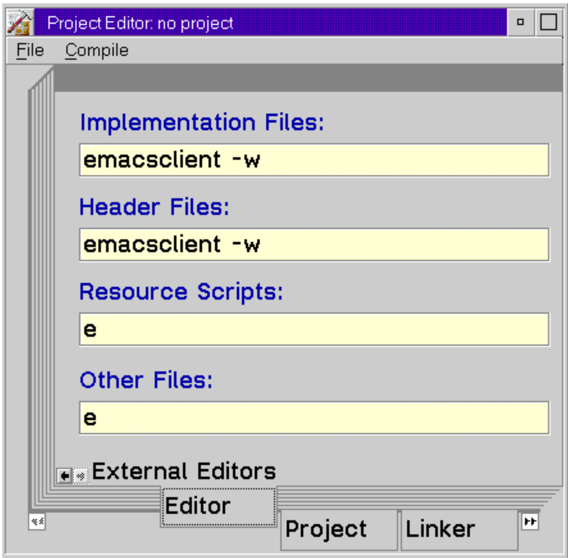

Figure 2.2: Settings page concerning the external programs used for editing the various data files

Before using the program to develop your own applications, you should first tell the Project Editor which applications you want to use for editing your source code files. On the left-most page of the Project Editor's notebook—as depicted in Figure 2.2—you can enter the names and command line arguments for the programs used for editing

- your classes' *implementation files*,
- the interface definitions of your classes and other *header files*,
- the *resource script* files which shall be compiled with the resource compiler and linked to your application program and
- all *other source code* files, e.g. C source code containing external functions your program makes use of.

To tell the Project Editor to use some external program when editing one of these special files just enter the name and the arguments when starting the external program in the appropriate entry field. When editing a file, this program will be started passing one more parameter, the *absolute path name* of the file to be edited consisting of the drive, the path and the file name where it is located.

After making these settings, save the data to the configuration file by choosing the item *Save Preferences* from the *File* menu.

2.4 A Description of the Menus

In the *File* menu you can find menu items to *load* or *save* project data, start a *new* project, to *edit* the classes in your project and to *exit* the application. This pull-down menu is depicted in Figure 2.3.

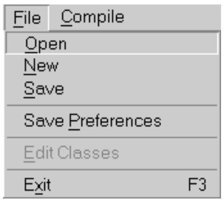

Figure 2.3: *File* menu of the Project Editor

The second pull-down menu the Project Editor application provides is labelled with *Compile*. In this menu, as shown in Figure 2.4, you can find menu items concerning compilation and running the application program you are developing.

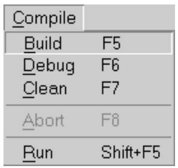

Figure 2.4: *Compile* menu of the Project Editor

- *Build* will compile and link the application program. The executable file as set on the linker page of your project will be created in the project directory.
- The menu item *Debug* will recompile all class implementation files which have been changed and will preserve debug-information usable by the *GNU Debugger* "gdb". An executable file containing debug-code called "`debug.exe`" will be created in the project directory. It is recommended to remove the object files before creating an executable file used for debugging using *File/Clean*.
- *Clean* will remove all object files and the executable files from the project directory. The term object files also refers to the binary resource files created with the resource compiler. In addition all temporary files ending in a tilde (~) will be deleted.
- The item *Abort* will stop the compilation process or kill your application if it was started using *File/Run*. When stopping the compilation process, a damaged object file is likely to exist, so it is recommended to remove all object files after aborting compilation.
- *Run* will start the application program to let you test your project. Before it will be recompiled if any changes were made since the last time the executable file was created.

While compiling or running the application, the output of the programs—*make* while compiling or your application when running—will be displayed on the *Log* page of the project notebook.

The two devices `stdout` and `stderr` are redirected to this notebook page (see Figure 2.1). This means, that you can use these two devices for debugging output which will automatically be displayed when your program is started from within the Project Editor application. Simply use `fprintf(stderr,...)` to print debugging output to the Project Editor's window.

Project Editor will use an external program for compiling and linking your applications. You have to install some *make* program, e.g. *GNU make* found on `hobbes.nmsu.edu`. The program is expected to be located in some directory listed in your `PATH` variable and is expected to be called `make.exe`.

The *make* program must support the `-C` option to specify the working directory for compiling, linking and the "`Makefile`" itself.

If the `make` subprocess exits with an error saying that there are too many open files in your environment and you are using a program depending on the *emx* libraries, you should increase the global file limit by setting the corresponding flag in the `EMXOPT` variable. To allow a maximum of 255 open files at once, put the line

```
SET EMXOPT=-h255
```

into your "`config.sys`". *make* will not be able to find the project directory if you set a *default drive* to be used for absolute pathnames starting with a slash. Be sure not to set the `-r*` option for the `EMXOPT` variable.

2.5 Editing Global Project Settings

Before starting work on a new project you will have to set some data necessary for compilation and linking the executable file you wish to produce.

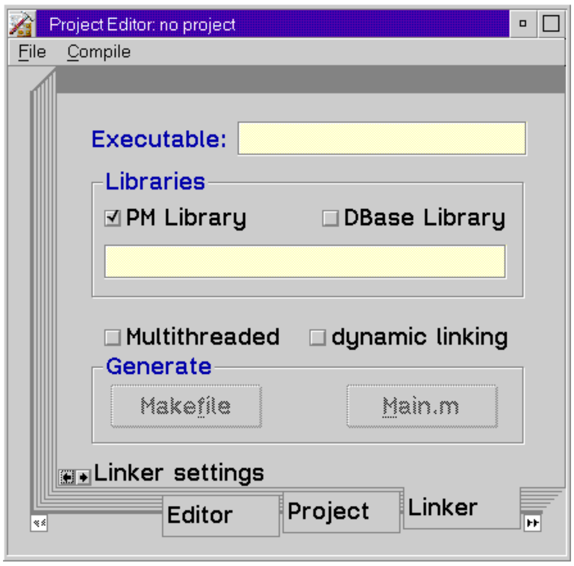

Figure 2.5: The project's linker settings page

To set up your project, turn the Project Editor's notebook pages to the one whose tab is labelled with *Linker*. This page is displayed in Figure 2.5. In the entry field to the right of *Executable:* you will have to specifiy the name of the executable file you wish to create. For example you could enter "`prog.exe`" there resulting in the program to be called "`prog.exe`" when compiling.

In the *Libraries* section of this page you can check or uncheck two boxes labelled with *PM Library* and *DBase Library*. If in checked state, the Presentation Manager library "`objcpm.a`" or the database library "`objcdb.a`" will be linked to your application program. The library file "`objcutil.a`" will always be linked to the executable file. In the entry field below you can specify additional linker options, e.g. enter `-lobjects` to add the library file "`objects.a`" to the list of files linked to your program.

If the check box control *Multithreaded* is checked, the linker flag `-Zmt` will be specified when linking the application causing the multi-threaded versions of the libraries to be used if existing. *Dynamic linking* will cause the C library to be linked dynamically with your program. In this case the dynamic link library "`emx.dll`" will be required by your program.

The two buttons at the bottom will allow you to generate a "`Makefile`" or a new main implementation file called "`Main.m`". You should never change these two files by hand because these changes will be overwritten the next time the files are generated.

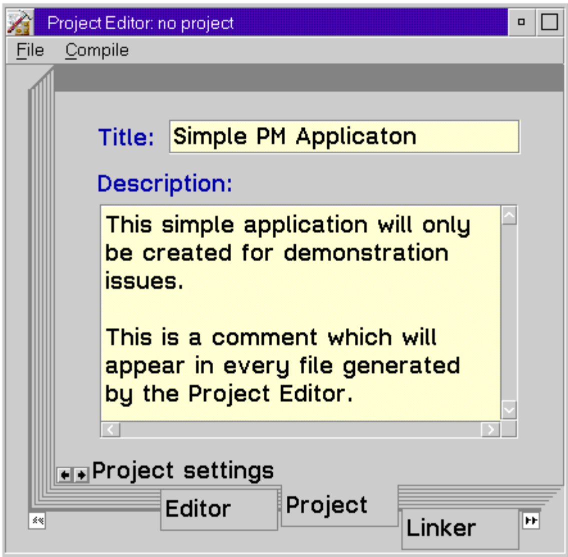

Figure 2.6: Global project settings

You must generate a new "`Makefile`" after you changed some of the settings on this notebook page. Before, you should save the project by choosing *File/Save* from the application's menu.

The page next to this linker settings page is labelled with *Project settings*. On this page, as can be seen in Figure 2.6, you can specify the project's title as a string and a number of lines shortly describing the project. This data will be inserted into some of the automatically generated files.

2.6 Adding Files to Your Project

At the moment only

- Objective C class implementation files,
- header files and Objective C class interface files and
- custom C source code files and resource scripts

are managed by the Project Editor application.

In addition to creating implementation and interface files as will be described in Section 2.8 on Page 9 you can simply add already existing source code files by dragging them from a folder object on the Workplace Shell to the appropriate notebook page in the project window.

Objective C class implementation files and other Objective C source code files having the suffix "`.m`" can be dropped on the first *Objects* page labelled with *Files—Classes*.

The second page is reserved for header files having the file extension "`.h`", on the third page you can drop C source code files (extension "`.c`") and resource scripts (extension "`.rc`"). You can drop more than one file onto one of these pages at once.

2.7 Editing Files

All source code files managed by the Project Editor will be displaced with their icons on the appropriate *Objects* pages.

To start editing one of the files, simply double-click on the icon using the first mouse button. The Project Editor will automatically invoke the external (text) editor program you specified for files of this type (see the global settings displayed in Figure 2.2) and the file will be loaded.

2.8 Creating Custom Classes and Generating Source Code

One of the most interesting features of the Project Editor application is its ability to create custom Objective C classes for use in your project. Adding and deleting classes or editing a classes' interface declaration where *action methods*[2] and *outlet variables*[3].

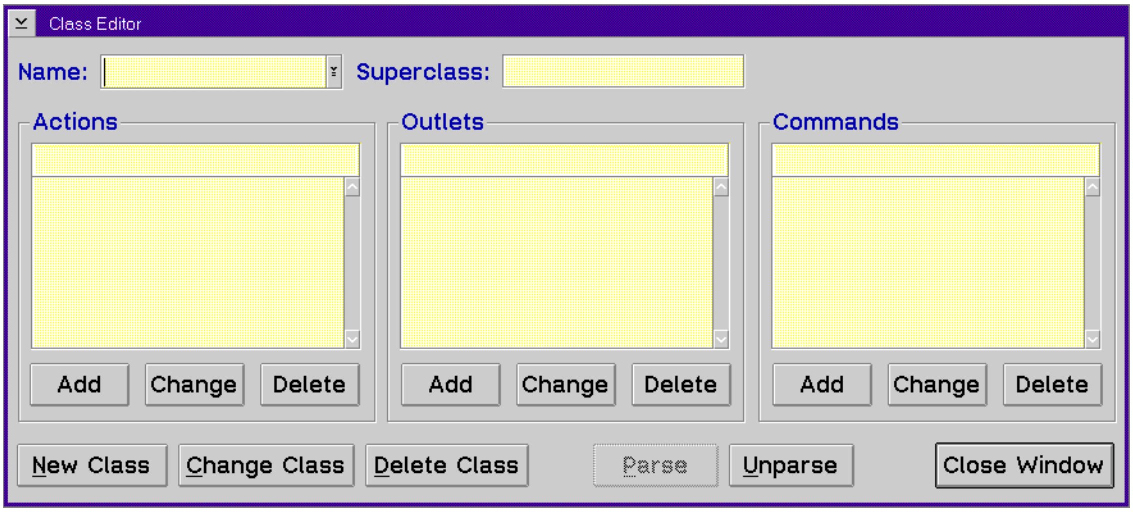

Figure 2.7: Dialog window used for editing information concerning your application's custom classes

To start editing a class interface declaration or create new classes choose *Edit Classes* from the *File* menu. This menu item—as well as most others concerning the project's data files—cannot be selected before a project directory has been created by saving the project data for the first time. The dialog window which is displayed by the Project Editor in response to this menu item is shown in Figure 2.7.

[2]action methods are special instance methods having a single parameter called `sender` being a pointer to the sending object and a return value of type `id` being a pointer to the receiving object in case the method could be executed successfully or `nil` otherwise

[3]instance variables of type `id` are called outlets. These are normally used to store pointers to other objects providing services for your object

On the top of the window you can see two entry field controls labelled with *Name* and *Superclass*. The left entry field really is a combo box only displaying its list after the downwards pointing arrow to the right is pressed. In this list all *custom* classes known by the Project Editor for this project are displayed.

Below three groups are displayed, each of them consisting of a combination box and three buttons to *add*, *change* or *delete* items in the list above.

The leftmost group is concerned with the selected classes' *action methods*, the middle one is used to edit data concerning the *outlet* instance variables. The group on the right margin can be used to add, delete or edit another kind of instance variables called *commands*. You should not care to edit instance variables of this kind now, this will make only sense when developing user interface classes you can edit using the Interface Editor. These special instance variables of type `Command` are used to store a target-action binding from a user-interface element to a specific action method of a target object. The data type is defined in "`<pm/os2types.h>`".

On the bottom you can find some buttons to create new classes, to change a classes name or superclass or to delete a class. Additionally you can press *Unparse* to generate a interface declaration and a skeletonal implementation file for the currently selected class in your project directory. *Close Window* will let you close the (modal) dialog window.

To add a new class, enter the name of the class in the entry field on top labelled with *Name*. The push the *New Class* button on the bottom of the dialog window. Immediately a new class will be created having no action methods, no outlets and no command instance variables. The superclass is set to `Object` if not specified otherwise before pressing *New Class*.

Using *Change Class* you can change the global data (name and superclass) of the currently selected class; *Delete Class* will remove the selected class from memory.

To add an *action*, an *outlet* or a *command*, type the name of the action (outlet/command) in the entry field on top of the the appropriate combination box and press the *Add* button in the group the combo box control is in. *Change* will change the definition of the currently action (outlet/command) to the definition displayed in the entry field.

The names of *action methods* must end in a colon (`:`), the Project Editor will automatically add the parameter `sender` on its own when generating the source code files. To define an action method called `-anAction: sender` simply add `anAction:` to the list of actions.

When generating source code files using *Unparse* you will be prompted to confirm overwriting already existing files. Overwriting will cause all changes you made to the source code files by hand obsolete. Normally you should not press *Unparse* after already editing a source code file. Simply modify the header file for the class and its implementation file by hand.

The name of the header file created by *Unparse* consists of the classname and a suffix of "`.h`". Implementation files are created by adding "`.m`" to the name of the class. Be careful when installing the package on a FAT drive, you will not prompted in case creating the file failed.

To clarify this, assume you are creating a class called `Controller`. *Unparse* will create an interface declaration for Controller in a file called "`Controller.h`" and a skeletonal implementation in "`Controller.m`". You are prompted if you wish to insert the two files into your project. The next time you generate a "`Makefile`" some lines will be inserted to make sure the implementation file of the class `Controller`, "`Controller.m`", is compiled and the object file "`Controller.o`" is linked to your application program.

2.9 Interacting with the Interface Editor program

The class definition file you create using the Project Editor will be used by the Interface Editor application. Using this program you can graphically create instances of your custom classes and connect them with various objects, e.g. with user interface elements as buttons. The Interface Editor will automatically read in the file “`pb.cls`” in the directory the interface file is located.

At the moment the Interface Editor will not notice any change in the class declarations you make using the Project Editor while the interface files concerned by these modifications are already loaded. To update the information the Interface Editor stores (in memory) about your project's custom classes you can choose *File/Reload Classes* from the menu in your interface file's main window.

Chapter 3

Interface Editor—Creating Your Program's User Interface

The *Interface Editor* is used to create the user interface of your programs. In addition to that you can create instances of your project's classes and some frequently used objects, e.g. a `Printer` object or a `Help` object.

An introduction into using the program itself can be found in the *Tutorial*. This chapter will only provide a brief overview of the program itself and explain most of the features.

3.1 What Is Edited

Basically, this program enables you to create objects of any kind in an easy to understand way. You can create user interface objects (windows, buttons,...) as well as non-visible objects (printer objects,...). The data is stored in a so-called *interface file*.

More exactly, the data is stored in an Objective C *Typed Stream*, a mechanism available for storing objects. The objects are divided into two kinds,

- *instances of your own classes*, and
- *instances of pre-defined classes.*

Instances of your own classes are not really stored as objects in an interface file. Only a reference (the name of the class and the title of the object) are stored. These objects are then not *unloaded* by the runtime library using the `-read:` and `-awake` methods but created from scratch via `+alloc` and initialized using `-init`. Every initialization for your classes should be done in the `-init` method.

After a user-defined object is created, the special method `-awakeFromInterfaceFile` is called, if implemented by the object's class. At the time this method is called, all connections between the object and the "rest of the world" are guaranteed to exist. So this is the method, you should put any initialization depending on other objects in the interface file.

Instances of pre-defined classes are unarchived using the standard methods `-read:` and `-awake`. You should not care too much about these methods, these are implemented correctly for the pre-defined classes and will perform as expected. The objects are just created and set up as they were within the Interface Editor program.

The objects and the connections are bundled together to a single instance of the class `InterfaceFile`. This class guarantees that every object is created and initialized as expected. Loading of an interface file is discussed in Section 3.15.

3.2 Limitations

At the moment you can only create instances of the following window classes:

- `CheckBox` and `AutoCheckBox`
- `ComboBox`
- `Container`
- `EntryField`
- `ListBox`
- `MainWindow`
- `MultiLineEntryField`
- `NoteBook`
- `PushButton`
- `RadioButton` and `AutoRadioButton`
- `ScrollBar`
- `Static`
- `StdDialog`
- `StdWindow` and
- `TriStateButton` and `AutoTriStateButton`.

Additionally, you can instantiate the classes

- `Help` and
- `Printer`

and create objects of self-defined classes.

Support for other user interface classes will be added in the next version, just as well as editing more of the so-called *Presentation Parameters* of the window objects. For now you can only edit the *most common* parameters and specify a *custom font* for the windows.

There is no way to create menus and icons using this program. For this you have to write a resource file as is e.g. described in the *Tutorial*.

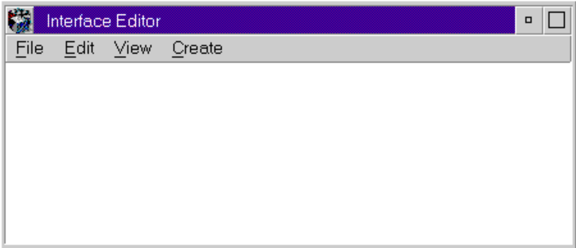

Figure 3.1: Main window of the Interface Editor application

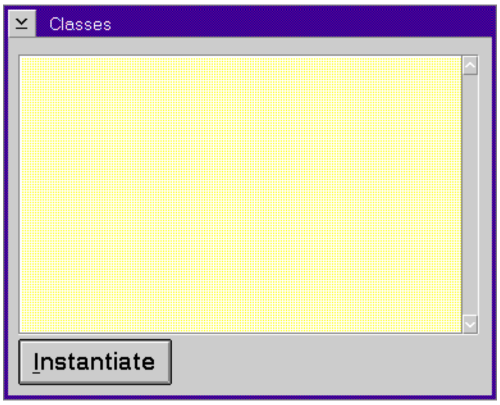

Figure 3.2: Dialog window used to create instances of user-defined classes

3.3 Starting the Program

When starting the Interface Editor program you will see the main window of the application. This window, as is depicted in Figure 3.1, consists of a menu bar and a container object.

In the container object all non-user interface objects and all instances of `MainWindow`, `StdDialog` and `StdWindow` will be represented as an icon.

From the *View* menu, you can cause the other windows of the application to be displayed. These are needed to instantiate user-defined classes (see Figure 3.2), to create user interface objects (Figure 3.3) and to edit the properties of an object, as shown in Figure 3.4.

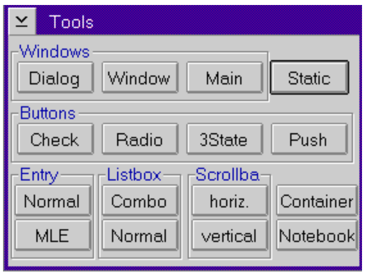

Figure 3.3: Tools palette of the Interface Editor

For practical purpose, you should always let the *Tools* window (Figure 3.3) and the *Preferences* window be displayed on screen. In addition to the main window, these are the most frequently

Figure 3.4: Preferences window used to change the settings of user interface and non-user interface objects

used windows in the application.

All three of the *Palette*, the *Preferences* window and the *Classes* dialog are non-modal dialog windows. They can—and in case of the *Palette* and the *Preferences* window also should—always been displayed on screen. These three windows are shared among all interface files currently open; every interface file is represented by its own *main window*.

3.4 Customization

The Interface Editor will try to read a system configuration file called "`ib.ini`" on startup. In this configuration file,

- the *positon*, *size* and the *font* used for the main window are stored. Additionally,
- the *position* of the *Tools* window (often referred as the *Palette*) and
- the *position* and *size* of the *Preferences* window will be remembered and saved. Of course, this also applies to
- the *position* of the *Classes* window.

The settings can be saved by clicking on *Save* on the last page displayed in the *Preferences* window.

Normally, the configuration file "`ib.ini`" resides in the application directory `\usr\bin\PM`.

In addition to the positions of the single windows, the *visibility* state of every window is also stored. So if *Palette* and *Preferences* are displayed when saving the settings, these windows will automatically shown when the application is started the next time.

On the *Global Preferences* page shown in Figure 3.5 of the *Preferences* window you can also set the *horizontal* and *vertical* raster for positioning and sizing the user interface elements within a

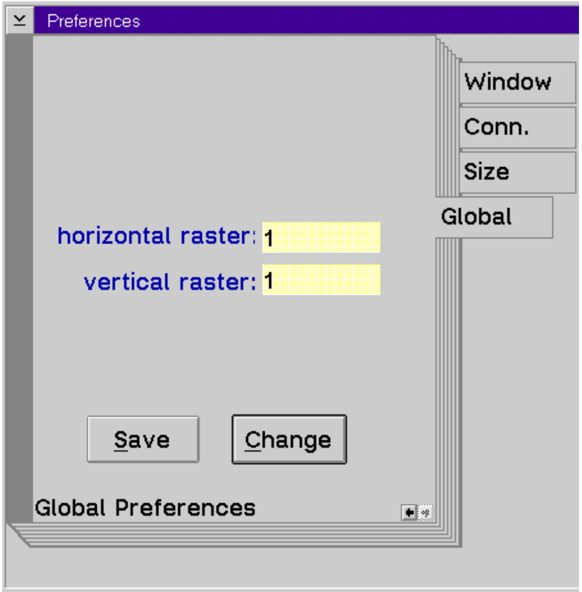

Figure 3.5: *Global Preferences* page of the *Preferences* window

window or dialog. If you set the horizontal raster to 2 and the vertical raster to 4 this means, that the objects can only be aligned to positions (x/y), where x is a multiple of 2 and y is a multiple of 4. This also applies to the size of the objects.

To change these settings, enter the horizontal and/or vertical raster in the entry fields labelled *hoizontal raster* respectively *vertical raster*. You must press *Change* to make the settings known to the program for the current session; to save the settings permanently, press *Save*.

3.5 Adding Objects

Starting here a distinction between the objects represented by an icon in the main window and the so-called *additional (user interface) objects* is made. The term *(user interface) object* will refer to an object that is represented in the main window. *Additional (user interface) objects* are objects just being part of some other object, e.g. a button in a window.

Creating "normal" objects is accomplished by either

- creating an instance of `MainWindow`, `StdDialog` or `MainWindow` by clicking on the appropriate buttons *Main*, *Dialog* or *Window* in the *Palette* or by
- instantiating a custom (user-defined) class using the *Classes* dialog or by
- choosing *Help Object* or *Printer Object* from the *Create* menu in the main window to create an instance of `Help` respectively `Printer`.

Each of these objects is represented by an icon in the main window and has a title. The title can be changed using the OS/2 direct manipulation features for *direct editing* of a text. This title string is used to uniquely identify the objects.

3.6 Adding Additional User Interface Objects

These so-called *additional user interface objects* are created by clicking on the corresponding button controls in the *Palette*. The following table will list the button text and the class an instance is created of when the button is clicked.

Button Text	*Class Name*
Windows/Dialog	`StdDialog`
Windows/Window	`StdWindow`
Windows/Static	`Static`
Buttons/Check	`AutoCheckBox` or `CheckBox`
Buttons/Radio	`AutoRadioButton` or `RadioButton`
Buttons/3State	`AutoTriStateButton` or `TriStateButton`
Buttons/Push	`PushButton`
Entry/Normal	`EntryField`
Entry/MLE	`MultiLineEntryField`
Listbox/Combo	`ComboBox`
Listbox/Normal	`ListBox`
Scrollbar/horiz.	`ScrollBar`
Scrollbar/vertical	`ScrollBar`
Container	`Container`
Notebook	`NoteBook`

3.7 Sizing and Moving User Interface Objects

There are two possible ways to set the size and/or the position of a user interface object.

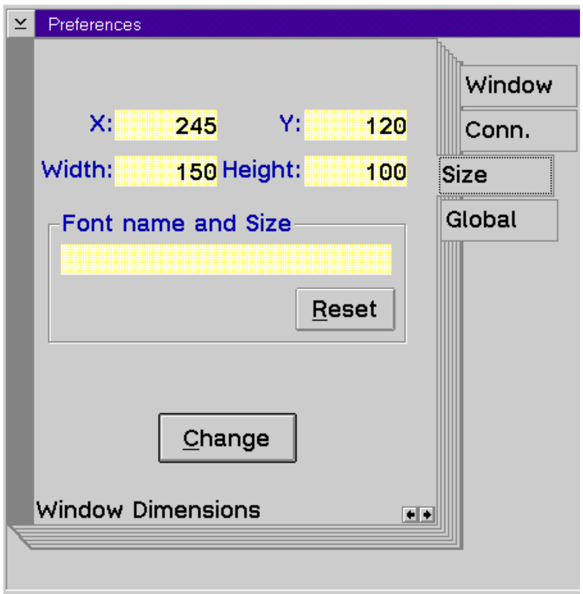

Figure 3.6: *Window Dimensions* page for user interface objects

First, you can "manually" enter the desired size and/or position of the object within its parent window on the *Window Dimensions* page of the *Preferences* window (see Figure 3.6). After doing so you should press *Change* on this page to actually perform the changes.

The other, the more intuitive and simpler way is to just change the position of an additional user interface object by *dragging* the object while holding down *mouse button 2*. Depressing button 2 while the pointer is over a window object will cause the object to "follow" the mouse pointer. As soon as you release the mouse button, the object will be "dropped" at its new position.

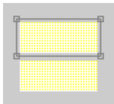

Figure 3.7: Two *entry field* objects in a dialog window. The object on top is selected.

To change the size of an additional user interface object you have to first select the object by clicking the *first mouse button* while the pointer is over the object. Immediately, a border will be drawn around the object (as is shown in Figure 3.7). Moving the pointer to the edges of the border you can see the pointer shape change to double arrows pointing into some direction. Pressing button 1 and dragging while the *arrow pointer* is visible will cause the object to be resized dynamically. As soon as you release button 1 the object will adjust to its new size. Some user interface objects, e.g. instances of `Container` or `MultiLineEntryField` will cause problems with redrawing themselves while resizing. If you experienced a slightly *damaged* window, just deselect the window and select it once more to cause a "clean redraw".

Moving "normal" user interface objects, such as dialog windows is done by dragging them using their title bar. If you have disabled the title bar, just enable it again, then change the position of the window and remove the title bar in the end.

To change the size of a window, you have to change its border type to *resizable*. Then you can resize it as any resizable window on the desktop.

3.8 Removing Objects

Removing objects is done by selecting their icon in the main window and choosing *Delete Object* from the *Edit* menu. You can also press the *Delete* key on your keyboard as a short-cut for this menu item.

If the object you want to remove is connected to other objects, you will be warned of this and have to confirm the delete action.

Be careful when deleting objects, because there is no *undo* function at the moment.

3.9 Removing Additional User Interface Objects

Additional user interface objects are removed by selecting the object you want to delete (see Figure 3.7 for a picture of a selected object) and the press the *Delete* key on your keyboard. If the object is not connected to other objects, it is deleted immediately. Otherwise, you will be warned of this fact and have to confirm the this action.

3.10 Changing Presentation Parameters for User Interface Objects

In contrast to most dialog editors available now, the scope of the Interface Editor application is quite limited when the "presentation parameters" of an object are concerned. At the moment it is only possible to change the font (font and size) of any user interface object.

To set the font of a user interface object start the *Font Palette* application in your *System Setup* folder and drag the desired font (using button 2) to the object you want to change. By simply dropping a font on the object the font change will be performed. This font will be stored to the interface file when saving the data.

You can inspect the name and size of the font on the page labelled with *Window Dimensions* in the *Preferences* window. An entry of the form `s.n` is used to denote that the object uses a font called `n` at size `s` point. E.g. the entry

```
12.Courier
```

means that the object uses a `Courier` font at `12` point for display of its data.

To remove the special font information from a user interface object, press *Reset* on the *Window Dimensions* page (see Figure 3.6 on page 18). Thereafter the object will be displayed using the default font for this kind of window object.

3.11 Editing Preferences of Objects

This section will try to explain which settings you can change using the *Preferences* window. At the moment this description is very short. For most of the flags you can set or unset, only the corresponding flags specified at object creation time are referenced. Have a look at the appropriate sections in the *Reference Manual* to find out what these flags are used for and how the look and feel of the object is influenced by them.

Before starting a complete description of the single *Preferences windows*, some general notes must be made.

Every user interface object, or seen in a more abstract way, every instance of a class derived from `Window` do have three things in common:

1. Every window object has a *title*. This title can be edited in the first of the preference pages.

2. In addition to that, a user interface object can store an integer value called the window's *tag*. This value can be quite useful to distinguish windows, or to add some numeric data. Just as the window title, this value is displayed and can be edited on the first page.

3. At last, every window has a presentation manager identifier, denoted by *ID* in the preferences window. This identifier can normally be ignored by the developer; it must be adjusted only in cases where resources should be automatically associated with the window object (e.g. a menu resource for an instance of `StdWindow`).

Changes to the settings of an object made with a radio button or a check box control are normally performed immediately. Only changes to the settings made in an entry field, as e.g. changes to the window title, must be acknowledged by the user by pressing the button labelled with *Change* on the corresponding preferences page.

3.11.1 “StdDialog”

Instances of the class `StdDialog`, which are created using the button *Dialog* in the *Palette* can be influenced by changing the settings on two different preferences pages.

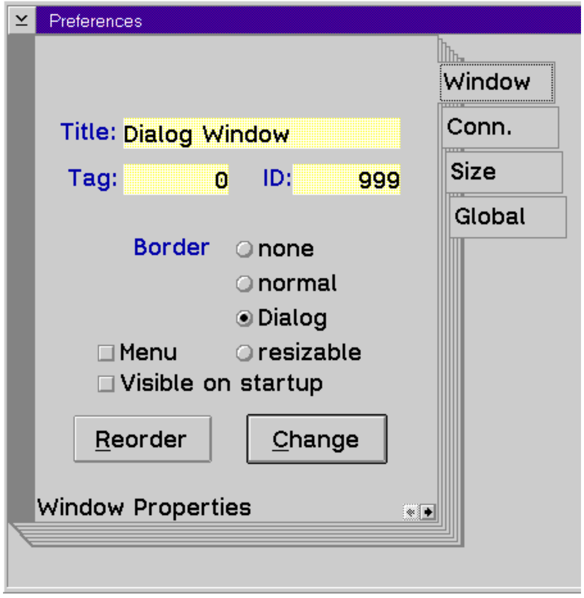

Figure 3.8: First preferences page for dialog windows

The first page, as depicted in Figure 3.8, is used to change the global settings (dialog title, window tag, etc.) and to set some flags.

The radio buttons and check boxes on this page correspond to the dialog window creation flags as

Title	*Flag*
Border/none	—
Border/normal	`FCF_BORDER`
Border/Dialog	`FCF_DLGBORDER`
Border/resizable	`FCF_SIZEBORDER`
Menu	`FCF_MENU`

Checking *Visible on startup* causes the dialog window to be displayed automatically when the interface file is loaded. The dialog is run non-modal (using `-makeKeyAndOrderFront:`). This also applies to instances of `StdWindow` and `MainWindow`.

To change the ordering of the child windows of the dialog window, click on *Reorder*. This causes a modal dialog window as depicted in Figure 3.9 to be displayed.

The upper list in this dialog window represents the current ordering of the user interface objects in the dialog window. Using the two buttons *Delete* and *Add* you can move the selected object from the upper list (*Add*) to the lower list or vice versa (*Delete*). The ordering is set by pressing *Apply*.

When selecting *Apply* while the list at the top is not empty, the items there are appended to the list at the bottom in the same ordering they have in the top list.

The second preferences window for a dialog (see Figure 3.10) is used to set some additional flags. These flags are explained in the following table:

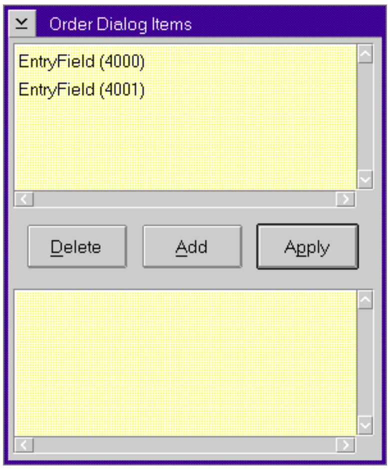

Figure 3.9: Reordering items in a dialog window, a main window or a standard window

Title	*Flag*
Titlebar	FCF_TITLEBAR
System menu	FCF_SYSMENU
Buttons/Minimize	FCF_MINBUTTON
Buttons/Maximize	FCF_MAXBUTTON
Buttons/Hide	FCF_HIDEBUTTON
Scrollbars/Vertical	FCF_VERTSCROLL
Scrollbars/Horizontal	FCF_HORZSCROLL
Resources/Accelerator table	FCF_ACCELTABLE
Resources/Icon	FCF_ICON
Tasklist	FCF_TASKLIST
No byte align	FCF_NOBYTEALIGN
No move with owner	FCF_NOMOVEWITHOWNER
Automatic Positioning	FCF_SHELLPOSITON

3.11.2 "StdWindow" and "MainWindow"

Instances of `StdWindow` and `MainWindow` are again created using the *Tools Palette*. The preferences pages look just alike the preferences pages for dialog windows.

The only difference is that on the first page the size of the *client area* of the window is displayed. The size of the client area is just the size of the window without the border and the rest of the frame controls.

Be careful when moving or resizing child objects of a main window or standard window. In some cases there are some problems with redrawing. If you experience a problem with redrawing a window object after resizing or moving, just deselect the object and select it once again.

3.11.3 "Static"

The radio buttons as displayed in Figure 3.11 correspond to the flag constants as

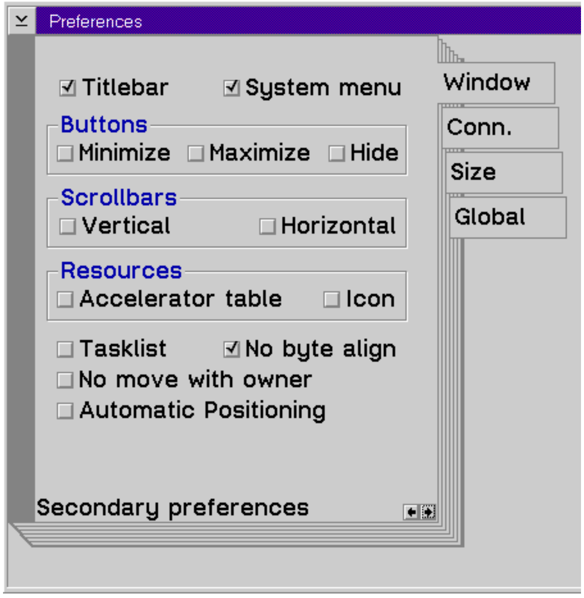

Figure 3.10: Second preferences page for dialog windows

Title	*Flag*
foreground frame	SS_FGNDFRAME
background frame	SS_BKGNDFRAME
halftone frame	SS_HALFTONEFRAME
foreground rectangle	SS_FGNDRECT
background rectangle	SS_BKGNDRECT
halftone rectangle	SS_HALFTONERECT
Text	SS_TEXT
Groupbox	SS_GROUPBOX

3.11.4 "CheckBox" and "AutoCheckBox"

Simple check boxes and automatic check boxes are both created using *Buttons/Check* in the *Palette*. Which class is used for creating the object is determined by a check box in the preferences page.

The two check boxes *Group* and *Tabstop* correspond to the window flags `WS_GROUP` respectively `WS_TABSTOP`. If *Autocheckbox* is in checked state, the object will be an instance of `AutoCheckBox`, otherwise, an instance of `CheckBox` will be used.

3.11.5 "RadioButton" and "AutoRadioButton"

Radio buttons and automatic radio buttons are both created using *Buttons/Radio* in the *Palette*. Which class is used for creating the object is determined by a check box in the preferences page.

The two check boxes *Group* and *Tabstop* correspond to the window flags `WS_GROUP` respectively `WS_TABSTOP`. If *Autoradiobutton* is in checked state, the object will be an instance of `AutoRadioButton`, otherwise, an instance of `RadioButton` will be used.

not Cursor selectable represents the flag `BS_NOCURSORSELECT`. If in checked state, the radio button (or automatic radio button) cannot be selected using the cursor keys.

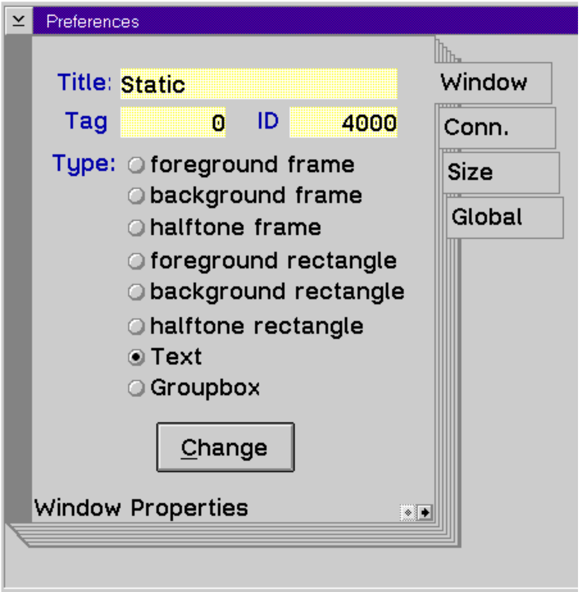

Figure 3.11: Preferences page for instances of `Static`

3.11.6 "TriStateButton" and "AutoTriStateButton"

Tri-state buttons and automatic tri-state buttons are both created using *Buttons/3State* in the *Palette*. Which class is used for creating the object is determined by a check box in the preferences page.

The two check boxes *Group* and *Tabstop* correspond to the window flags `WS_GROUP` respectively `WS_TABSTOP`. If *Autotristatebutton* is in checked state, the object will be an instance of `AutoTriStateButton`, otherwise, an instance of `TriStateButton` will be used.

3.11.7 "PushButton"

In the preferences window for push buttons the radio buttons and check boxes correspond to the flag constants as shown in the following table:

Title	*Flag*
Group	`WS_GROUP`
Tabstop	`WS_TABSTOP`
Default	`BS_DEFAULT`
Border	not `BS_NOBORDER`
Command	—
Help	`BS_HELP`
System Command	`BS_SYSCOMMAND`

3.11.8 "EntryField"

The look and feel of an entry field control as created from the *Palette* using *Entry/Normal* can be modified by using the *Window Properties* page of the *Preferences* notebook.

The setup page as depicted in Figure 3.12 contains four entry fields and some radio buttons and check boxes used to set the control flags.

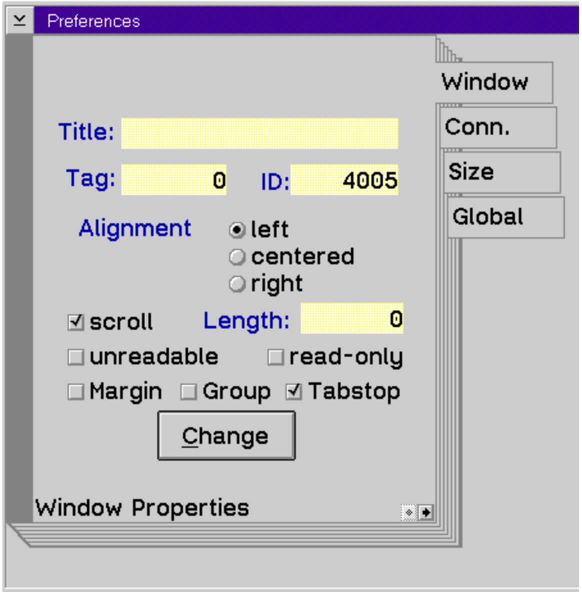

Figure 3.12: Setup page for entry field controls

The three entry fields labelled with *Title*, *Tag* and *ID* are just the same as already described previously. The one titled *Length* can be used to specify the maximal length of the data which can be entered by the user. A value of 0 means that the length of the string is not limited, a value of $n \neq 0$ otherwise will cause the input string to be limited to at most n characters.

The check boxes and radio buttons correspond to the window flags as will be shown in the following table. For a description of the flags and how setting or unsetting them will modify the entry field's look and feel see the *Reference Manual*.

Title	*Flag*
Alignment/left	`ES_LEFT`
Alignment/centered	`ES_CENTER`
Alignment/right	`ES_RIGHT`
scroll	`ES_AUTOSCROLL`
unreadable	`ES_UNREADABLE`
read-only	`ES_READONLY`
Margin	`ES_MARGIN`
Group	`WS_GROUP`
Tabstop	`WS_TABSTOP`

3.11.9 "MultiLineEntryField"

Instances of the class `MultiLineEntryField` are created by simply pressing *Entry/MLE* in the *Tools Palette*. These objects are special entry fields suited for entering more than a single line of data.

The behaviour of MLE windows can be influenced by checking or unchecking some of the check boxes in the *Window Properties* page of the *Preferences* window. The single check boxes there are associated with the `MLS_*` flags as

Title	*Flag*
word-wrap	MLS_WORDWRAP
read-only	MLS_READONLY
ignore TAB key	MLS_IGNORETAB
horizontal scrollbar	MLS_HSCROLL
vertical scrollbar	MLS_VSCROLL
Margin	MLS_BORDER
Group	WS_GROUP
Tabstop	WS_TABSTOP

3.11.10 "ComboBox"

Comboboxes are just a combination of an entry field and a normal list box. The appearance of objects of the class `ComboBox` can be controlled using the *Preferences* notebook.

You can choose the type of the control by choosing one of *Type/Simple*, *Type/Dropdown* and *Type/Dropdown list*. These radio buttons correspond to the flag constants CBS_SIMPLE, CBS_DROPDOWN and CBS_DROPDOWNLIST.

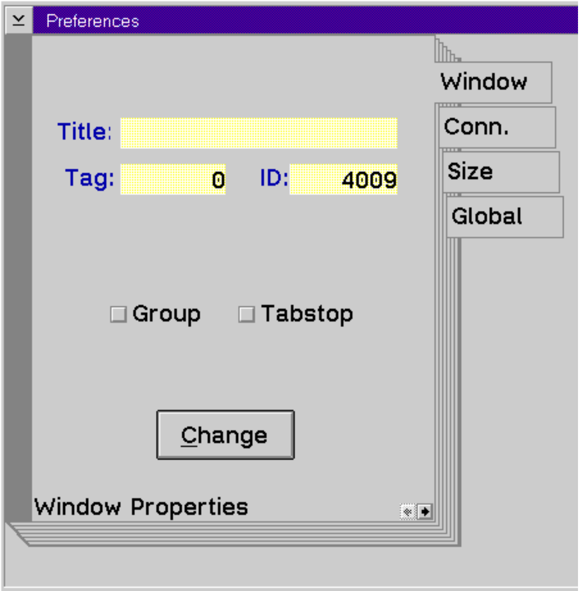

Figure 3.13: "Standard" *Window Properties* page for use with user interface objects that do not have special properties, e.g. scroll bars and container windows.

The two check boxes *Group* and *Tabstop* represent the values of the flags WS_GROUP and WS_TABSTOP.

3.11.11 "ListBox"

User interface objects created by pressing *Listbox/Normal* in the *Tools Palette* are instances of the library class `ListBox`. You can add a horizontal scroll bar to the list box by checking *horizontal Scrollbar* (LS_MULTIPLESEL) or enable multiple selection (LS_MULTIPLESEL) or extended selection techniques (LS_EXTENDEDSEL) by checking the boxes beneath *multiple Selection* or *extended Selection*.

The two check boxes *Group* and *Tabstop* again represent the state of the flags WS_GROUP respectively WS_TABSTOP.

3.11.12 "ScrollBar"

At the moment you can use *Scrollbar/horiz.* and *Scrollbar/vertical* to create instances of the PM class `ScrollBar`. Horizontal and vertical scroll bars to have the same preferences pages and provide only support for modifying the `WS_GROUP` and `WS_TABSTOP` properties via *Group* and *Tabstop*.

The "standard" *Window Properties* page shown in Figure 3.13 is used whenever there are no special settings available for the selected window. This page is equally used for some windows, e.g. scroll bars or container windows. Better support for these windows will be added in future versions of the Interface Editor.

3.11.13 "Container"

As was already mentioned, container windows do not have special properties to be set via the *Preferences* window. See the last section for a description of the *Window Properties* page of the *Preferences* window. Figure 3.13 depicts this simple settings page.

3.11.14 "NoteBook"

Instances of the class `NoteBook` can be seen as a container for many different dialog windows—normally all being the same size. The pages of the notebook are set using the instance methods (see the *Reference Manual* for more information on this topic), the general appearance of the notebook is set by the user on two different preferences pages.

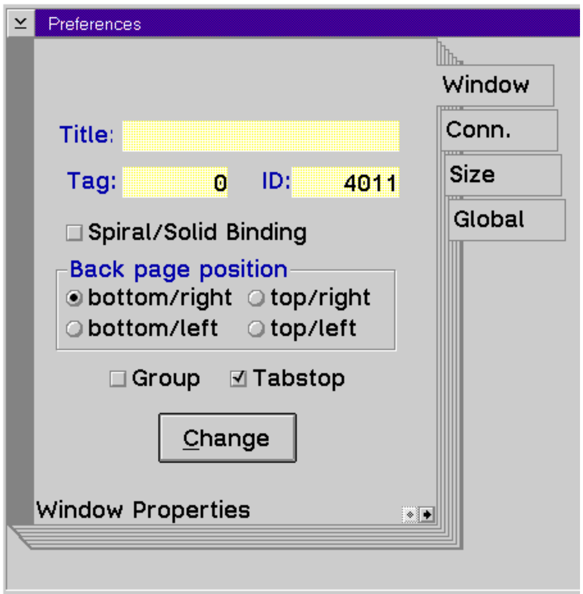

Figure 3.14: First preferences page for instances of `NoteBook`

The first page—as depicted in Figure 3.14 is used to set the general window data as the window *Title*, the *Tag* value and the presentation manager identifier *ID*. Further the behaviour of the window in a dialog is set using the check boxes *Group* (`WS_GROUP`) and *Tabstop* (`WS_TABSTOP`).

There is a check box labelled *Spiral/Solid Binding* which is responsible for the type of "binding" the user will see. If in checked state, a spiral binding (flag `BKS_SPIRALBIND`) is used, otherwise a notebook having solid binding (`BKS_SOLIDBIND`) is displayed.

The rest of the page contains a group of radio buttons (*Back page position*) determining the position of the notebook's back page. The mutual exclusive settings defined by the four radio buttons are

Title	*Flag*
bottom/right	BKS_BACKPAGEBR
bottom/left	BKS_BACKPAGEBL
top/right	BKS_BACKPAGETR
top/left	BKS_BACKPAGETL

On the second page of the special preferences for the notebook window you will see four distinct group of radio buttons.

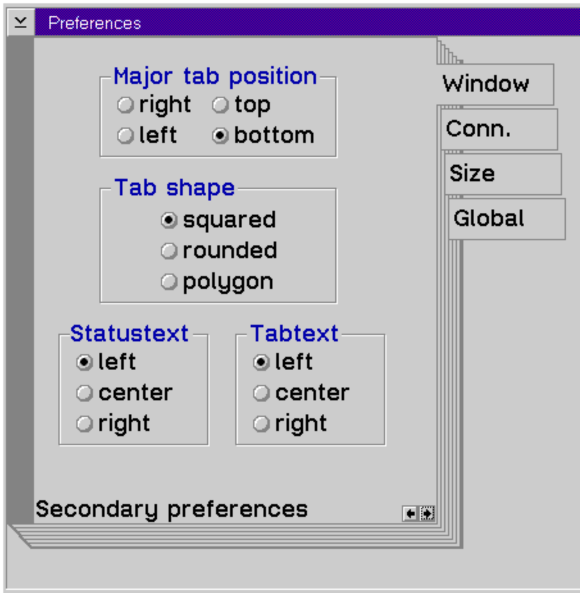

Figure 3.15: Second preferences page for notebook windows

These are labelled with *Major tab position*, determining the position where the (major) tabs are displayed, *Tab shape* defining the shape of the tabs, *Statustext* and *Tabtext* each defining the position of either the status text for every page or the tab text for the (major) tabs. The window is depicted in Figure 3.15.

The first group, *Major tab position*, is used to determine which of the flags defining the major tab position is used:

Title	*Flag*
right	BKS_MAJORTABRIGHT
left	BKS_MAJORTABLEFT
top	BKS_MAJORTABTOP
bottom	BKS_MAJORTABBOTTOM

The group of radio buttons titled *Tab shape* will change the flags according to the following table:

Title	*Flag*
squared	BKS_SQUARETABS
rounded	BKS_ROUNDEDTABS
polygon	BKS_POLYGONTABS

The next group (the left group on the bottom of the page) is used to set the position of the status text for the pages and is labelled *Statustext*. It is used to modify the notebook flags

Title	*Flag*
left	BKS_STATUSTEXTLEFT
center	BKS_STATUSTEXTCENTER
right	BKS_STATUSTEXTRIGHT

The horizontal alignment of the tab text is specifed using the last group named *Tabtext*. The three radio buttons again are mutually exclusive and will define the flags as shown in the following table:

Title	*Flag*
left	BKS_TABTEXTLEFT
center	BKS_TABTEXTCENTER
right	BKS_TABTEXTRIGHT

3.11.15 "Printer"

At the moment there is no disting preferences page for instances of the library class `Printer`.

To specify a custom setup dialog for your printer objects, use the Interface Editor's object connection facilities.

3.11.16 "Help"

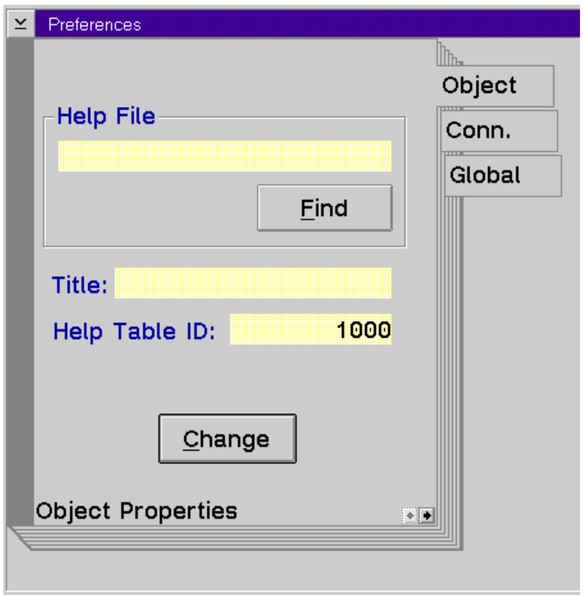

Figure 3.16: Preferences page for `Help` objects

The preferences page for `Help` objects is shown in Figure 3.16. You must specify a *Help File* in the corresponding entry field. Just enter the name of the file, e.g. `program.hlp` in this field. The *Find* button below this filed does not work at the moment. You should not enter an absolute path name for the help file. Just enter the name of the help file without a full path.

The entry field labelled with *Title* is used to specify the title text of the help window. The *Help Table ID*, 1000 by default, is the identifier of the help table in the resource file. To utilize `Help` objects, you must create a help file, e.g. using `ipfc`, and create an entry for the main help table in the resource file for the application.

3.12 Connecting Objects

In the *Tutorial* as well as in the *Reference Manual* the design philosophy of the library package is introduced. Every user action is normally processed by one of the standard PM classes in the beginning. If an object thinks it is not able to react to a user action as expected then it will try to send a message to some other object (in plain C you would call this method a *call-back function*) letting it perform the additional work. Which method is called is either specified by the user or by the sending object itself.

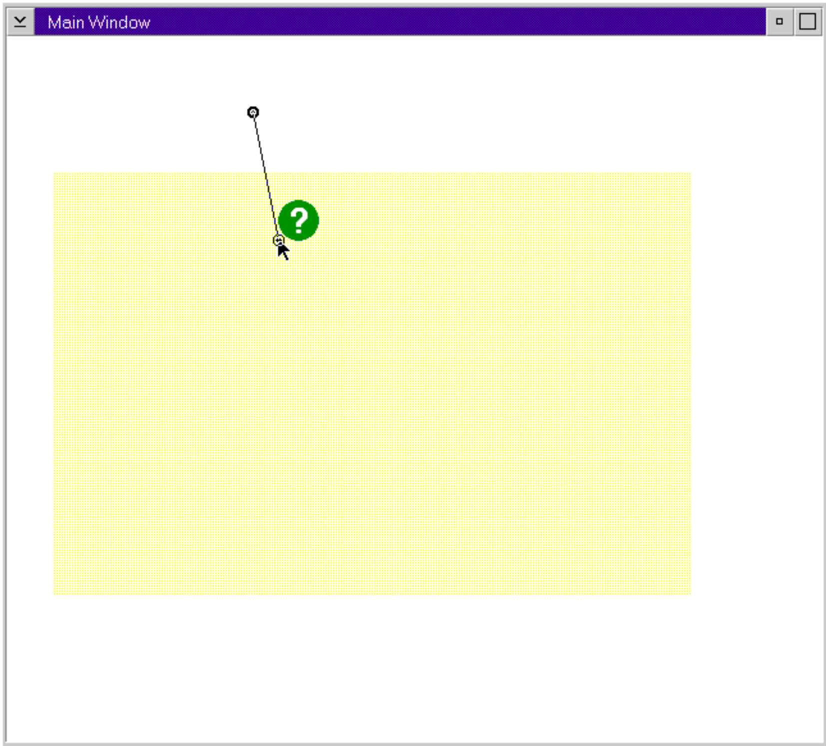

Figure 3.17: Connecting a main window with a multi-line entry field

The first case is used for user actions issuing some kind of *command*. The application programmer has to decide which method of some object should be called in case the user issues a special command. You can set up such *command-action bindings* by using the methods `-bindCommand:withObject:selector:` (for standard, main and dialog windows) or `-bindWith:selector:` in case the command is to be issued from a button control. Previously, command-action bindings were used to connect menu items or button controls with special methods to be executed when the item was chosen or the button was pressed. It must be noted that the term *command-action binding* will only indicate a binding directly originating from the sending object itself, i.e., bindings set up via `-bindCommand:withObject:selector:` will not fall into this category.

The other way of sending a message to some object is called *delegating*. An object will *delegate* some of its functions to another object. Only the object must be specified by the application programmer. The sending object knows, what special methods to call if it is required to do so. Delegate objects are normally set using the `-setDelegate:` method. Connections of this kind are called *outlet connections*. You will remember that in the original "Textview" application the delegate object of the main window provided a method `-windowDidResize:` which was called whenever the main window was resized. As a consequence, this method adapted the size of the multi-line entry field to fill the complete client area of the main window.

It has already been mentioned before that the Interface Editor program provides more functionality than a normal dialog editor. In addition to creating instances of user-defined classes, the Interface Editor can also be used to set up *command-action bindings* and *outlet bindings*. A *command-action binding* is defined to be a connection between two objects uniquely connecting a *command* (represented by an instance variable of type `(Command)` in the sending object) with a specific *action method* of the target object. *Outlet bindings* are simply connections of an outlet instance variable (an instance variable of type `(id)` in the source object) with a target object. Every time a command is issued, the associated action method of the target object is automatically called, and in case of an outlet binding, the outlet instance variable is initialized to be a pointer to the target object.

To set up a command-action binding or an outlet binding you can simply use the OS/2 direct manipulation facilities. Move the mouse pointer to the source object (either a window object or an object in the main window of the interface file) and make an *object reference* to the target object as you would do to make an object reference on the workplace shell. You can either press the *Ctrl* and the *Shift* key and "drag" the object with the mouse, or simply press *mouse button 3* (normally the middle mouse button) while moving the pointer from the source object to the target object.

In Figure 3.17 the process of connecting an object (a main window in this special case) with another object is shown. The purpose of this type of connection will be to initialize an *outlet* instance variable of the source object, the main window.

After "dropping" the main window on the multi-line entry field, a dialog letting you choose the command-action or outlet to connect will appear on the screen. This dialog is depicted in Figure 3.18.

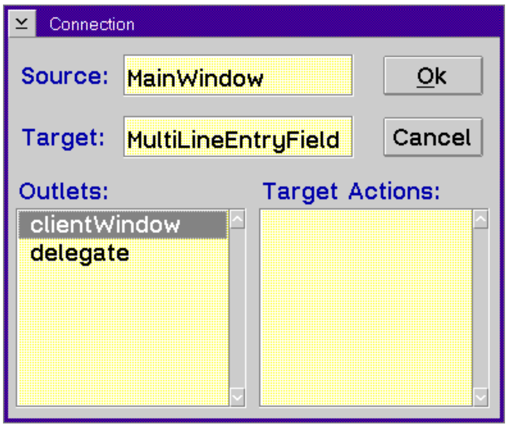

Figure 3.18: Dialog for connecting the an outlet instance variable of a main window with a multi-line entry field

In the list box control on the left a list of outlets (and even of available commands) is presented. The right list box will be used to choose the appropriate action if a command-action binding is to be established. In our case, an outlet connection from the main window to the multi-line entry field via the outlet `clientWindow` shall be established. Select the appropriate item (`clientWindow`) in the *Outlets* list box and press *Ok*. This will cause the outlet connection to be established.

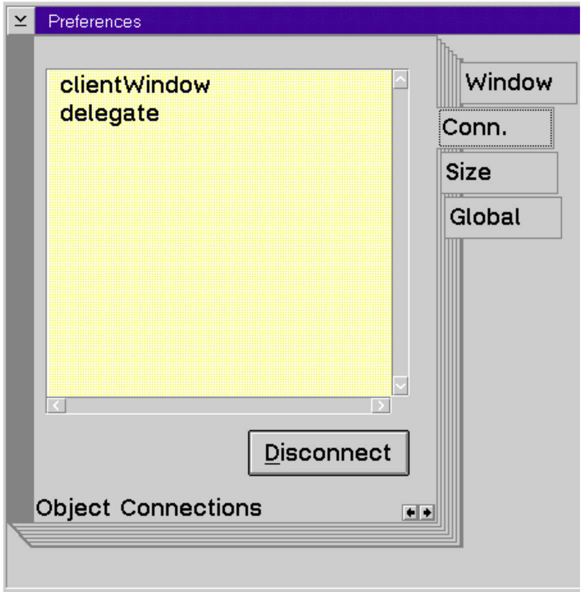

Figure 3.19: *Object Connections* page in the *Preferences* window

The list box will display all available outlet instance variables and commands. You can differ these two kinds of items by the way they are presented. Any variable representing a command is put between brackets, so if `delegate` was a command instead of an outlet, it would be displayed as `(delegate)`. Variables already connected are preceded by a `*`. The next time you will try to make a connection originating from the main window, the first item in the list box to the left will be displayed as `*clientWindow`. To change a connection, you can simply overwrite it by choosing it when making another connection.

To determine which outlet instance variables or commands of the selected objects are already connected to some target object, have a look at the *Object Connections* page of the *Preferences* window. Using this page—as shown in Figure 3.19—you can also delete connections established previously.

In Appendix A on page 41 a list of all outlet instance variables, commands and action methods supported by the various classes is given.

3.13 Removing Object Connections

Deleting a connection is achieved by using the *Preferences* window (see Figure 3.19). All connections are displayed on a page labelled with *Object Connections* (the tab text of this page is *Conn.*).

On this page a list of all available outlets and commands for the selected object (the source object) is displayed. Just as in the dialog displayed when connecting objects (Figure 3.18) commands can be identified by having their name enclosed in brackets. Again, already connected outlets or commands are denoted by a leading `*` in the list.

Using the *Disconnect* button located at the bottom of the page you can delete the currently selected connection.

Be careful when deleting a connection, there is no *undo* function for this action at the moment.

You do not have to delete a connection before connecting an instance variable or a command to another target object. By connecting a variable or command again the previous connection is deleted automatically.

3.14 Updating Class Information

When changing some class definitions using the Project Editor application while editing an interface file, you will have to update the information Interface Editor has of the user-defined classes manually. To perform an update simply choose *Reload Classes* from the *File* menu after saving the project data in Project Editor.

In future versions of this development suite I will try to integrate the different tools more closely so that these actions will be performed automatically.

3.15 Using Interface Files

To use an interface file in your applications, you have to unarchive the objects from the *Objective C Typed Stream* they are stored on.

The simplest way to achieve this is by using the `-loadIBFile:` method of the class `StdApp`. This method will return an instance of `InterfaceFile` you can use to access the objects stored in the interface file. The only access method you should use is `-objectWithTitle:` to query a pointer to one of the stored objects.

The connections you create visually will be set up automatically when loading the interface file.

When using Project Editor to create a project, a main implementation file called "`main.m`" will be created automagically. This file will load an interface file named "`main.oib`" on startup.

Chapter 4

The Database Editor Utility

When using the database library, and you want to create empty database files, there are two possible ways, you can go. First, you can create an empty file from scratch using the appropriate methods of `DBFile`, or you can simply create an empty file (a template) by using this Presentation Manager application.

Figure 4.1 shows the main window of the Database Editor program. A listbox is displayed, currently not storing any items. There the data fields will be displayed.

Figure 4.2 on the following page displays the dialog box used to add new data fields to the database file.

At the moment, only new database files can be created. There is no way to edit the field structure of an existing file. This will be implemented in the next version. So be careful when creating a database file!

As this application is simple to use, the functions are not described in more detail.

The only thing worth mentioning are the data types provided and the length of the data fields, the user can choose.

4.1 Data Types

The library currently supports the data types *boolean*, *character*, *date* and *numeric*. Additionally, the Database Editor allows you to create *memo* fields.

Memo fields require a separate file, where the data is stored. This file is *not* created by the program.

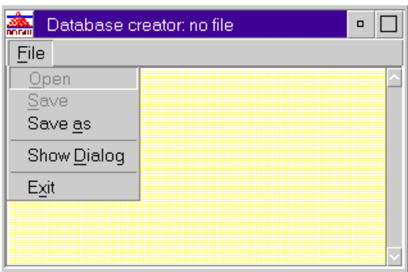

Figure 4.1: Database builder main window

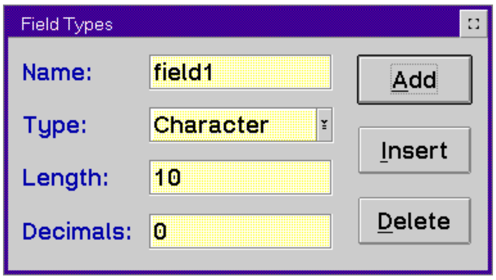

Figure 4.2: Dialog box used to add new fields

4.2 Length of Fields

The field `Length:` is used to specify the total length of the field currently selected. This also includes decimals (for numeric fields) and, if decimals are used, also one character for the comma.

Date fields always have a length of 8, decimals is 0.

Memo fields have a length of 10, no decimals.

Character fields are not allowed to have decimals.

Boolean fields (logic fields) have a length of 1, without decimals.

Chapter 5

Displaying DBase Database Files using DBView

The program *DBView* was at first developed as a sample program demonstrating how to use database files together with the `Container` class. The source code can be found in the directory `\usr\samples\dbview`. As I think this program can be quite useful in everyday (computer) life, it was also put into the binary directory together with the other development tools.

Its purpose is to simply display DBase III database files in a Presentation Manager window. All data fields currently supported by the database library, *character fields*, *numeric data*, *logical fields* and *date fields* will be displayed.

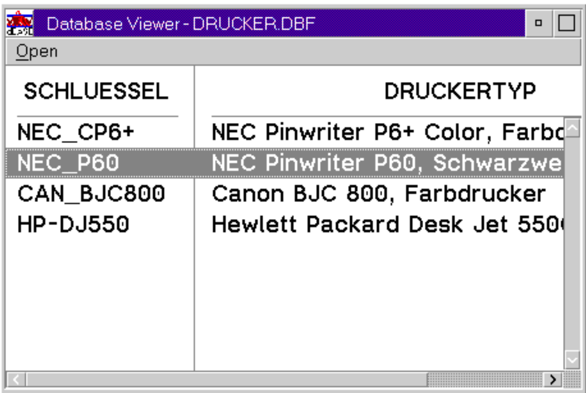

Figure 5.1: DBView displaying a simple DBase database file

The data itself will be displayed in form of a table. Every single record in the database file is displayed in a line, every column represents a data field. The data is displayed using the fields' `-stringValue` method resulting in some awkward format for e.g. date fields.

Figure 5.1 depicts the program window displaying a simple database file. The only menu item, *Open*, will cause a file dialog to be displayed where you can choose the database file—ending in "`.dbf`"—you wish to display.

You can also pass a single parameter to *DBView* when starting the program, to load a database file on startup.

Type

```
dbview file.dbf
```

which causes to automatically load and display the file "`file.dbf`" when starting the program.

It is recommended to set up an *association* for all files ending in "`.dbf`" with this program using the program object's settings notebook. This will let you start the program and display a data file by simply double-clicking a data file.

Feel free to extend the source code of the program and add more functionality. Adding capabilities for e.g. *editing* or *sorting* will be some nice programming exercises.

Chapter 6

Displaying Your Program's Debugging Output Using Console

To simplify quick & dirty debugging of your programs withoud using the GNU debugger, a program called *Console* is provided. This program will display a window and read data from a pipe your programs can write to. Using this program you can e.g. redirect your applications output files `stdout` and `stderr` to a Presentation Manager window.

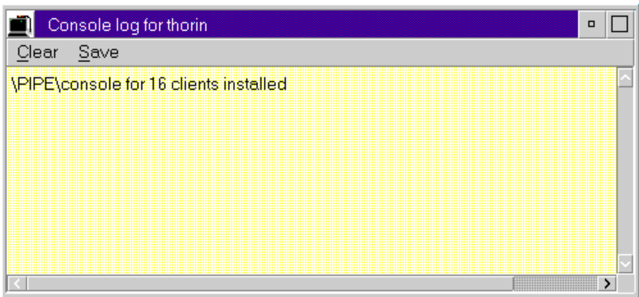

Figure 6.1: "Console" window

When started a window similar to the one depicted in Figure 6.1 will be displayed. You can see two menu items in the menu bar called *Save* and *Clear*. *Save* will store the current size and position of the console window in your system configuration file "`os2.ini`". *Clear* will clear the display area of the window.

In the title bar of the console window the name of your host system will be displayed. It is taken from the environment variable `SYSTEMNAME`. Use something like

```
SET SYSTEMNAME=mysystem
```

in your "`config.sys`"-file to set the name of your computer system.

By default, the program will read from the file `\PIPE\Console` and display everthing written there in the window. At most `16` client processes can write to this file at once.

To specify more client processes or another name for the pipe, use the command line options as shown below

```
console [-c count] [-n name]
```

`count` is the number of processes granted simultaneously access to the pipe and `name` is the name of the pipe file.

To start the console program listening on the file `\PIPE\p1` for suitable for at most `30` clients type

```
console -c 30 -n \PIPE\p1
```

on the command prompt.

To redirect your program's output files `stdout` and `stderr` to the console window, type

```
prog > \PIPE\Console 2> \PIPE\Console
```

when starting the program `prog.exe`. Every output your program does using

- `printf()`,
- `fprintf(stdout,...)` or
- `fprintf(stderr,...)`

will be displayed in the console window. This can be quite useful for quickly debugging your application without the need to start a full-featured debugger.

Appendix A

List of Action Methods, Commands and Outlet Variables

This chapter will list all pre-defined classes currently supported by the Interface Editor application. For every class the action methods and the outlet instance variables are given. For a description of these variables and methods see the *Reference Manual*.

Every classes' description structured as follows:

1. The *name* of the class, e.g.

   `EntryField`

   will denote that a description of the class `EntryField` will be presented.

2. Coming next is the list of *action methods*, e.g.

   `-pasteSelection:`, `-cutSelection:`, `-copySelection:`, `-clearSelection:`

3. The list of *commands* will enumerate all commands a single object can issue. These commands can automatically call action methods. Mostly this line is empty.

4. The last line will enumerate all *outlet instance variables*. For the `EntryField` class, this would simply be

   `delegate`

AutoCheckBox

Actions: —

Commands: `command`

Outlets: `delegate`

AutoRadioButton

Actions: —

Commands: `command`

Outlets: `delegate`

AutoTriStateButton

Actions: —

Commands: `command`

Outlets: `delegate`

CheckBox

Actions: —

Commands: `command`

Outlets: `delegate`

ComboBox

Actions: —

Commands: —

Outlets: `delegate`

Container

Actions: `-deselectAll:`, `-detailView:`, `-hideColumn:`. `-hideNotSelectedRecords:`, `-hideRecord:`, `-hideSelectedRecords:`, `-iconView:`, `-nameView:`, `-selectAll:`, `-showAllColumns:`, `-showAllRecords:`, `-showColumn:`, `-showRecord:`, `-sortByTitleWithoutCase:`, `-sortByTitleWithCase:`, `-textview:`, `-treeView:`

Commands: —

Outlets: `delegate`

EntryField

Actions: `-clearSelection:`, `-copySelection:`, `-cutSelection:`, `-pasteSelection:`

Commands: —

Outlets: `delegate`

Help

Actions: `-associateWith:`, `-helpExtended:`, `-helpFor:`, `-helpForHelp:`, `-helpIndex:`

Commands: —

Outlets: —

ListBox

Actions: —

Commands: —

Outlets: `delegate`

MainWindow

Actions: `-centerOnScreen:`, `-makeKeyAndOrderFront:`, `-performClose:`, `-performQuit:`

Commands: —

Outlets: `clientWindow`, `delegate`

MultiLineEntryField

Actions: `-clearAll:`, `-clearSelection:`, `-copySelection:`, `-cutSelection:`, `-pasteSelection:`

Commands: —

Outlets: `delegate`

NoteBook

Actions: —

Commands: —

Outlets: `delegate`

Printer

Actions: `-jobProperties:`, `-printerSetup:`

Commands: —

Outlets: `setupDialog`

PushButton

Actions: —

Commands: `command`

Outlets: `delegate`

RadioButton

Actions: —

Commands: `command`

Outlets: `delegate`

ScrollBar

Actions: —

Commands: —

Outlets: `delegate`

Static

Actions: —

Commands: —

Outlets: `delegate`

StdDialog

Actions: `-centerOnScreen:`, `dismiss:`, `-dismissOK:`, `-makeKeyAndOrderFront:`, `-performClose:`, `-performQuit:`, `-runModalFor:`

Commands: —

Outlets: `clientWindow`, `delegate`

StdWindow

Actions: `-centerOnScreen:`, `-makeKeyAndOrderFront:`, `-performClose:`, `-performQuit:`

Commands: —

Outlets: `clientWindow`, `delegate`

TriStateButton

Actions: —

Commands: `command`

Outlets: `delegate`