OS/2 Objective C Libraries

Version 0.8

Tutorial

February 1996

Thomas Baier

baier@ci.tuwien.ac.at

Abstract

This manual is an introduction into OS/2 PM programming using Objective C and this class library. In addition to explaining the basics of the PM class library, an overview of the database library and the more generic classes is provided. In later sections, the development tools supporting some kind of *visual programming* are introduced.

Before beginning to use this library in application development, you should read this manual carefully. Most of the sample programs are explained here in detail, a fact that can save you lots of time from studying the source code of the samples itself.

At the end of this document, you can find some recommendations, which books to read, if you have any specific questions concerning *Objective C*, or *OS/2 Programming*.

If you already know the principles on which the class libraries are built on, you might better look into the Reference Manual for specific information.

If you are searching for specific information concerning

- *Installation* ··· Read the Installation Manual.
- *Basics of Application development* ··· Read the appropriate sections in this manual, the Tutorial. Here you can find a gentle introduction into using this library package for developing OS/2 PM applications.
- *Classes and Methods provided by the libraries* ··· You can find special information about the provided classes and methods in the Reference Manual.
- *References on the graphical development tools (database editor, interface editor, project editor)* ··· Read the appropriate sections in the Application Programming Tools Manual.
- *Literature* ··· Look in the Literature section of this Manual.

Contents

List of Figures

Chapter 1

Introduction

Programming OS/2 PM applications is mostly done using the programming language C. Because the OS/2 application programming interface (*API*) is in most parts object oriented, more and more programmers choose an object oriented programming language for their purposes. The most popular object oriented programming language for development of OS/2 applications today is *C++*.

Because of the mostly static binding and its nearly completely missing run-time system many people are searching for easy-to use alternatives to C++. One of the most popular alternatives in object oriented programming to C++ is *Smalltalk*. Due to its features, such as dynamic binding, messaging, etc., it is better suited for developing complex applications using a graphical user interface with PM.

There is another object oriented programming language, which is as easy to learn as pure C (because it is not much more than C itself), but supports dynamic binding just like Smalltalk does. This language is *Objective C*.

Objective C only adds some few new features to its "father" C, so it is an easy to learn language for C programmers.

Another advantage of Objective C is that an Objective C compiler is part of GCC, the GNU C compiler.

So—get it and start developing native OS/2 32bit programs using Objective C.

This document is a simple—not yet complete—tutorial, showing you how to start using this library package, and also showing some of the basic classes provided. It is by no means a reference manual, if you're searching for some special information, look in the *Reference Manual* which you should also have received. In addition to an introduction into the concepts of the libraries themselves, later chapters will show an alternate way of developing OS/2 PM applications using Objective C. By making use of the development tools shipped together with the libraries some kind of *visual development* can be performed.

Therefore freed from routine jobs, just as writing a Makefile, the programmer can concentrate on his "real" job, writing a simple and easy-to-use application. By support of rapid prototyping of the graphical user interface on the one hand, and even the classes used by the application and the creating and linking of objects on the other hand, OS/2 applications can be created within some minutes. Fully utilizing the existing tools, your application is just a few mouse clicks away...

In the end, some hints on debugging Objective C programs are given. Be sure to read this section, this can save you a lot of troubles when developing applications.

This document assumes the reader is familiar with the programming language C, and—to some extent—of Objective C. There are many good books on the C language, but the situation for the

Objective C language seems to be a bit worse—not to say catastrophic. Sources of information can be found in the *Objective C FAQ* regularly posted in `comp.lang.objective-c`. There you can also find a gentle introduction into this object-oriented flavour of C and many pointers to sources of information.

It must be stated, that writing multi-threaded applications using Objective C is not that simple at the moment. The Objective C runtime system is *not* thread-safe now, so only one thread (normally the main thread) is allowed to use the runtime library (and therefore Objective C objects). As this problem is worked on at the moment, I am sure writing multi-threaded apps will be possible soon.

Chapter 2

Writing a Simple PM Application

Programming OS/2 Presentation Manager can be a quite hard job, if you rely on pure C and the OS/2 API functions. This is why I developed this class libraries. As you will see throughout this and the following chapters, using Objective C normally spares you the time to read the complex documentation of the OS/2 Application Programming Interface. There are just some basics you should know before starting development.

Before doing any *real work* your program must do some initialization, which means it has to allocate all necessary resources to run and it has to *register* itself at the graphical subsystem of OS/2, the Presentation Manager (PM).

After the program is run, all resources must be freed again.

So, lets look at a simple PM application written using C

2.1 Application Main Function

```
#define INCL_PM
#include <os2.h>
.
.

main ()
{
  HAB  hab;   /* handle to the anchor block of the application */
  HMQ  hmq;   /* handle to the main message queue of the appl. */
  QMSG qmsg;  /* message structure */

  hab = WinInitialize (0);       /* register application at PM */
  hmq = WinCreateMsgQueue (hab,0);/* create main message queue */

  .
  . /* other initialization, allocate resources, ... */
  .

  while (WinGetMsg (hab,&qmsg,(HWND) NULL,0,0))
    WinDispatchMsg (hmq,&qmsg);        /* process all messages */

  . /*
  .  * free all allocated resources,
  .  * prepare application to terminate
```

```
  .  */

  WinDestroyMsgQueue (hmq);      /* destroy main message queue */
  WinTerminate (hab);               /* de-register application */
}
```

This application skeleton shows the necessary steps, a program has to go through to be run with OS/2 Presentation Manager.

1. *Initialization:* registration at PM, create message queue, ...
2. *Message loop:* receive all messages for the application and process them
3. *Cleanup:* destroy message queue, de-register application, ...

The Objective C PM class library provides a class, called `StdApp` to meet the purpose of standard initialization and message processing for every PM application. The following piece of source code demonstrates how to use it:

```
#include <pm/pm.h>
.
.
main ()
{
  StdApp *application; /* pointer to our instance
                           of a StdApp class */

  application = [StdApp alloc]; /* create application object */

  [application init]; /* initialize application */
  .
  .
  .
  [application run];  /* process all messages */
  .
  .
  .
  [application free]; /* free application object */
}
```

As you can see the first line of the sample includes `<pm/pm.h>`. This include file causes all include files of the PM class library to be read in. After doing this, you can use all classes of the library and their methods without any restrictions.

And here a more compact version of the same piece of code:

```
#include <pm/pm.h>
.
.
main ()
{
  StdApp *application = [[StdApp alloc] init];
  .
  .
  .
  [application run];
  .
  .
```

```
  .
  [application free];
}
```

As you can see using this class library can really simplify your life. Instead of creating and initializing dozens of local or—even worse—global variables, you simply allocate and initialize a single object.

2.2 A Simple Application

To demonstrate the complexity of a complete PM application written in pure C, I will show you a program that just creates a standard window, waits until this window gets closed by the user and then terminates its operation. At first, again the standard C version is shown, only using OS/2 API functions:

```
#define INCL_PM
#include <os2.h>

#define NEWCLASSNAME "NewClass"

MRESULT EXPENTRY windowFunction (HWND hwnd,ULONG msg,
                                 MPARAM mp1,MPARAM mp2)
{
  switch (msg) {
  case WM_ERASEBACKGROUND:
    return (MRESULT) FALSE;
  default:
    return WinDefWindowProc (hwnd,msg,mp1,mp2);
  }
}

main ()
{
  HAB   hab;
  HMQ   hmq;
  QMSG  qmsg;
  HWND  mainWindow;
  HWND  clientWindow;
  ULONG createFlags;

  hab = WinInitialize (0);
  hmq = WinCreateMsgQueue (hab,0);

  WinRegisterClass (hab,NEWCLASSNAME,windowFunction,0L,0);

  createFlags = FCF_SYSMENU | FCF_TITLEBAR | FCF_MINMAX |
                FCF_SIZEBORDER | FCF_SHELLPOSITION |
                FCF_TASKLIST;

  mainWindow = WinCreateStdWindow (HWND_DESKTOP,
                                   WS_VISIBLE,
                                   &createFlags,
                                   (PSZ) NEWCLASSNAME,
                                   (PSZ) "",
                                   0L,
                                   NULLHANDLE,
```

Figure 2.1: Sample application “simple_o.exe”

```
                                   1000,
                                   &clientWindow);

  while (WinGetMsg (hab,&qmsg,(HWND) NULL,0,0))
    WinDispatchMsg (hab,&qmsg);

  WinDestroyWindow (mainWindow);

  WinDestroyMsgQueue (hmq);
  WinTerminate (hab);
}
```

The following piece source code shows how much simpler it is to use the PM class library than working with “normal” OS/2 PM API functions.

```
#include <pm/pm.h>

main ()
{
  StdApp     *application = [[StdApp alloc] init];
  MainWindow *mainWindow = [[MainWindow alloc]
                           initWithId: 1000
                             andFlags: (FCF_SIZEBORDER | FCF_SHELLPOSITION |
                                        FCR_TITLEBAR | FCF_SYSMENU |
                                        FCF_MINMAX | FCF_TASKLIST)];

  [mainWindow makeKeyAndOrderFront: nil];
  [application run];

  [mainWindow free];
  [application free];
}
```

In addition to inititializing an application object, the main window is created as an instance of MainWindow. The OS/2 window identifier is 1000, the window is created with a resizable border.

Calling the method -makeKeyAndOrderFront: shows the window.

Figure 2.1 shows the window created by this simple piece of source code.

2.3 Necessary Include Files

To use the OS/2 PM class library simply include the file <pm/pm.h> into your application. This automatically includes all Objective C *Interface Files* and the patched OS/2 API header file <objc/os2.h>. The patches are based on the files \emx\include\os2.h from *emx0.9b*.

When using the Database library, you have to include <db/db.h>. To use any of the supporiting classes found in the utility library, just include <util/util.h>.

After installing the libraries, these include files can be found in the directories \usr\include\pm, \usr\include\db, \usr\include\util respectively \usr\include\objc.

2.4 Compilation

To compile programs using the PM class library just link the executable file with the class library file, the utility library *and* the Objective C runtime library. The utility library is *always* needed. Some classes of either the PM library and the database library will use classes of the utility library (objcutil), this library therefore must be linked to all of your programs just as the Objective C runtime library.

The above examples (C and Objective C version) can be found in \usr\samples\simple. To compile the C version, type

- gcc -c simple_c.c ··· to produce the object file simple_c.o.
- gcc -o simple_c.exe simple_c.o ··· to produce the executable file simple_c.exe.
- emxbind -ep simple_c.exe ··· to set the application type for simple_c.exe to *OS/2 Presentation Manager Application*.

To compile the—noticable shorter and simpler—Objective C version of the program, use

- gcc -c simple_o.m ··· to produce the object file simple_o.o.
- gcc -o simple_o.exe simple_o.o -lobjcpm -lobjcutil -lobjc ··· to produce the executable application file simple_o.exe.
- emxbind -ep simple_o.exe ··· to set the application type for simple_o.exe to *OS/2 Presentation Manager Application*.

See Figure 2.1 on the facing page for the output produced by the simple application. The look and feel of the window was determined by the FDS_xxxx flags specified at window creation (see the reference manual for more information concerning these flags).

After linking and setting the application type you can strip all debug symbols off the executable file by using the -s option of emxbind. Using the command emxbind -s simple_c.exe respectively emxbind -s simple_o.exe will strip all debug information from the executable files. It must be stated that the Objective C version is significantly larger than the pure C version. In this simple case, the Objective C version `simple_o.exe` is about 15 times larger in size than `simple_c.exe`. This is caused by the large overhead of the statically linked libraries `objcpm`, `objcutil` and `objc`. The larger your programs get—no one will call such a small program a *useful* application—the difference in size between the Objective C and C versions will decrease. So, don't be afraid of creating too large applications. The library overhead—assuming statically linking the applications—will always be only about 150K.

Normally it's better to use a makefile for compiling and linking applications. A sample makefile is provided in `\usr\samples\make`. Just copy the two files `makefile.preamble` and `makefile` to your source code directory and fill in the blanks in `makefile`. For a description of how to do this, see section 3.7 on page 18.

Later in this document it will be shown, how to simplify even these tasks of *project management*. The makefiles for your programs can be created for you by one of the graphical development tools, the *Project Editor*.

Chapter 3

A Simple File-Browser

This chapter describes a simple application, which already can provide quite useful. Its purpose is to read a text file and display it in an OS/2 PM window. The name of the text file is given as the first and only parameter at the command line. The program itself will be called `textview`.

To accomplish such a behaviour, one of the classes of the PM library—capable of displaying various amounts of text by itself—will be used. Information on the available classes in general, and on the *Multi-Line entry field* (MLE) used here, can be found in the reference manual.

The window should be resizable and its contents area (the MLE window) should have the same size as the window itself.

If you, for example, want to take a look at your main OS/2 configuration file, just type `textview c:\config.sys`. The file will be loaded and displayed.

Figure 3.1 shows the application main window displaying the source code of the program itself.

3.1 Parts of the Program

As shown before, the program consists of three parts, *Initialization*, the *Message loop* and a *Cleanup* section.

3.1.1 Initialization

The first section, *Initialization*, has to do the following:

- Check for the command line parameters. There must be *exactly one* parameter when starting the program, which is the name of the file to be displayed.
- Check, if the file exists; create a buffer area in memory with enough size to store the contents of the whole file.
- Read the data stored in the file into the buffer area.
- If all is o.k., create the application instance and a window. Insert a *multi-line entry field* into the window, where the text will be displayed.
- load the text buffer in memory to the display area of the multi-line entry field.[1]

[1] It must be noted that only about 64KBytes of text can be copied to the MLE window at once. Using more data would require the data to be split in 64KByte blocks. As this is not performed in this simple application, the file size is limited to 64KBytes.

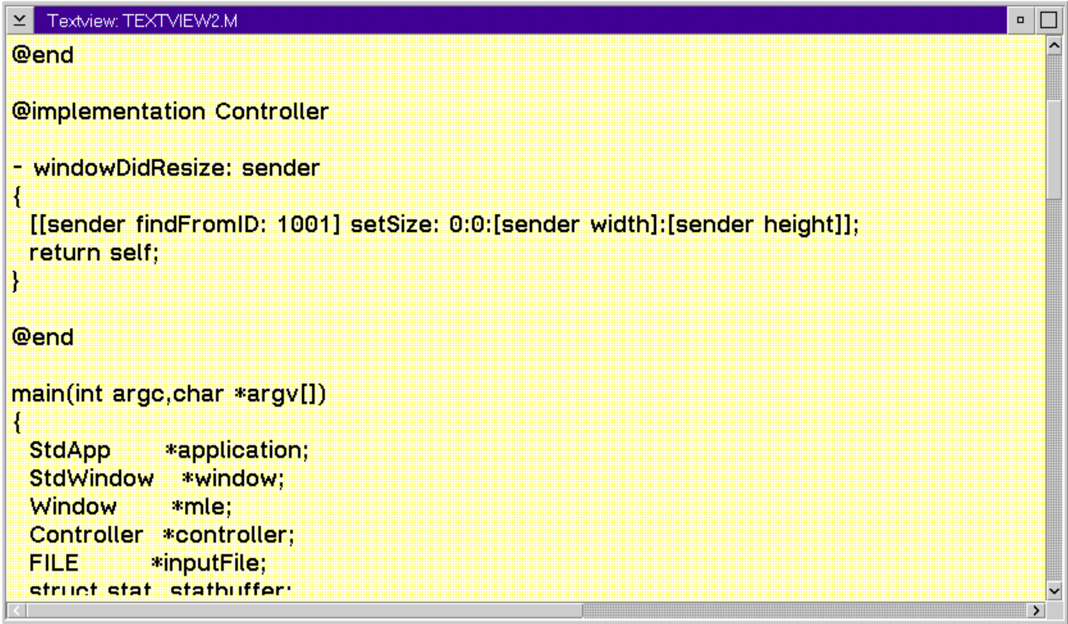


Figure 3.1: “Textview” application displaying its own source code

The first three sections of the initialization don’t have anything to do with this class library. They only use functions of the EMX C-Library and are simple to understand:

```
main(int argc,char *argv[])
{
  FILE        *inputFile;
  struct stat  statbuffer;
  char        *contents;

  /*
   * check for command line arguments and
   * check given file (struct stat)
   */
  if (argc != 2) /* check for command line arguments,
                    must be exactly one */
    exit (-1);

  if (stat (argv[1],&statbuffer) < 0) /* check file */
    exit (-1);

  /*
   * open file and read contents to buffer
   */
  inputFile = fopen (argv[1],"r"); /* open text
                                      file read-only */

  contents = (char *) malloc (statbuffer.st_size + 1);
             /* allocate buffer */
  fread (contents,statbuffer.st_size,1,inputFile);
             /* read contents of file */
```

`inputFile` is a pointer to a file structure returned by `fopen()`. `statbuffer` is used to retrieve information about the file using the C-Library function `stat()`. Here the size of the file is stored.

After reading file information, `contents` is allocated via `malloc()` and the file is opened and its contents are read to `contents`.

For more documentation on the C library functions used here, either check a book concerning the programming language C and the standard library functions as available on most *nix systems, or consult the online reference manuals to the emx C library (normally found in `\emx\book`.

Following this part of the code, the initialization of the used PM classes takes place.

Just add some more variable declarations to the first section of the code:

```
StdApp      *application;
StdWindow   *window;
Window      *mle;
char        *title;
```

`title` is used as a buffer area to store the title of the main window, where the text will be displayed, `mle` is a pointer to a generic window object, which will be initialized as a `MultiLineEntryField`. `application` and `window` will hold pointers to the instances of the main application object and the main window respectively.

The initialization of these variable is performed as shown below:

```
/*
 * create app instance and window,
 * create MLE for text display
 */
application = [[StdApp alloc] init]; /* initialize
                                        application
                                        object */
window = [[MainWindow alloc] initWithId: 1000
                               andFlags: (FCF_SIZEBORDER | FCF_TITLEBAR |
                                          FCF_SYSMENU | FCF_MINMAX |
                                          FCF_SHELLPOSITION | FCF_TASKLIST)];
             /* create main window */

[window createObjects]; /* create child windows
                           of main window */

mle = [[MultiLineEntryField alloc]
          initWithId: 1001
            andFlags: (WS_VISIBLE | MLS_READONLY |
                        MLS_HSCROLL | MLS_VSCROLL)
                  in: window];
[window insertChild: mle]; /* insert MLE into window */

/*
 * calculate title of window and set it
 */
title = (char *) malloc (11 +  /* allocate buffer for title */
                         strlen (argv[1]));
sprintf (title,"Textview: %s",argv[1]); /* fill title buffer */

[window setTitle: title]; /* set window title */

free (title); /* free title buffer */
```

This section of code creates and initializes the application object and creates a main window with PM identifier `1000`.

Afterwards all existing child objects of the window are created in memory using -createObjects. Then a PM MLE window is created (id 1001) and inserted into the main window.

The last part of the code simply allocates memory to hold the title string and creates the title string, which consists of the name of the application (Textview) and the name of the file to be displayed.

The MLE window is created in *read-only* mode with a horizontal and a vertical scrollbar (flags MLS_READONLY, MLS_HSCROLL and MLS_VSCROLL).

After creating the objects and initializing the variables the main window is displayed and the size of the MLE window is adjusted to the size of the main window, to fill its complete interior:

```
/*
 * show window, set MLE size and  display contents of file
 */
[window makeKeyAndOrderFront: nil]; /* show window */
[mle setSize: 0:0:[window width]:
                  [window height]]; /* set MLE size */
[mle setText: contents]; /* display contents of file */
```

This piece of code also sets the text displayed in the MLE window to be the buffer area contents.

3.2 Message Loop

The main message loop is started by calling [application run]. As mentioned before, this method terminates, when the main window gets closed.

3.3 Cleanup

After the window was closed, all objects are destroyed and the previously allocated buffer area is freed again:

```
/*
 * free all resources
 */
free (contents); /* free contents buffer */
fclose (inputFile); /* close file */

[application free]; /* free application */
[window free]; /* free window */
```

Note, that [window free] automatically destroys all its child windows, in our case, also the MLE window.

If you encountered any problems in understanding the code presented above—especially if the concepts of messaging in Objective C still seem to be a bit obscure—consult one of the sources of information recommended at the end of this document.

3.4 Compilation

To compile this application, store the code shown in the following subsection in the file textview.m (it can be found in \usr\samples\textview) and type:

```
gcc -c textview.m
gcc -o textview.exe textview.o -lobjcpm -lobjcutil -lobjc
emxbind -ep textview.exe
```

For comfort a makefile is provided in every directory containing sample code, so you won't have to bother typing in the command lines starting compilation and linking.

3.4.1 Complete Source Code of "textview.m"

```
#include <pm/pm.h>
#include <io.h>
#include <sys/types.h>
#include <sys/stat.h>

main(int argc,char *argv[])
{
  StdApp      *application;
  StdWindow   *window;
  Window      *mle;
  FILE        *inputFile;
  struct stat  statbuffer;
  char        *contents;
  char        *title;

  /*
   * check for command line arguments and
   * check given file (struct stat)
   */
  if (argc != 2) /* check for command line arguments,
                  must be exactly one */
    exit (-1);

  if (stat (argv[1],&statbuffer) < 0) /* check file */
    exit (-1);

  /*
   * open file and read contents to buffer
   */
  inputFile = fopen (argv[1],"r"); /* open text
                                      file read-only */

  contents = (char *) malloc (statbuffer.st_size + 1);
             /* allocate buffer */
  fread (contents,statbuffer.st_size,1,inputFile);
             /* read contents of file */

  /*
   * create app instance and window,
   * create MLE for text display
   */
  application = [[StdApp alloc] init]; /* initialize
                                          application
                                          object */
  window = [[MainWindow alloc] initWithId: 1000
                                 andFlags: (FCF_SIZEBORDER | FCF_TITLEBAR |
                                            FCF_SYSMENU | FCF_MINMAX |
                                            FCF_SHELLPOSITION | FCF_TASKLIST)];
```

```
               /* create main window */

  [window createObjects]; /* create child windows
                             of main window */

  mle = [[MultiLineEntryField alloc]
            initWithId: 1001
              andFlags: (WS_VISIBLE | MLS_READONLY |
                          MLS_HSCROLL | MLS_VSCROLL)
                    in: window];
  [window insertChild: mle]; /* insert MLE into window */

  /*
   * calculate title of window and set it
   */
  title = (char *) malloc (11 +  /* allocate buffer for title */
                           strlen (argv[1]));
  sprintf (title,"Textview: %s",argv[1]); /* fill title buffer */

  [window setTitle: title]; /* set window title */

  free (title); /* free title buffer */

  /*
   * show window, set MLE size and  display contents of file
   */
  [window makeKeyAndOrderFront: nil]; /* show window */
  [mle setSize: 0:0:[window width]:
                    [window height]]; /* set MLE size */
  [mle setText: contents]; /* display contents of file */

  /*
   * run application
   */
  [application run];

  /*
   * free all resources
   */
  free (contents); /* free contents buffer */
  fclose (inputFile); /* close file */

  [application free]; /* free application */
  [window free]; /* free window */
}
```

If you compile and link this program (just type `make` in the appropriate directory) you will see, that although the main window is resizable, the MLE window inside the window remains the same size, no matter what size its parent window is of.

The rest of this chapter will show how an object can be automatically notified, e.g. when the main window resizes, to be able to adapt the size of the MLE window as expected by the user of the application.

Always keep in mind, that only a little fraction of the classes services are mentioned here. For full information read the appropriate sections on every class in the reference manual. Instances of `MainWindow` will not only notify an object of a change in its size, but also of, e.g., an action causing the window being moved, closed, etc.

3.5 Delegate Objects

One of the major advantages of the Objective C language when compared to most other object-oriented programming languages is the possibility to check at runtime, if an object implements a specific method or if it does not. This provides a simple way for objects to send messages to other objects, if these messages can be processed, whenever an object thinks another object has to be notified of some special occurrence.

An object implementing methods called by another object, to be notified of some special events, is called a *delegate object*.

So it is possible to create classes, and thereafter objects of these classes, which can change one predefined classes behaviour without the need of subclassing one of the predefined classes.

Delegation is used by some objects in this library—mostly by window objects at the time. The information on which specific *delegate methods* are supported by an object of some class can be found in the reference manual. In the documentation for every class supporting delegate methods—if this is not only inherited behaviour from its superclass—you can find a section called *Methods implemented by the delegate*. Any of these methods is called *automatically* by an object if the delegate implements this special method and there is need to inform another object of some special occurence, e.g. when a window is resized or closed.

Using the method `-setDelegate:` you can assign a special object, implementing some *delegate functions*, as the delegate object of an instance of any window class, for example to an instance of `MainWindow` or `StdDialog`.

If the delegate object implements any of the methods described in the section *Methods implemented by the delegate* which is part of some class descriptions in the reference part of this manual, these methods get called at the occurrences described there.

For our purposes, we will use the delegate method `-windowDidResize:`, which is called whenever the window gets resized by the user or the application program.

This method will then query the size of the sending instance of `StdWindow` and accustom the size of the MLE window according to this.

3.6 Implementing the Delegate

First we have to define a new class, implementing the method `-windowDidResize:`. The class declaration is quite simple:

```
@interface Controller : Object
{
}

- windowDidResize: sender;

@end
```

This declaration defines a new class, a subclass of `Object`, called `Controller`, which has no new instance variables but those inherited from its superclass and implements one method called `-windowDidResize:`.

The implementation of this simple class looks like this:

```
@implementation Controller

- windowDidResize: sender
```

```
{
  [[sender findFromID: 1001] setSize:
         0:0:[sender width]:[sender height]];
  return self;
}

@end
```

This is a simple method, just calling some methods of `sender` and of the previously created MLE window.

By calling `[sender findFromID: 1001]` the method queries a pointer to an instance of `Window` or one of its subclasses. This window must be a child window of `sender` and have the OS/2 PM identifier `1001`.

This method returns a pointer to the MLE window's associated Window object. This method is sent a `-setSize::::` message to adapt its size to the size of the sending window.

`-setSize::::` takes the coordinates of the lower left corner of the window (the first and second parameter) relative to its parent's lower left corner. The last two parameters represent the *width* and *height* the window should be resized to.

The lower left corner of the MLE window should be the same as the lower left corner of its parent, `(0/0)`. The width and height of the MLE window is queried from the sender by using the appropriate methods `-width` and `-height`.

As this method has a return type of `id`[2], `self` is returned on successful execution of the method.

The following section shows the modified source code of `textview.m`, which is stored in the directory `\usr\samples\textview` under the name `textview2.m`.

3.6.1 Modified Version of Textview: "textview2.m"

```
#include <pm/pm.h>
#include <io.h>
#include <sys/types.h>
#include <sys/stat.h>

@interface Controller : Object
{
}

- windowDidResize: sender;

@end

@implementation Controller

- windowDidResize: sender
{
  [[sender findFromID: 1001] setSize:
         0:0:[sender width]:[sender height]];
  return self;
}

@end
```

[2] id is a pointer to a generic Objective C object

```
main(int argc,char *argv[])
{
  StdApp      *application;
  StdWindow   *window;
  Window      *mle;
  Controller  *controller;
  FILE        *inputFile;
  struct stat  statbuffer;
  char        *contents;
  char        *title;

  /*
   * check for command line arguments
   * and check given file (struct stat)
   */
  if (argc != 2) /* check for command line arguments,
                    must be exactly one */
    exit (-1);

  if (stat (argv[1],&statbuffer) < 0) /* check file */
    exit (-1);

  /*
   * open file and read contents to buffer
   */
  inputFile = fopen (argv[1],"r"); /* open text file read-only */

  contents = (char *) malloc (statbuffer.st_size + 1);
                                   /* allocate buffer */
  fread (contents,statbuffer.st_size,1,inputFile);
                                   /* read contents of file */

  /*
   * create app instance and window, create MLE for text display
   */
  application = [[StdApp alloc] init]; /* initialize application
                                          object */
  window = [[MainWindow alloc] initWithId: 1000
                                 andFlags: (FCF_SIZEBORDER | FCF_TITLEBAR |
                                            FCF_SYSMENU | FCF_MINMAX |
                                            FCF_SHELLPOSITION | FCF_TASKLIST)];
                                   /* create main window */
  controller = [[Controller alloc] init];

  [window createObjects]; /* create child windows of
                             main window */
  [window setDelegate: controller];

  mle = [[MultiLineEntryField alloc]
           initWithId: 1001
             andFlags: (WS_VISIBLE | MLS_READONLY |
                        MLS_HSCROLL | MLS_VSCROLL)
                   in: window];
  [window insertChild: mle]; /* insert MLE into window */

  /*
   * calculate title of window and set it
   */
```

```
  title = (char *) malloc (11 + /* allocate buffer for title */
                           strlen (argv[1]));
  sprintf (title,"Textview: %s",argv[1]); /* fill title buffer */

  [window setTitle: title]; /* set window title */

  free (title); /* free title buffer */

  /*
   * show window and display contents of file
   */
  [mle setText: contents]; /* display contents of file */
  [window makeKeyAndOrderFront: nil]; /* show window */

  /*
   * run application
   */
  [application run];

  /*
   * free all resources
   */
  free (contents); /* free contents buffer */
  fclose (inputFile); /* close file */

  [application free]; /* free application */
  [window free]; /* free window */
  [controller free]; /* free controller */
}
```

3.7 Sample Makefiles

In the directory `\usr\samples\makefile` you can find a sample `makefile` together with the associated include file `makefile.preamble`.

To use this makefile, just copy `makefile` and `makefile.preamble` to your application directory and fill in the correct places in `makefile`.

1. Add the name of your application file to the line containing `APPLICATION =` (including the suffix `.exe`).
2. Add the names of your object files to the line containing `OBJECTS =`.
3. Add all OS/2 resource files (the files with extenstion `.res`) to the line containing the statement `RESOURCES =`.

This makefile was written for *GNU make*. Possible targets are:

- no target ⋯ this automatically compiles and links the application program
- *dep* or *depend* ⋯ check all files for dependencies and create a `.depend` file, which is automatically included.
- *clean* ⋯ removes all temporary files (compiled resources, application program, object files, core dump file, ...)

```
# Makefile for PM programs using Objective C class library

include Makefile.preamble

ifeq (.depend,$(wildcard .depend))
include .depend
endif

APPLICATION =
OBJECTS =
RESOURCES =

all: $(APPLICATION)

depend dep:
	$(CPP) -MM *.m > .depend

$(APPLICATION): $(OBJECTS) $(RESOURCES)
	$(CC) -o $(APPLICATION) $(OBJECTS) $(RESOURCES) \
	         -lobjcpm -lobjc
	emxbind -ep $(APPLICATION)
	$(STRIP) $(APPLICATION)

clean:
	rm -rf $(OBJECTS) $(RESOURCES) $(APPLICATION) core *~
```

If you plan to write applications from scratch, these makefiles can be of some value to you, sparing you the time of writing one every time you start a project. Later on the development tools shipped together with this libraries will be introduced, showing how to visually build an application, design its user interface and compile and link with a single mouse-click.

Chapter 4

Loading Resources

Using the OS/2 Resource Compiler `RC.EXE`, you can create a *binary resource file* from a *resource definition file*. This binary resource file can be linked to your application main module just like normal object files. The application can then load some of the *resource templates* instead of creating *dialog windows*, *menus* or many other window objects from scratch by creating and inserting window objects into a parent window.

In contrast to previous versions of this product using resource definition files is only needed to define menus—as well for “normal” pull-down menus for windows and dialog windows as for popup menus—and bitmap/icon resources. The preferred way of creating other user interface elements, just as dialog or main windows, is to use the Interface Editor program for visual development. Resource files should only be used for those elements currently not supported by the Interface Editor.

4.1 Adding a Menu Resource to Textview

Just for demonstration issues, I’d like to show how to add a simple menu resource to the main window (the only window) of the previously described Textview application.

Only one menu shall be added to Textview, a menu called *File*, which just includes the following menu items:

- *Open...* $\cdots$ to open and display a textfile
- *Exit* $\cdots$ to close the application window and exit

The definition of these menu items are as follows:

```
MENU 1000
{
  SUBMENU      "~File",                    2000
  {
```

Figure 4.1: Simple menu for “Textview”

```
    MENUITEM   "Open...",                  2001
    MENUITEM                               SEPARATOR
    MENUITEM   "Exit",                     2002
  }
}
```

The menu *File* has id 2000, the menu items *Open...* and *Exit* the ids 2001 respectively 2002.

Between the two menu items *Open...* and *Exit* a separator item should be inserted.

The resulting menu is shown in figure 4.1.

To load this menu, just create a resource definition file, type in the menu declaration and use RC.EXE to produce a binary resource file. When linking the application, do not forget to specify the name of the binary resource file (just like any other object file).

Note that one of the reasons your application will not behave as expected, e.g. nothing happens when starting the program although it seems to be running, can be that you just forgot to add the resource file to the linker command line. Any program creating a window and thereby specifying some resource flags, e.g. FCF_MENU, and the resource object could not be found in the executable file, the program just stops and waits, and waits ...

When creating the main window of Textview, add FCF_MENU to the given flags (FCF_SIZEBORDER, etc.). When creating the window, the menu resource will be loaded and displayed in the window's actionbar. Which menu will be loaded depends on the OS/2 PM identifier of the window, which you specify at window creation. It must be the same as the identifier specified in the resource definition for the menu (in our case, it is 1000).

For detailed information on the syntax of resource script files see the online documentation supplied with the IBM OS/2 developer's toolkit (or some other source of information, e.g. the OS/2 Redbooks).

4.2 Dialogs

Using a dialog editor, you can easily create dialog windows and either store a resource definition file or a binary resource file to disk.

Just like normal windows, dialog windows are created by the application using the appropriate dialog window class StdDialog. In addition to creating the window object, the contents of the dialog are loaded from the main resource file linked to the application.

After creation, dialog windows can be displayed using -makeKeyAndOrderFront:. In addition to normally displaying the dialog windows, which causes the dialog to run non-modal, you can also run a dialog modal for a given parent window. Using -runModalFor: the dialog window is displayed, but working with its parent window, which it runs modal for, is not possible until the dialog window gets closed again (*dismissed*).

4.3 Command Bindings

After a menu bar has been created, or a dialog window was loaded from a resource file, some of the menu items or window objects in the dialog send command messages to their owner. By processing these messages, the program can react to user actions.

Using the classes provided by this library, you can bind command messages to designated methods of an object. When a special command message was sent to a window, the appropriate method of an object gets called.

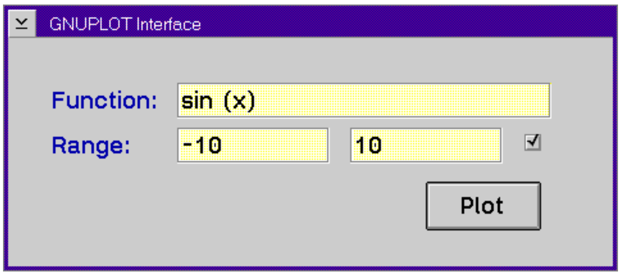


Figure 4.2: Simple PM interface to "Gnuplot"

4.3.1 Outlets and Actions

All methods which can be bound to command messages must be of the form `-nameOfMethod: sender`. The parameter `sender` stores a pointer to the sending instance of a `StdWindow` or a `StdDialog`, which calls the method.

In the following methods (messages) of this style are called *actions* if they are intended to be called from some other object as a reaction to some user action. In addition to this special kind of method, there are also some special instance variables. Every instance variable of type `id` which is intended to just store a pointer to another object—normally a delegate object of some kind—is called an *outlet*. Take care to remember these two terms, they will be the basis the visual development supported by the Interface Editor is built on.

Command messages can be bound to objects and appropriate methods using `-bindCommand: withObject: selector:`. The first parameter of this method is the identifier of the PM object, which posts command messages, the second is a pointer to the Objective C object, which implements the method to be called, the third and last is the *selector* of the method to be called. The selector of a method can be queried using `@selector(nameOfMethod)`.

To bind the command message sent by the menu item *Exit*, which has an OS/2 PM id of 2002 to the `-performClose:` method of the window object, just insert

```
[window bindCommand: 2002
         withObject: window
        andSelector: @selector(performClose:)];
```

into the source code of textview before the `-makeKeyAndOrderFront:` statement.

This results in calling `[window performClose: window]` whenever the menu item *Exit* gets selected by the user.

4.4 An Application Using a Dialog and Command Bindings

To demonstrate how to use and load dialog windows from a binary resource file and command bindings, let's look at a simple application providing a (very limitated) interface to the powerful plotting program *Gnuplot*.

The backend (`gnuplot.exe`) is assumed to be installed somewhere in the program search path. This interface doesn't check, if the program could be successfully found and started.

The program itself only consists of a dialog, which is displayed when starting the program. This dialog contains three entry fields, a checkbox and a pushbutton.

The first entry field is used to specify, which function to plot, the other two to specify the horizontal plotting range. The plotting range is only used, when the checkbox is in checked state. After pressing the pushbutton Plot, the entry fields and the checkbox are computed and the function is plotted.

Figure 4.2 shows what the dialog looks like.

The main implementation file called plot.m is really simple. It just creates the necessary instances of StdApp and StdDialog. In addition to this a controller object is instantiated which reads and interprets the data from the entry fields and performs the plotting.

After creating all objects, a command binding is set up for the pushbutton Plot with the method -plot: of controller.

Then the dialog is shown and run modal and afterwards all previously allocated objects get freed again.

4.4.1 "plot.m", the Main Implementation

```
#include <pm/pm.h>

#include "gnuplot.h"
#include "controller.h"

main()
{
  StdApp     *application = [[StdApp alloc] init];
  StdDialog  *mainDialog = [[StdDialog alloc]
                              initWithId: IDD_MAIN];
  Controller *controller = [[Controller alloc] init];

  [mainDialog createObjects];

  [mainDialog bindCommand: DID_OK withObject: controller
               selector: @selector(plot:)];

  [mainDialog runModalFor: nil];

  [controller free];
  [mainDialog free];
  [application free];
}
```

[[StdDialog alloc] initWithId: IDD_MAIN] creates a dialog object and loads its binary resource template from the main binary resource file. The dialog id is IDD_MAIN.

[mainDialog bindCommand: ...] binds the command message sent by the pushbutton, which has id DID_OK to the -plot: method of the object controller.

[mainDialog runModalFor: nil] runs a modal dialog. Normally, this dialog is run modal for a certain window, but when nil is specified, this only causes the method to wait for termination of the dialog window.

The class Controller itself has to load the program gnuplot.exe and send it appropriate commands to plot the given function.

The class implements one instance variable, gnuplot to store a file handle to the gnuplot program, and three methods, -init to open the plotting program, -free to close it at the end and -plot:, which does the plotting work. The following interface declarations is stored as controller.h in \usr\samples\gnuplot.

4.4.2 "controller.h", Gnuplot PM Interface

```
#include <pm/pm.h>
#include <stdio.h>

@interface Controller : Object
{
  FILE *gnuplot;
}

- init;
- free;
- plot: sender;

@end
```

The implementation uses some of the unix-like features of the *emx C-Library*.

```
- init
{
  [super init];
  gnuplot = popen ("gnuplot.exe","w");
  return self;
}
```

`-init` first initializes its superclass `Object` and thereafter opens a *pipe* for writing to the plotting program `gnuplot.exe`. This binds `stdin` of `gnuplot.exe` to the pipe, which is represented as the file structure stored in the instance variable `gnuplot`.

```
- free
{
  pclose (gnuplot);
  return [super free];
}
```

`-free` just closes the pipe and frees its instance by calling the `-free` method of its superclass.

The following source code for the method `-plot:` is a bit more complicated. Using the `-findFromID:` method of `sender`, pointers to the entry field and checkbox objects are found out.

The function to be plot is stored in `text`, the left and right range boundaries are stored in `leftX` and `rightX`.

If the checkbox is checked, the left and right boundaries are read and converted to double numbers. Then `gnuplot` is sent the appropriate plot string used to plot a function in a given horizontal range.

If the checkbox is unchecked or one of the boundaries is not valid, `gnuplot` is sent a normal string to plot the function without specifying a plot range.

```
- plot: sender
{
  char   *string;
  char   *leftX,*rightX;
  double  left,right;

  string = [[sender findFromID: IDD_PLOTSTRING] text: NULL];

  if ([[sender findFromID: IDD_RANGECHECK] checked]) {
    leftX = [[sender findFromID: IDD_LEFTX] text: NULL];
    rightX = [[sender findFromID: IDD_RIGHTX] text: NULL];
```

```
    if ((sscanf (leftX,"%lf",&left) == 1) &&
        (sscanf (rightX,"%lf",&right) == 1) &&
        (right > left)) {
      fprintf (gnuplot,"plot [%lf:%lf] %s\n",left,right,string);
    } else
      fprintf (gnuplot,"plot %s\n",string);

    free (leftX);
    free (rightX);
  } else
    fprintf (gnuplot,"plot %s\n",string);

  fflush (gnuplot);
  free (string);

  return self;
}
```

The following section shows the complete source code of the implementation of the class `Controller`.

4.4.3 "controller.m", Gnuplot PM Interface

```
#include "Controller.h"
#include "gnuplot.h"

@implementation Controller

- init
{
  [super init];
  gnuplot = popen ("gnuplot.exe","w");
  return self;
}

- free
{
  pclose (gnuplot);
  return [super free];
}

- plot: sender
{
  char   *string;
  char   *leftX,*rightX;
  double  left,right;

  string = [[sender findFromID: IDD_PLOTSTRING] text: NULL];

  if ([[sender findFromID: IDD_RANGECHECK] checked]) {
    leftX = [[sender findFromID: IDD_LEFTX] text: NULL];
    rightX = [[sender findFromID: IDD_RIGHTX] text: NULL];

    if ((sscanf (leftX,"%lf",&left) == 1) &&
        (sscanf (rightX,"%lf",&right) == 1) &&
        (right > left)) {
```

```
      fprintf (gnuplot,"plot [%lf:%lf] %s\n",left,right,string);
    } else
      fprintf (gnuplot,"plot %s\n",string);

    free (leftX);
    free (rightX);
  } else
    fprintf (gnuplot,"plot %s\n",string);

  fflush (gnuplot);
  free (string);

  return self;
}

@end
```

4.4.4 Resource Definition

The resource definition consists of three files, the main resource definition file, which only includes the dialog template definition. The dialog template definition file defines the main dialog; and the header file to declare all constants used by the dialog definition.

```
#define INCL_PM
#define INCL_NLS

#include <os2.h>
#include "gnuplot.h"

rcinclude gnuplot.dlg
```

The file depicted above is stored as `gnuplot.rc`. It only includes the files `os2.h` and `gnuplot.h`, which are the headerfiles used for the resource definition, and afterwards includes the dialog definition file `gnuplot.dlg`. In contrast to the Objective C source code files, the file `<os2.h>` is included instead of `<objc/os2.h>`. The reason for this is that only Objective C programs need a special version of this include file because some of the data types and definitions made in `<os2.h>` are not compatible with some of the Objective C datatypes and definitions.

```
DLGTEMPLATE IDD_MAIN LOADONCALL MOVEABLE DISCARDABLE
{
  DIALOG "GNUPLOT Interface",
         IDD_MAIN, 158, 90, 210, 65,
         FS_NOBYTEALIGN | FS_DLGBORDER |
         FS_SCREENALIGN | NOT WS_VISIBLE |
         WS_CLIPSIBLINGS | WS_SAVEBITS,
         FCF_TITLEBAR | FCF_SYSMENU | FCF_NOBYTEALIGN
  {
    CONTROL "",
            IDD_PLOTSTRING, 60, 43, 127, 8, WC_ENTRYFIELD,
            ES_MARGIN | ES_AUTOSCROLL | WS_TABSTOP | WS_VISIBLE
            CTLDATA 8, 32, 0, 0
    CONTROL "Function:",
            0, 15, 43, 40, 8, WC_STATIC,
            SS_TEXT | DT_LEFT | DT_TOP | DT_MNEMONIC | WS_GROUP |
            WS_VISIBLE
    CONTROL "Range:",
            0, 15, 30, 40, 8, WC_STATIC,
```

```
            SS_TEXT | DT_LEFT | DT_TOP | DT_MNEMONIC | WS_GROUP |
            WS_VISIBLE
    CONTROL "",
            IDD_LEFTX, 60, 30, 50, 8, WC_ENTRYFIELD,
            ES_MARGIN | ES_AUTOSCROLL | WS_TABSTOP | WS_VISIBLE
            CTLDATA 8, 8, 0, 0
    CONTROL "",
            IDD_RIGHTX, 120, 30, 50, 8, WC_ENTRYFIELD,
            ES_MARGIN | ES_AUTOSCROLL | WS_TABSTOP | WS_VISIBLE
            CTLDATA 8, 8, 0, 0
    CONTROL "",
            IDD_RANGECHECK, 179, 30, 10, 10, WC_BUTTON,
            BS_AUTOCHECKBOX | WS_TABSTOP | WS_VISIBLE
    CONTROL "Plot",
            DID_OK, 145, 10, 40, 14, WC_BUTTON,
            BS_PUSHBUTTON | BS_DEFAULT | WS_TABSTOP | WS_VISIBLE
  }
}
```

`gnuplot.dlg` defines a dialog template for a dialog window having the PM identifier `IDD_MAIN`. This resource template is normally created by a dialog editor. Using resource scripts is not the preferred way to store user interface data when using the Objective C class libraries. Instead you should create user interface objects—as far as it is possible at the moment—using the Interface Editor application also found in this package. See Chapter 9 and the following chapters and the Application Development Tools manual for an introduction into this matter.

```
#define IDD_MAIN            3000
#define IDD_PLOTSTRING      3001
#define IDD_PLOT            3002
#define IDD_LEFTX           3003
#define IDD_RIGHTX          3004
#define IDD_RANGECHECK      3005
```

The include file `gnuplot.h` normally is also created by the dialog editor. It contains definitions for the constants used in the resource definition file.

The binary resource file can be created using `RC.EXE` by typing the command sequence `rc -r gnuplot.rc` at an OS/2 command line. This creates the binary resource file `gnuplot.res`, which can be linked to the application as the main resource file.

Compare the following `makefile` to the makefile template described in section 3.7 at page 18 to realize, how to fill in the blanks to create a makefile specific for your applications.

```
# Makefile for PM programs using Objective C class library

include Makefile.preamble

ifeq (.depend,$(wildcard .depend))
include .depend
endif

APPLICATION = plot.exe
OBJECTS = plot.o controller.o
RESOURCES = gnuplot.res

all: $(APPLICATION)

depend dep:
	$(CPP) -MM *.m > .depend
```

```
$(APPLICATION): $(OBJECTS) $(RESOURCES)
	$(CC) -o $(APPLICATION) $(OBJECTS) $(RESOURCES) \
		-lobjcpm -lobjc
	emxbind -ep $(APPLICATION)
	$(STRIP) $(APPLICATION)

clean:
	rm -rf $(OBJECTS) $(RESOURCES) $(APPLICATION) core *~
```

Chapter 5

Using the Database Library

In contrast to the previous chapters this chapter will not deal with writing Presentation Manager programs. Instead of this a short overview of the database library is presented. To simplify things the first application making use of the database features will have no PM interface. The later chapters will describe how to make use of both libraries—for PM and for database programming—in one application program.

5.1 Preparations

After installing the library package, the include files for the database library can be found in `\usr\include\db`. Modules which use some of the classes provided, simply have to include the file `<db/db.h>`. Just as with `<pm/pm.h>`, this single include file includes all necessary files to provide full access to all classes and methods.

After compilation the program must be linked with `objcdb.a`, the library file, and again with the utility classes library `objcutil.a`. This can be accomplished by specifying `-lobjcdb` and `-lobjcutil` when linking the program with *gcc*.

5.2 Accessing a DBase III File

As was already mentioned before, the database library provides read and write access to DBase III data files. At the moment, *Memo-fields* are not supported.

There's also no way to use indexing or sorting now. As this restricts the usability of the library to small database files, indexing is worked on at the moment.

The basic class for accessing DBase III files is `DBFile`. When allocating and initializing an object of this class, an *existing* data file is opened for reading and writing. This implies, that concurrent access to the same database files is not provided. So be careful not to write programs accessing the same database files at the same time.

A main function of a C program using database files would look like that:

```
main ()
{
  DBFile *myDBFile;

  .
  .
```

```
  .

  myDBFile = [[DBFile alloc]     // allocate and initialize
              init: "test.dbf"]; // data file "test.dbf"

  .  /*
  .   * here the database file can be used, records can be
  .   * read, modified or written back.
  .   */

  [myDBFile free]; // close data file and free resources
}
```

Access to a `DBFile` is record-oriented. Every `DBFile` object contains a buffer area large enough to store the data of exactly one record.

This *record buffer* can be filled with a record already stored in the database file and can be written back to disk. Additionally the record can be modified by the application program using the database library.

The program fragment shown above allocates and initializes a `DBFile` object and opens a database file called `test.dbf` for reading and writing.

This program—whose functionality will be extended throughout this and the next chapter—can be found in `\usr\samples\dbtest` after installing the sample files.

It defines a simple database file with two fields. The first field is called *NAME*. It can hold a string with a maximum length of 30 characters, the second is called *PHONE*, and is able to store a string with a maximum length of 20 characters.

The following records are already stored in the database file:

Nr.	**NAME**	**PHONE**	**remark**
0	Joe	23987-3	
1	Frank	6334589	
2	Sue	523593	
3	Michael	9845-43	deleted
4	Kurt	2543	

As you can see, the database file contains five records. The fourth one is marked as deleted. In the following, it will be shown how access to this file is achieved.

`DBFile` provides a method to read in a record from the associated database file. This method is called `-readRecord:`. It takes exactly one parameter, which is the position where in the file the record is stored. After reading the record, the data is copied into the record buffer where it can be accessed by the application program. Take care never to use direct record access using `-readRecord:` (or the accompanying method for modifying a record, `-writeRecord:`) when access to the records using an index file is enabled.

The access to the data is accomplished by using methods of objects of type `DBField` or one of its subclasses. `DBField` implements the methods `-stringValue` to read the data stored in the record buffer and `-setStringValue:` to write data into the record buffer.

When initializing a `DBFile` object, all needed `DBField` objects are automatically created and can be used thereafter by the application program.

Access to a specfic `DBField` object is provided through the `-field:` method of `DBFile`. If you want to print the *NAME* field of the currently loaded record, you could, for example, type

```
printf ("Name: %s\n",[[myDBFile field:0] stringValue])
```

Now, here's a simple program (it is stored as `dbtest1.m` in `\usr\samples\dbtest`) which reads all records and prints the name and the phone number of every entry:

```
#include <db/db.h>

main ()
{
  DBFile *myDBFile = [[DBFile alloc] init: "test.dbf"];
  int     i;

  printf ("NAME                           PHONE              \n");
  printf ("==================================================\n");
  for (i = 0;i < 5;i++)
  {
    [myDBFile readRecord: i];
    printf ("%-30s %s\n",[[myDBFile field:0] stringValue],
                         [[myDBFile field:1] stringValue]);
  }

  [myDBFile free];
}
```

Using `-readRecord:` all records are read and by using the `-stringValue` method of `DBField`, the two fields *NAME* and `PHONE` are printed to `stdout`.

To compile this program simply type in

```
gcc -o dbtest1.exe dbtest1.m -lobjcdb -lobjcutil -lobjc
```

As you may have noticed the name and the phone number of Michael are also printed after compiling, linking and starting the application. It seems the library does not take note of the fact that some records are marked as deleted and some are not.

Using `-readRecord:` the application program can access all stored records, even those, which have been marked as deleted. There is an instance method of `DBFile` called `-deleted`, returning a boolean value, to determine if the current record is deleted or not.

So by extending the program with the line

```
if (![myDBFile deleted])
```

before the line printing the contents, only those records, which are not marked as deleted would be printed. Look at `dbtest2.m` to see the whole source code including this simple modification.

In this program, the number of records stored in the database is hard-coded into the program. But normally, we do not know, how many records there are stored in a database file. Therefore a method called `-recordCount` was implemented for instances of the class `DBFile`. This method returns the number of records stored in the database file.

This example just reads every record stored in the database and prints only those records, which are not marked as deleted. As many programs are likely to use this kind of "linear addressing scheme", two methods are implemented which allow to read all active (not deleted) records sequentially. These two methods are called `-findFirst` and `-findNext`.

`-findFirst` tries to find the first active record in the database. It starts searching at the beginning of the database file, first checking record 0. This method returns `NO`, if there is no active record in the whole database. On success `YES` is returned and the first active record of the database file—eventually determined by index search order—is loaded into the buffer area.

`-findNext` then searches for the next appearance of an active record in the file. If no more active records are in the database, `NO` is returned. In case a record was found, its data is copied to the record buffer and the method returns `YES`.

Rewriting the application `dbtest2.m` to `dbtest3.m` utilizing these methods would result in the following simple program:

```
#include <db/db.h>

main ()
{
  DBFile *myDBFile = [[DBFile alloc] init: "test.dbf"];

  if ([myDBFile findFirst]) {
    printf ("NAME                           PHONE               \n");
    printf ("==================================================\n");

    do {
      printf ("%-30s %s\n",[[myDBFile field:0] stringValue],
                           [[myDBFile field:1] stringValue]);
    } while ([myDBFile findNext]);
  }
  [myDBFile free];
}
```

This program first checks, whether an active record exists, and only if this condition is met, the title lines are printed. Thereafter the record itself is printed and the next record is read in. This procedure of printing a record and reading a new record is continued until `-findNext` notifies the program that no more active records exist.

Using the `-delete` method, you can mark a record as deleted. The changes are written to the database file immediately. The record is only marked as deleted in the database file, it is not removed from disk automatically. This enables you to *undelete* a record if needed.

Deleting the second record, the name and phone number of Frank, would look like this:

```
  [myDBFile readRecord: 1]; // read record
  [myDBFile delete];        // mark record as deleted
```

Again note that—just alike array elements in C—the indices of the five records range from 0 to 4.

As simple as deleting a record, you can mark an already deleted record as active again. Just use `-undelete`. The following two lines will mark the fourth record, Michaels name and phone number, as active again:

```
  [myDBFile readRecord: 3]; // read record
  [myDBFile undelete];      // mark record as active
```

When using `-findFirst` and `-findNext` the index of the record in the buffer area can be queried using the instance `-currentRecord`. This can provide useful when editing more than one record in memory at once. Remember that undeleting a record is not possible when using an index file.

The order of the records as returned by `-findFirst` and `-findNext` depends on the "natural" order the records are stored in the database file. In case a (secondary) index is used, the retrieve order normally reflects the sort order of the index key. For more information see the Reference manual.

Chapter 6

Modifying Data

As shown in the last chapter, opening a database file and reading records from it is quite simple. Some pages ago I mentioned that the database file is opened for reading *and* writing. This chapter will describe how to modify data and append new records to an existing database file.

In this chapter, an application for managing the data in the previously introduced database file `test.dbf` will be created. This application shall be able to

- display all records stored in the database,
- add new records,
- modify existing records,
- delete record and
- mark deleted records as active again.

For this purpose a new class called `AddressDatabase` is defined which can handle all these functions as instance methods. The interface declaration for this class looks like this:

```
@interface AddressDatabase : Object {
  DBFile *database;
}

- init;
- free;

- (int) menu;
- printInfo;
- deleteRecord;
- undeleteRecord;
- addRecord;
- modifyRecord;
@end
```

The class is derived from `Object` and has only a single instance variable called `database`, which ist used to store a pointer to the `DBFile` instance responsible for all database operations.

The two methods `-init` and `-free` are implemented to create the `DBFile` instance and initialize it, respectively to free it.

`-menu` shall print all active records stored in the database and display a simple menu, where the user of the application can choose what he wants to do.

`-printInfo` is used to print a list of all active records in the database file.

`-deleteRecord` and `-undeleteRecord` will ask the user, which record he wants to mark as deleted or active, and then commit the task of deleting or activating the specified record.

`-addRecord` prompts the user for the necessary data (NAME, PHONE), which is to be stored in a new record and then appends this newly created record to the database file.

`-modifyRecord` on the other hand allows the user to change the data of a record already stored in the database file.

The program will look like this:

```
#include <db/db.h>

@interface AddressDatabase : Object {
 .
 .
 .
@end

@implementation AddressDatabase
 .
 .
 .
@end

main ()
{
  AddressDatabase *mydb = [[AddressDatabase alloc] init];
  int chosen;

  while ((chosen = [mydb menu]) != 5) {
    switch (chosen) {
    case 1:
      [mydb addRecord];
      break;
    case 2:
      [mydb modifyRecord];
      break;
    case 3:
      [mydb deleteRecord];
      break;
    case 4:
      [mydb undeleteRecord];
    default: ;
    }
  }

  [mydb free];
}
```

As you can see, the main function only creates and initializes the object `mydb` and then displays and queries the menu. If menu selection 5 is chosen (*end program*), `mydb` is freed again and the application terminates.

6.1 "-init" and "-free"

At the beginning, we will implement the two methods `-init`, which is the proposed *constructor method* for `AddressDatabase` and `-free`, which is the *destructor method*.

```
- init
{
  [super init];
  database = [[DBFile alloc] init: "test.dbf"];
  return self;
}

- free
{
  [database free];
  return [super free];
}
```

It should be obvious what these simple methods are doing so no further discussion on them is necessary.

6.2 Printing all Records in the Database File

`-printInfo` is used to read all active records and print their contents to `stdout`[1]. This is accomplished by the following lines:

```
- printInfo
{
  if ([database findFirst]) {
    printf ("Nr. NAME                           PHONE              \n");
    printf ("=======================================================\n");

    do {
      printf ("%3d %-30s %s\n",
              [database currentRecord],
              [[database field:0] stringValue],
              [[database field:1] stringValue]);
    } while ([database findNext]);
  }
}
```

This looks like the simple program `dbtest3` which was described in the previous chapter. In addition to printing the fields *NAME* and *PHONE*, the number of the record is printed in the first column. The number of the currently read record (the record fetched into the internal record buffer of the `DBFile` object) can be queried using `-currentRecord`.

6.3 Displaying the Menu

To provide an easy to use user-interface—which is by no means as elegant as a PM application program—a simple menu is printed, to let the user choose what actions he wants to perform on the database file.

[1] `stdout` is normally connected to the window or fullscreen session the program is started from

This user-interface is implemented in the method `-menu`, which displays all active records and a menu, and then lets the user choose which program function he would like to execute. `-menu` returns the number of the chosen program function; a list of these functions is shown in the following table:

Nr.	Program function
1	add new record
2	modify existing record
3	delete record
4	mark record as active
5	exit program

This method—just a simple sequence of C statements, without using some of the methods provided by the database library—is implemented like this:

```
- (int) menu
{
  int chosen;

  printf ("\n             Address Database\n\n");

  [self printInfo];

  printf ("\n    (1) ... add Record     (2) ... modify Record\n");
  printf ("    (3) ... delete Record  (4) ... restore Record\n");
  printf ("    (5) ... quit Program\n");
  printf ("\n        What shall I do? ");
  scanf ("%d",&chosen);
  printf ("\n");

  return chosen;
}
```

6.4 Deleting a Record

```
- deleteRecord
{
  long recNumber;

  printf ("\nWhich record shall I delete? ");
  scanf ("%d",&recNumber);

  [database readRecord: recNumber];

  if ([database deleted]) {
    printf ("\nThis record is already deleted!\n");
    return nil;
  }

  [database delete];

  return self;
}
```

6.5 Marking a Record as Active

```
- undeleteRecord
{
  int  i;
  int  j = [database recordCount];
  BOOL found = FALSE;

  for (i = 0;i < j;i++)
  {
    [database readRecord: i];
    if ([database deleted]) {
      if (!found) {
        printf ("Nr. NAME                           PHONE               \n");
        printf ("=======================================================\n");
        found = TRUE;
      }
      printf ("%3d %-30s %s\n",i,
              [[database field:0] stringValue],
              [[database field:1] stringValue]);
    }
  }

  if (!found) {
    printf ("No deleted records found!\n\n");
    return nil;
  }

  printf ("\nWhich record shall I restore? ");
  scanf ("%d",&i);

  [database readRecord: i];

  if (![database deleted]) {
    printf ("\nThis record is not deleted, no need to restore!\n");
    return nil;
  }

  [database undelete];

  return self;
}
```

6.6 Adding a New Record

```
- addRecord
{
  char nameField[31];
  char phoneField[21];

  printf ("\nName:  ");
  scanf ("%s",nameField);

  printf ("Phone: ");
  scanf ("%s",phoneField);
```

```
  [database clear];
  [[database field: 0] setStringValue: nameField];
  [[database field: 1] setStringValue: phoneField];
  [database append];

  return self;
}
```

6.7 Modifying an Existing Record

```
- modifyRecord
{
  long recNumber;
  char nameField[31];
  char phoneField[21];

  printf ("\nWhich record would you like to modify? ");
  scanf ("%d",&recNumber);

  [database readRecord: recNumber];

  if ([database deleted]) {
    printf ("\nThis record is deleted! You can't modify it!\n");
    return nil;
  }

  printf ("\nName:  ");
  scanf ("%s",nameField);

  printf ("Phone: ");
  scanf ("%s",phoneField);

  [database clear];
  [[database field: 0] setStringValue: nameField];
  [[database field: 1] setStringValue: phoneField];

  [database replace];

  return self;
}
```

This program is stored as `dbtest4.m` in `\usr\samples\dbtest`. You can compile it by typing

```
gcc -o dbtest4.exe dbtest4.m -lobjcdb -lobjcutil -lobjc
```

Note that this application does very little error-checking. It is not meant to be an end-user application, but only to demonstrate the usage of the diverse methods provided by the `DBFile` class.

As you can see, it is really simple to utilize the provided class `DBFile` to create your own database applications. Most of the code shown above is concerned with printing information on the screen and prompt for user actions. In the next chapter you will see, how to combine the features of the OS/2 PM class library and the database library in a simple Presentation Manager application.

Chapter 7

A Sample PM Application Using the Database Library

This chapter will demonstrate the concepts of integrating the classes of the PM library with the one provided by the database library to create an application used to store phone numbers and e-mail addresses.

7.1 Purpose of the Application

This sample program will be used to demonstrate an implementation of a presentation manager application DBase III files. In addition to this the program provides a framework for an address database which can eventually provide useful to some developers. The main focus was *not* laid upon creating a full-featured and easy-to-use application program but to show the concepts of combining the two different topics dealt with up to now in this document.

The application will store records in a database having a structure as specified in Table 7.1.

The main window of the application will contain a single listbox, only displaying the first field of every record (NAME).

Adding new records, editing records, deleting records and displaying all information stored for a specific record will be realized with dialog windows.

Figure 7.1 on the following page shows the main window of the address database displaying some records.

Field Nr.	**Name**	**Length**	**Type**	**Description**
1	NAME	40	Character	Name of the person.
2	ADDRESS	40	Character	Address of the person.
3	PHONE	40	Character	Phone number of the person.
4	FAX	40	Character	Fax number of the person.
5	EMAIL	40	Character	E-Mail address of the person.

Table 7.1: Data format used for storing addresses

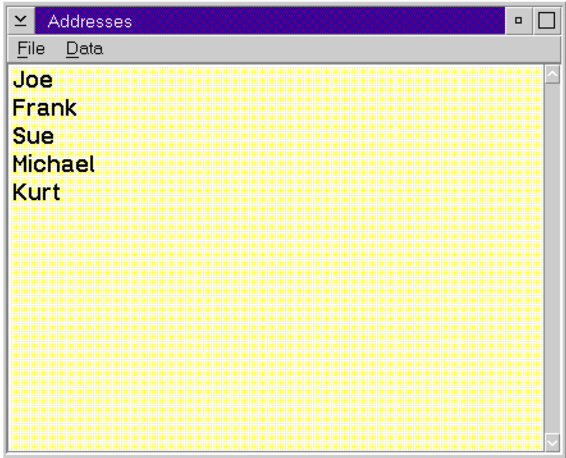


Figure 7.1: Main window of address database

7.2 Application Menu

To process user actions the application will provide a menu bar with two menus in it.

The menu *File* will only provide one menu item to let the user exit the application. It is named *Exit*.

The second menu used by this program is called *Data*. Here all actions concerning data manipulation, such as adding new records, deleting records etc. can be found.

The menu items found here are *New...* to add a new record to the end of the database, *Edit...* to change the data stored for the selected record in the listbox, *Delete* to delete a record from the database and *Info...* to display the information associated with the selected record in the listbox control.

This program will only display the records which are not marked as deleted in the database file. Deleting a record only marks it as deleted, but by means of this program, the record cannot be marked as active again.

7.3 Database File

The program automatically opens a database file called "`address.dbf`" in the current directory at startup and writes the changes to this file when exiting the application.

An empty database template file is provided in `\usr\samples\address`; it is stored with the name "`empty.dbf`".

7.4 Classes Used in the Application

In addition to the standard PM and database classes only one new class is required. This class will be called `Controller`. It will implement all necessary methods required by the program to carry out its job, for example *record insertion* etc.

The methods defined here can be identified as belonging to three different classes. The *initializing* methods are used to open the database file and read in all records at program start. The *database*

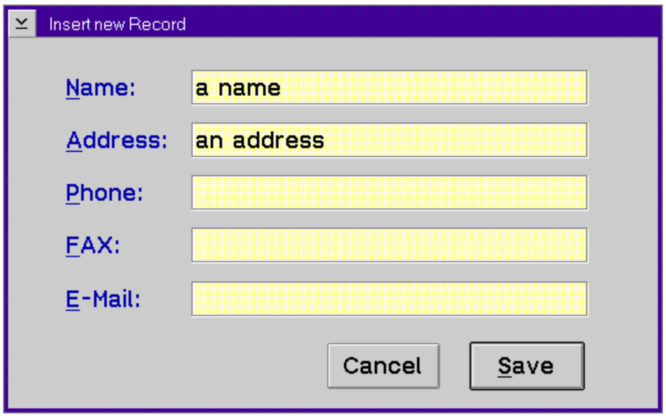


Figure 7.2: Dialog window to add a new record

access methods will be called directly from the appropriate menu items and all display dialog windows to change or view the data stored. At last, the *delegate* methods are implemented, which are used to react to some user actions, such as resizing the window and exiting the program.

In addition to these methods, `Controller` defines many instance variables which are used by some of the methods to access the database or the dialog windows on the screen.

7.4.1 Instance Variables

The following instance variables are defined and used by this class.

```
  @interface Controller : Object
  {
    StdDialog *insertRecord;
    StdDialog *replaceRecord;
    StdDialog *infoRecord;

    id insertName;
    id insertAddress;
    id insertPhone;
    id insertFax;
    id insertEMail;
    id replaceName;
    id replaceAddress;
    id replacePhone;
    id replaceFax;
    id replaceEMail;
    id infoName;
    id infoAddress;
    id infoPhone;
    id infoFax;
    id infoEMail;

    DBFile *database;
    DBList *recordList;
}
```

`-insertRecord`, `-replaceRecord` and `-infoRecord` are used to store pointers to the dialogs for adding a new record (see Figure 7.2), editing and saving the data for an existing record (the dialog

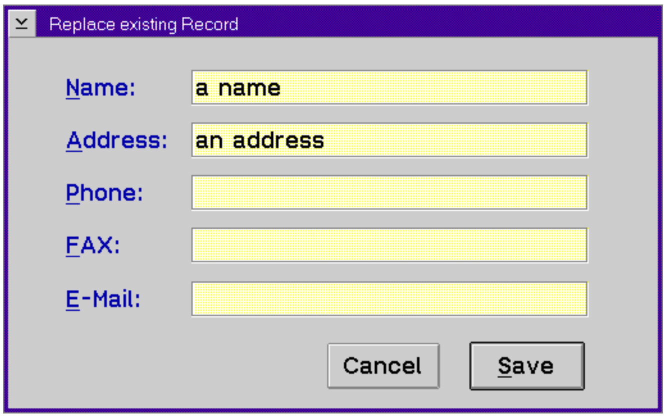


Figure 7.3: Dialog window for editing already existing records

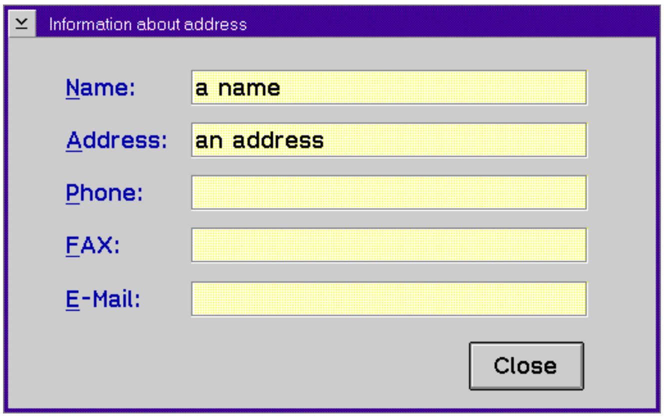


Figure 7.4: Dialog window used to display all information stored in a record

is shown in Figure 7.3) and for displaying the information associated with a record (see Figure 7.4).

The instance variables defined as `ids` are initialized to provide easy and fast access to the dialog items themselves. They are initialized at dialog creation.

`database` is a pointer to a `DBFile` object which is used to retrieve and save records from and to the database file itself.

`recordList` is a list object to enable the application to hold more than one record at a time in memory. This provides faster and easier access to the single records. When the program is started, all record information is retrieved into this object and when the application exits all modified or new records are written to the database again.

7.4.2 Initializing Methods

The initializing methods are used to initialize the object itself and all objects used by it. Additionally, the destructor method `-free` is also described here.

- `-init` is used to create all dialog windows in memory. Here all instance variables of this object are initialized. Additionally, the database file is opened and the record list object is initialized.

- `-free` is the destructor method of this object. Here all objects are destroyed again.
- `-readList:` retrieves all records stored in the database file into the record list and afterwards displays the first data field of every record in the listbox. See Figure 7.1 on page 42 for an example main window as created and filled by this method.

The `Controller` object used in this application is created and initialized by the `main ()` function of the program. This function also creates the main window, also including the listbox, and calls `-readList:`.

The method `-init` of the `Controller` class looks like this:

```
- init
{
  insertRecord = [[StdDialog alloc] initWithId: IDD_INSREP];
  replaceRecord = [[StdDialog alloc] initWithId: IDD_INSREP];
  infoRecord = [[StdDialog alloc] initWithId: IDD_INFO];

  [insertRecord setText: "Insert new Record"];
  [replaceRecord setText: "Replace existing Record"];

  [insertRecord createObjects];
  [replaceRecord createObjects];
  [infoRecord createObjects];

  insertName = [insertRecord findFromID: IDD_NAMEENTRY];
  insertAddress = [insertRecord findFromID: IDD_ADDRESSENTRY];
  insertPhone = [insertRecord findFromID: IDD_PHONEENTRY];
  insertFax = [insertRecord findFromID: IDD_FAXENTRY];
  insertEMail = [insertRecord findFromID: IDD_EMAILENTRY];
  replaceName = [replaceRecord findFromID: IDD_NAMEENTRY];
  replaceAddress = [replaceRecord findFromID: IDD_ADDRESSENTRY];
  replacePhone = [replaceRecord findFromID: IDD_PHONEENTRY];
  replaceFax = [replaceRecord findFromID: IDD_FAXENTRY];
  replaceEMail = [replaceRecord findFromID: IDD_EMAILENTRY];
  infoName = [infoRecord findFromID: IDD_NAMEENTRY];
  infoAddress = [infoRecord findFromID: IDD_ADDRESSENTRY];
  infoPhone = [infoRecord findFromID: IDD_PHONEENTRY];
  infoFax = [infoRecord findFromID: IDD_FAXENTRY];
  infoEMail = [infoRecord findFromID: IDD_EMAILENTRY];

  database = [[DBFile alloc] init: "address.dbf"];
  recordList = [[DBList alloc] initForDatabase: database];

  return self;
}
```

In the beginning the three necessary dialog windows are created. Note, that the dialog windows for adding a new record and editing an existing record are created from the same dialog template. They only differ in the dialog title. Therefore the next two lines set the title strings for these two dialog windows to either “`Insert new Record`” or “`Replace existing record`”.

Afterwards the user interface objects in the dialogs are created and the instance variables used to simplify access to them are initialized.

In the end the database object is created and initialized and the database file “`address.dbf`” is opened. The record list used to hold all records in memory is created and the method returns to the caller.

`-free` only frees all objects associated or created with/by this object. The simple source code for this method looks like this:

```
- free
{
  [[recordList freeObjects] free];

  [database free];
  [insertRecord free];
  [replaceRecord free];
  [infoRecord free];

  return [super free];
}
```

The last initializer methods—called `-readList:`—is used to fill the record list with all data in the database. This can simply be accomplished by calling `[recordList fetchAllRecords: nil]`. The only parameter `-fetchAllRecords:` takes is totally ignored by the `DBList`-object. It is only used for the method to become an *action* method which can provide useful when using the visual development tools.

Then the first data field of every record is extracted from the list and displayed in the listbox in the main window.

```
- readList: sender;
{
  ListBox *nameListBox = [sender findFromID: IDD_PUSHBUTTON1];
  int i;

  [recordList fetchAllRecords];

  for (i = 0;i < [recordList count];i++) {
    [nameListBox insertItem: LIT_END
                     text: [[[recordList objectAt: i] field: 0] stringValue]];
  }

  return self;
}
```

You can see that a simple `for`-loop is used to visit all records in the list. Then the string stored in the first data field is appended to the listbox as a new listbox item. The class `DBRecordList` is a subclass of `SimpleList` which can be found in the utility library (`objcutil.a`). For detailed information on the methods for access to single objects stored in the list, see the reference manual. Just as `DBFile` every `DBRecord` object contains a list of fields which can be used for access to the field data.

7.4.3 Database Access Methods

To modify the information stored in the database file and displayed in the listbox four methods—called `-insert:`, `-replace:`, `-info:` and `-delete:`—have been implemented. These methods are called directly as response to user actions. Therefore these methods have to be declared as *actions*, resulting in every method having exactly one parameter called `sender`.

The simplest of these methods is `-delete:`. This method shall display a message box querying the user if he really wants to delete the selected record. If no record was selected `-delete:` has to return `nil` without further processing.

```
- delete: sender
{
  ListBox *nameListBox = [sender findFromID: IDD_PUSHBUTTON1];
```

```
  SHORT selected = [nameListBox selected];

  if (selected < 0)
    return nil;

  .
  .
  .

  return self;
}
```

This piece of source code first queries a pointer to the listbox object using `-findFromID:` of the sending object, which always is the main window. This pointer is stored in `nameListBox`. The variable `selected` is used to store the index of the selected item in the listbox. If no item is selected, `selected` is less than `0`.

Additionally, the following variables are declared in this method:

```
  char  numberBuffer[6];
  long  numberOfRecord;
  char *nameBuffer;
```

The *working part* of the method first shows a message box querying the user if he really wants to delete the record, and then marks the record as deleted and removes it from the listbox. The record must also be deleted from the record list.

```
  if (WinMessageBox (HWND_DESKTOP,[sender window],
                     "Do you really want to delete the selected Item?",
                     "Addresses",
                     0,MB_YESNO | MB_QUERY) == MBID_YES) {
    numberOfRecord = atoi(numberBuffer);

    [database readRecord: [[recordList objectAt: selected] recNo]];

    [nameListBox deleteItem: selected];
    [database delete];
    [recordList deleteRecordAt: selected];
  }
```

For a description of `WinMessageBox()` see the OS/2 PM API documentation.

The next method, `-info:`, is used to display a dialog window which was created throughout the initialization sequence of the `Controller` object (`infoRecord`) and fill the entry fields with the data associated with the selected record.

The lines of source code for the *interesting* part of the implementation are shown below:

```
- info: sender
{
  .
  .
  .

  record = [recordList objectAt: selected];

  [infoName setText: [[record field: 0] string]];
  [infoAddress setText: [[record field: 1] string]];
  [infoPhone setText: [[record field: 2] string]];
  [infoFax setText: [[record field: 3] string]];
```

```
  [infoEMail setText: [[record field: 4] string]];

  [infoRecord runModalFor: sender];

  return self;
}
```

A pointer to the record object representing the selected listbox item is stored in `record`.Afterwards the text in the data fields is displayed in the entry field objects, and the dialog window is run modal for the main window.

The instance method `-replace:` is used to edit the data of an already existing record. In the beginning the corresponding entry fields must be filled with the strings stored in the data fields of the selected record, as shown previously.

After processing the dialog the data must be copied from the entry fields into the record list and the database file is updated to reflect the changes.

```
- replace: sender
{
  .
  .
  .
  [replaceRecord runModalFor: sender];

  if ([replaceRecord result] == DID_OK) {
    nameBuffer = [replaceName text: NULL];
    addressBuffer = [replaceAddress text: NULL];
    phoneBuffer = [replacePhone text: NULL];
    faxBuffer = [replaceFax text: NULL];
    emailBuffer = [replaceEMail text: NULL];

    [[record field: 0] setStringValue: nameBuffer];
    [[record field: 1] setStringValue: addressBuffer];
    [[record field: 2] setStringValue: phoneBuffer];
    [[record field: 3] setStringValue: faxBuffer];
    [[record field: 4] setStringValue: emailBuffer];

    [record replace];

    [nameListBox deleteItem: selected];
    [nameListBox insertItem: selected text: nameBuffer];

    free (nameBuffer);
    free (addressBuffer);
    free (phoneBuffer);
    free (faxBuffer);
    free (emailBuffer);
  }

  return self;
}
```

As the memory used for temporarily storing the strings in the entry field controls is allocated automatically by `-text:` this buffer area must be freed again later using `free ()`. The implementation is really simple. The text strings are retrieved from the entry fields, written into the `DBRecord` object and then the modifications are written back to the database file.

The implementation of `-insert:` is just the same as `replace:` with the difference, that the entry fields must be initialized with empty strings because a new record is allocated at the end. In contrast to `-replace:` this method will store the field data *directly* in the database buffer area of the `DBFile` object and will not use the buffer area of an already existing `DBRecord` object in the record list. After appending the record to the database file—using the `DBFile` method `-append`—a new `DBRecord` object is created and inserted into the record list. It has to be taken care of also updating the display area of the listbox control.

These functionality can be achieved by implementing `insert:` as shown below. For better understanding the complete code for this method is shown:

```
- insert: sender
{
  ListBox *nameListBox = [sender findFromID: IDD_PUSHBUTTON1];
  char    *nameBuffer;
  char    *addressBuffer;
  char    *phoneBuffer;
  char    *faxBuffer;
  char    *emailBuffer;

  [database clear];

  [insertName setText: ""];
  [insertAddress setText: ""];
  [insertPhone setText: ""];
  [insertFax setText: ""];
  [insertEMail setText: ""];

  [insertRecord runModalFor: sender];

  if ([insertRecord result] == DID_OK) {
    nameBuffer = [insertName text: NULL];
    addressBuffer = [insertAddress text: NULL];
    phoneBuffer = [insertPhone text: NULL];
    faxBuffer = [insertFax text: NULL];
    emailBuffer = [insertEMail text: NULL];

    [[database field: 0] setStringValue: nameBuffer];
    [[database field: 1] setStringValue: addressBuffer];
    [[database field: 2] setStringValue: phoneBuffer];
    [[database field: 3] setStringValue: faxBuffer];
    [[database field: 4] setStringValue: emailBuffer];

    [database append];
    [recordList addObject: [[DBRecord alloc] initForEntity: database]];

    [nameListBox insertItem: LIT_END text: nameBuffer];

    free (nameBuffer);
    free (addressBuffer);
    free (phoneBuffer);
    free (faxBuffer);
    free (emailBuffer);
  }

  return self;
}
```

This method appends a new record to the data file and also creates a new data object at the end of the record list. At last a new item is inserted and displayed in the listbox control.

All methods described above are fully responsible for displaying all necessary information (dialog windows, message boxes) and leave the database and the record list in memory in a consistent state. Do not forget to update all storage structures (record list, data file, listbox) every time you modify, add or delete one of the records.

The next two methods are used as *delegate* methods for the main window.

7.4.4 Delegate Methods

As shown in Section 3.6 on page 15, the following simple method is called every time the size of the main window is changed by the user. Its only purpose is to adjust the size of the listbox to the size of the main window.

```
- windowDidResize: sender
{
  ListBox *nameListBox = [sender findFromID: IDD_PUSHBUTTON1];

  [nameListBox setSize: 0:0:[sender width]:[sender height]];
  return self;
}
```

Instead of implementing `-windowDidResize:` the window object used in this application could be used directly to react to a user action of resizing the window. If only a single window object is displayed in the *client area* of an instance of `StdWindow` or `MainWindow` the window object is capable to take care of resizing its *client window* itself. To set the client window of an object use `-setClientWindow:`. This will be the preferred way of handling this special situation when using the Interface Editorprogram. An example of this can be found in Chapter 9 on page 61.

For a more complete description of the methods used here and the other ones provided by the database library, see the reference manual.

7.5 The main() Function of the Application

The `main ()` function of the application can be found in the file `addresses.m`. It only performs the standard actions, as creating and initializing the objects, and after creating all necessary objects but before starting execution of the application, the record list is read in by calling `[controller readList: mainWindow]`. The complete source code is shown below.

Here you can see that all menu items which are used are bound directly to the previously described methods of the class `Controller`. Only *Exit* will directly call `-performClose:` of the main window.

```
main()
{
  StdApp *addresses = [[StdApp alloc] init];
  MainWindow *mainWindow = [[MainWindow alloc] initWithId: IDD_MAIN
                              andFlags: (FCF_MENU | FCF_ACCELTABLE |
                                         FCF_SIZEBORDER)];
  Controller *controller = [[Controller alloc] init];

  [mainWindow setTitle: "Addresses"];

  [mainWindow setDelegate: controller];
  [mainWindow bindCommand: IDM_EXIT withObject: mainWindow
               selector: @selector(performClose:)];
  [mainWindow bindCommand: IDM_NEWAD withObject: controller
```

```
                selector: @selector(insert:)];
  [mainWindow bindCommand: IDM_EDITAD withObject: controller
                selector: @selector(replace:)];
  [mainWindow bindCommand: IDM_INFOAD withObject: controller
                selector: @selector(info:)];
  [mainWindow bindCommand: IDM_DELETEAD withObject: controller
                selector: @selector(delete:)];

  [mainWindow createObjects];

  [mainWindow insertChild: [[ListBox alloc] initWithId: IDD_PUSHBUTTON1
                andFlags: WS_VISIBLE | WS_TABSTOP
                      in: mainWindow]];

  [controller readList: mainWindow];

  [mainWindow makeKeyAndOrderFront: nil];

  [addresses run];

  [mainWindow free];
  [addresses free];
}
```

Chapter 8

Using the Container Class

One of the most interesting window classes provided by the OS/2 Presentation Manager is the container class. Newly introduced with OS/2 2.0, it provides a storage for general objects of any type. These objects can be displayed in several different styles: as icons, as text, as a combination of both, or—very similar to multi-column listboxes—in the so-called detail's view, also supporting direct editing of the values shown.

This chapter will show how to use the Objective C class `Container`, which is provided by `objcpm.a`. Section 8.1 will show how to create a container from scratch and display the contents as icons. The following sections will present a simple example of using the container as a multi-column listbox including direct editing.

8.1 Simple Container—Icon Display

Containers can be used to store *any* data you want. The only restriction set up is that all data which is to be stored in a container object must be encapsulated in an Objective C object.

The information presented in this section assumes that you already have a simple PM application including a main window. The container which will be used here will fill the whole client area of the main window.

8.1.1 Creating the Application

Here you will once more be shown how the typical source code for creating a PM application looks like—including creation and display of a main window.

```
#include <pm/pm.h>

main ()
{
  StdApp     *application = [[StdApp alloc] init];
  MainWindow *mainwindow = [[MainWindow alloc]
                              initWithId: 1000
                                andFlags: FCF_SIZEBORDER];

  [mainwindow createObjects];

  [mainwindow makeKeyAndOrderFront: nil];
```

```
  // => here goes the container stuff

  [application run];

  [mainwindow free];
  [application free];
}
```

8.1.2 Step One: Creating the Container

The first thing we will have to do is to declare a new variable of type `(Container *)`, which we will use to access the container control.

So, add `Container *container` to the declaration section of the main function.

Creating the object itself is just as simple as creating any other window object. Using `-initWithId: andFlags: in:` a new object, previously allocated with `+alloc`, will be initialized.

```
  container = [[Container alloc] initWithId: 1001
                                   andFlags: (CCS_MINIRECORDCORE |
                                              WS_VISIBLE)
                                         in: mainwindow];
  [mainwindow insertChild: container];
  [container setSize: 0:0:[mainwindow width]:[mainwindow height]];
```

The container object is created using the flags `CCS_MINIRECORDCORE` and `WS_VISIBLE`. The second one only tells the container to be visible. `CCS_MINIRECORDCORE` is used to specify which kind of data will be stored in the container. At the moment only this style is supported. Do not use the flag `CCS_RECORDCORE`.

Then the container is defined to be a child window of `mainwindow`, and the size of the newly created control is set accordingly to fill the complete client region of its parent window.

8.1.3 Step Two: Inserting Data

After creation it is possible to insert data into the container. As was stated before all kinds of Objective C objects can be stored in a container object. For simplicity, only instances of `Object` are used here.

Insertion of single objects itself can be performed using three distinct methods, `-insertObject:`, `-insertObject: withTitle:` and `-insertObject: withTitle: andIcon:`. If you are going to display the data stored in the container as icons you will normally use the third method shown above because it lets you decide on which icon will be used for display. To insert many objects at once you can use the instance methods `-insertObjectList:`, `-insertObjectList: withTitles:` and `-insertObjectList: withTitles: andIcon:`. These methods are prepared to process many single objects stored in instances of `SimpleList`. See the reference manual for more information. There you can also find information concerning a hierarchical structure of the objects in a container object as is used in the *tree view*.

- `-insertObject:` ··· inserts a new object into the container. In Icon View, no Icon text will be displayed. The Icon used for this object will be the clock-mouse pointer.

- `-insertObject: withTitle:` ··· analogous to `insertObject:` an object is inserted into the container, but here a title text can be specified.

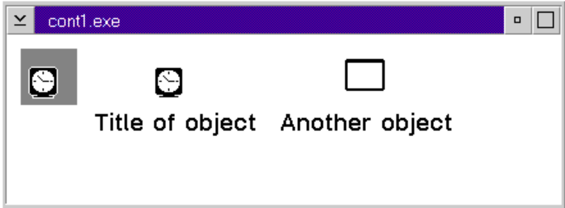


Figure 8.1: First container sample "cont1.exe"

- `-insertObject: withTitle: andIcon:` ··· here all data, which will be displayed in Icon View, can be specified. The Icon parameter must be a valid Icon resource. To query the resource handle of a valid Icon resource e.g. use the functions `WinQuerySysPointer()` or `WinLoadPointer()`.
- The methods `-insertObjectList:`, `-insertObjectList: withTitles:` and `-insertObjectList: withTitles: andIcon:` are used to insert a list of objects into a container object at once. The first parameter—which is the only one for `-insertObjectList:`—is a `SimpleList` instance containing the objects to be inserted. The second parameter is another list object containing objects storing the object titles. Every object must implement a `-stringValue` method. E.g. use istances of `String`. The third and last parameter is the resource handle of a *single* valid icon resource for all objects.

We will use the first three methods described above to insert three different objects into the container. The sample code for the record-insertion is:

```
[container insertObject: [[Object alloc] init]]; // first object

[container insertObject: [[Object alloc] init]
              withTitle: "Title of object"];     // second object

[container insertObject: [[Object alloc] init]
              withTitle: "Another object"
                andIcon: WinQuerySysPointer (HWND_DESKTOP,
                                             SPTR_APPICON,
                                             FALSE)]; // third object
[container arrange];
```

The icon resource used for the third object is queried using the API function `WinQuerySysPointer()`. Here the resource handle for the current application Icon is queried.

At the end an `-arrange` message is sent to the container to re-arrange the icons now displayed. This is necessary in this example program, otherwise all icons would cover the same space in the container in the lower left corner.

8.1.4 Complete Source Code

After inserting this code into the application program, the source code should look like this.

```
#include <pm/pm.h>

main ()
{
  StdApp     *application = [[StdApp alloc] init];
  MainWindow *mainwindow = [[MainWindow alloc]
```

```
                              initWithId: 1000
                                andFlags: FCF_SIZEBORDER];
  Container  *container;

  [mainwindow createObjects];

  [mainwindow makeKeyAndOrderFront: nil];

  container = [[Container alloc] initWithId: 1001
                                   andFlags: (CCS_MINIRECORDCORE |
                                              WS_VISIBLE)
                                         in: mainwindow];
  [mainwindow insertChild: container];
  [container setSize: 0:0:[mainwindow width]:[mainwindow height]];

  [container insertObject: [[Object alloc] init]]; // first object

  [container insertObject: [[Object alloc] init]
               withTitle: "Title of object"];     // second object

  [container insertObject: [[Object alloc] init]
               withTitle: "Another object"
                 andIcon: WinQuerySysPointer (HWND_DESKTOP,
                                              SPTR_APPICON,
                                              FALSE)]; // third object
  [container arrange];

  [application run];

  [mainwindow free];
  [application free];
}
```

Just compile and link this program:

```
gcc -o cont1.exe cont1.m -lobjcpm -lobjc
emxbind -ep cont1.exe
```

Figure 8.1 on the page before shows the output produced by this application.

8.2 Using the Details View

The details view of a container object looks like a multi-column Listbox. In addition to the features of a simple Listbox control the container class makes it possible to edit the data displayed by simply selecting it—using the system-defined direct-manipulation action for editing an objects title text, which normally is depressing the *Alt*-key and single-clicking with mouse button 1 on the text—and entering the new data.

Container objects in details view are widely used across the user interface provided by the Workplace Shell. Every folder object can be opened in details view to provide access to all data associated with the data or program objects stored inside.

As already mentioned before a container object in details view looks like a multi-column Listbox. So—in contrast to all other views of container objects—the data is always sorted in some way. In the details view, some kind of ordering—in a mathematical sense—is used to determine the position of the records in the display area. Every record (item) stored in a container is displayed in a single line of the container. Every data field of the items is displayed in one column.

8.2.1 Creating Columns

To utilitze the details view we first have to specify the number of columns to be displayed in the container object. After allocation and initialization using `+alloc` and `-initWithId: andFlags: in:` the program must create all columns needed for display. This is done using `-addColumn:`.

`-addColumn:` takes one parameter of type `(char *)`, which is the column title. This column title normally is displayed at the head of the column in the topmost line of the display area of the container object.

Adding columns at the moment is done one column at a time. So creating more than one column results in calling `-addColumn:` for each of them. Take care that the created column is always appended to the list of columns already existing. There is no way of deleting columns or rearranging them.

After creating the columns the container window should be displayed in details view, which can be accomplished using `-detailView:`. Below is a simple piece of source code adding four columns to a container object and switching the display to details view.

```
.
.
[container addColumn: "first Column"];
[container addColumn: "second Column"];
[container addColumn: "third Column"];
[container addColumn: "fourth Column"];

[container detailView: self];
.
.
```

You should add all used columns in the beginning. If you do not do so you must manually change the data stored for *all* records already stored in the container object. As you normally will know which columns will be needed for a container object at creation time of the object, this complicated procedure is not described here.

8.2.2 Which Data is Displayed?

Normally OS/2 container windows store all kinds of data structures as items. When using this library container windows can store all kinds of *Objective C objects*. But how does the container window know what data shall be displayed in every column?

Plain OS/2 API programming makes this quite difficult. You have to define a data structure, put pointers to some memory areas in it, where strings to be displayed are really stored, and every column must be initialized with an index pointer to the column data.

The container class only expects the objects stored as items to implement some methods to provide access to the data.

Every object must implement the usual initializer and destructor methods `-init` and `-free`. As `-init` is never called directly, you can also use some other method. `-free` is used whenever an item is deleted, and it must take care of freeing all memory allocated by the object.

To provide access to the single data fields, the methods `-setFieldData: withString:` and `-fieldData:` must be present. If only read-only access, i.e., data is only to be displayed, is required, only the `-fieldData:` method hast to be implemented.

`-setFieldData: (ULONG) field withString: (char *) aString` is used to set a string to field number `field`. Field numbering starts at `0`. Do not expect `aString` to be a pointer to a `(const char)` area. This memory area can be reused again, so your method must copy the data to a memory area in your object.

`-(char *) fieldData: (ULONG) field` is used to retrieve a pointer to the string stored for field `field`. As the data pointed to is never modified by the container object, this method can return a pointer to the memory area used internally for storing the data.

A simple class designed to store four fields could look like this:

```
@interface ContainerObject : Object
{
  char fields[4][30];
}

- init;

- setFieldData: (ULONG) field withString: (char *) aString;
- (char *) fieldData: (ULONG) field;

@end
```

As no data will be allocated dynamically, the class does not have to implement a `-free` method.

The implementation is just as simple as

```
@implementation ContainerObject
- init
{
  [super init];

  fields[0][0] = 0x0;
  fields[1][0] = 0x0;
  fields[2][0] = 0x0;
  fields[3][0] = 0x0;

  return self;
}

- setFieldData: (ULONG) field withString: (char *) aString
{
  if (field > 3)  // field number not valid ?
    return nil;

  strncpy (fields[field],aString,29); // copy at most 29 chars
  fields[field][29] = 0x0;            // last character is NULL

  return self;
}

- (char *) fieldData: (ULONG) field
{
  if (field > 3)  // field number not valid ?
    return NULL;
  else
    return fields[field];
}

@end
```

Note that the method `-init` is only implemented to initialize the four strings, so the data displayed is always valid.

`-setFieldData: withString:` just copies at most 29 characters into the buffer area for the string. As `strncpy()` does not append a `NULL` character if the source string is longer than the specified maximum size, the 30^{th} character is set to `0x0`.

`-fieldData:` just returns a pointer to the internal buffer area of the field data referenced by `field`.

`-free` must be implemented if any dynamic allocation is used by the object. Otherwise the default method inherited from the superclass will provide enough functionality.

8.2.3 Direct Editing

Direct editing support is not implemented at this time. If you want to use this, you have to catch the window message `CN_REALLOCPSZ` and return the appropriate value.

When using static storage, it should be enough to just catch the message and return `TRUE`. Take care, that also the title string of the column can be edited, so a distinction must be made.

A sample implementation of the `-handleMessage: withParams: and::` method of a delegate object of the window containing the container object could look like this:

```
- (MRESULT) handleMessage: (ULONG) msg withParams: (MPARAM) mp1
                     and: (MPARAM) mp2 : sender
{
  // catch WM_CONTROL message of type CN_REALLOCPSZ
  if ((msg == WM_CONTROL) && (SHORT2FROMMP (mp1) == CN_REALLOCPSZ)) {
    CNREDITDATA *cnrEditData = PVOIDFROMMP (mp2);

    // check what text was edited; either column data or title data
    switch (cnrEditData->id) {
    case CID_LEFTDVWND:         // column data was edited
      return (MRESULT) TRUE;    // just return TRUE, memory is static!

    case CID_LEFTCOLTITLEWND: { // column title was edited
      char **buffer = (char **) cnrEditData->ppszText;            // old text
      char  *buffer2 = (char *) malloc (cnrEditData->cbText + 1); // new text

      free (*buffer);     // free memory occupied by the column title
      *buffer = buffer2;  // set memory area for new column title. The
                          // data is copied into this area automatically
      return (MRESULT) TRUE;    // memory for column title was allocated

    default:                    // nothing for us!
      return (MRESULT) FALSE;
    }
  }
```

This looks quite complicated. But simply this means that the container window sends a message asking the program if the new text (length is given in `cnrEditData->cbText`) should be copied or not. If `TRUE` is returned, the text is copied to the string pointed to by `*(cnrEditData->ppszText)`. When this message is sent, `*(cnrEditData->ppszText)` stores a pointer to the string currently stored.

If you do not want the column title or the data to be copied, just return `FALSE`.

Formatting the columns or setting read-only flags can be done separately for the column data and the column title. Just use `-setColumnTitleAttributes:` to set the attributes for the title data and `-setColumnDataAttributes:` to set the data attributes. See the description of the `Container` class in the reference manual for more information.

Chapter 9

Creating an Application using the Visual Development Tools

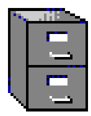
Up to now this document was mainly concerned with some kind of "raw" programming using the library package provided by this product. If you carefully read through these sections you should have got some knowledge of the structure of an OS/2 PM program written in Objective C and already got a limited overview of some of the classes which can be directly used by end-user applications.

Throughout the rest of this document the use of the *Visual Development Tools* will be discussed. These tools were created to let the users benefit from the dynamic structure of the Objective C language on the one hand, and, on the other hand, to free application developers from the time-consuming tasks of e.g. writing makefiles and doing most of the *standard* initialization work.

In the beginning an overview of the development tools themselves will be presented. The use of each of them will be discussed shortly and some of the limitations—most of them will be removed in later versions of the package—are pointed out.

9.1 Visual Development Tools

The development tools shipped together with the library package at the moment consist of three programs. The *Database Editor*, the *Interface Editor* and the *Project Editor*. Additionally, a program called *Console* is provided which is quite useful when debugging your own applications.



9.1.1 The Database Editor Application

The Database Editor Application is used to create *empty* database template files. You must use this program to set up a database structure before accessing a DBase file. Instead of this program e.g. DBase itself could be used to set up the database structure.

At the moment, you cannot modify existing database files. This limitation will be dealt with in one of the next versions, then providing full access to every DBase file.

9.1.2 The Project Editor Application

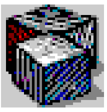
This program is the *core* of the development environment. It provides program functions for interactively creating classes, and it is designed to handle all necessary tasks to maintain the files in a project.

You can tell the program what files your application is built from, and it can then generate a makefile suitable for compilation. Just alike folders on the Workplace Shell, the files are organized in some categories and can be added, removed or edited.

When creating classes some skeleton implementation and header files are generated for you. The only thing you have to do is to fill in these “class templates”. Additionally, a main function suited for most applications is created automatically.

The program also provides some kind of *control center* where you can edit source code files from or compile and test your application.

At the moment, interaction with the other development tools (e.g. the Interface Editor application) is not fully implemented. In future versions it will be possible to start editing for *all* files of a project from this central place. The visual development tools will be more closely integrated in future versions of this development system.



9.1.3 The Interface Editor Application

While the Project Editor application is used to edit *project data*, this program is used to interactively create a program’s user interface. In contrast to normal dialog editor programs you can also create instances of some classes of your application and access other objects from the library package.

Most window classes of the libraries are implemented up to now. In the following versions, easy access to the database classes will be provided, too.

In addition to creating user interface elements—and even Objective C objects—interactively, this program also allows you to set up connections between these objects. Most of the *commands* can be bound to *actions* interactively and the *outlet* connections of the objects can be initialized by using the drag and drop features of this application.

The data is saved in an Objective C typed stream, just storing the objects you create interactively on disk, and can be loaded easily by your application program.

Using this program can save you lots of time; you just create the user interface and program objects visually and when your application starts, everything gets initialized automatically.

This and the following chapters will provide an introduction into this kind of *magic*; just try it, and you will see how simple development can be.

9.2 “Textview”—The Easy Way

For demonstration issues an application already discussed in this manual will be developed using the visual development tools. The purpose of one of the applications dealt with previously was to simply read a text file and display its contents in a window. Throughout the rest of this chapter the “Textview” application will be developed newly.

Figure 9.1: Project Template

First, drag a *project template* (see Figure 9.1) from the *Development Folder* to a folder somewhere located on your drives[1]. Using the OS/2 direct manipulation facilities, you should rename the project folder from its original name "`Objc_project`" to some name of your choice, e.g. "`Textview.prj`". Figure 9.2 shows an icon view of the "`samples`" folder, where this project can already be found.

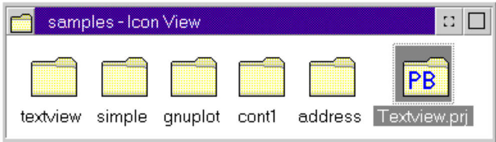


Figure 9.2: "`samples`" folder containing the new Project

Opening the new project folder will reveal its contents as depicted in Figure 9.3. There you can see the files "`pb.prj`", "`pb.cls`", "`main.oib`", "`Makefile`", and "`Main.m`".

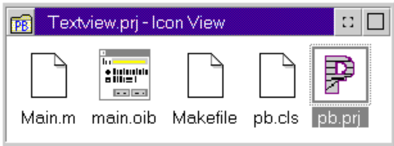


Figure 9.3: Contents of the project folder when created newly from the project template

The project data itself is stored in the two files "`pb.prj`" and "`pb.cls`". Normally an association of files ending in "`.prj`" should have been set up while installing the development package, so starting the Project Editor application can be achieved by simply double-clicking on "`pb.prj`". After either newly starting Project Editor by opening the data file or by manually loading the project data file "`pb.prj`" from the menu bar of the Project Editor application you will see the *Log page* of the program. In this page messages are eventually displayed while compiling or testing the program to be written.

9.2.1 Setting up a New Project

Before starting the work you must tell the Project Editor application some information about your project, e.g. the name of the executable file which is to be created. This can be achieved by changing the data on the *Linker page* of the notebook control representing the project data.

In Figure 9.4 this page is depicted. The entry field on the top of the page is used to tell the application the name of the executable file. Because in our case the application will be called "`textview.exe`" just enter this name here.

Below this entry field you can choose which of the (standard) libraries you wish to use with this program. If you want to create simple PM programs, just check the box asking you about the *PM Library*, to also make use of the classes contained in `objcdb`, also check the field next to *DBase Library*. The utility classes and the Objective C runtime library are *always* linked to your application because either of the libraries requires them to be part of the executable program. In the text field you can enter additional linker options or libraries manually. For example to add `objects` to the list of the library files enter "`-lobjects`" here; you can also specify any other linker options here. For more information consult the C compiler manuals themselves.

[1]You should not use the Desktop folder or a folder whose name does contain special characters such as spaces as your new project directory because the make program normally will have troubles accessing directories of this kind

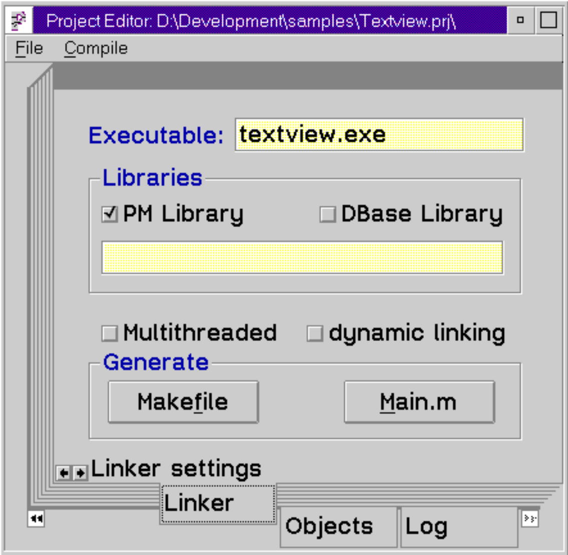


Figure 9.4: Linker page of the project. Here you can set linker flags and generate a "`Makefile`" or the source code file containing the `main()` function.

For a complete description of every of the controls on this notebook page and on the other pages see the *Application Programming Tools Manual*.

For this time, only modify the page you can see to look like the one shown in Figure 9.4.

After changing this settings, or after adding or removing source code files from the project, you have to generate a new "`Makefile`". This can be done by just clicking on the pushbutton `Makefile` also to be found on this notebook page. Before doing so, the current project data has to be saved on disk using *File/Save*. Do not rely on the Project Editor application to use the current data in memory when generating the "`Makefile`". This also applies to compiling and/or testing the application after making some changes. *Always save the project data to disk and generate a new "`Makefile`" before compiling/testing the program.*

9.2.2 Adding a Resource Script

After setting up the project, copy the resource script "`Textview.rc`" which was already created when discussing the first version of the *Textview* program to the project directory. Then switch to the third object page of the project notebook.

To add the resource file to the project, simply drag the resource file from the open project folder where a file object labelled "`Textview.rc`" should be displayed to the objects page reserved for *other objects*, e.g. resource scripts or simple C source code files. See Figure 9.5 where this procedure is depicted.

After adding the resource script file to the project, the corresponding notebook page will reflect the changes immediately. There you can see—as is shown in Figure 9.6—that a new file object has been added to the data area of the page (a simple container control was used for this purpose). The icon indicates that this file is a resource script file.

Thus far a new source code file has been added to the project. After saving the project data file and re-generating a new "`Makefile`" all project data will be updated. Compiling after this stage would create a binary resource file called "`Textview.res`" from the resource script file and

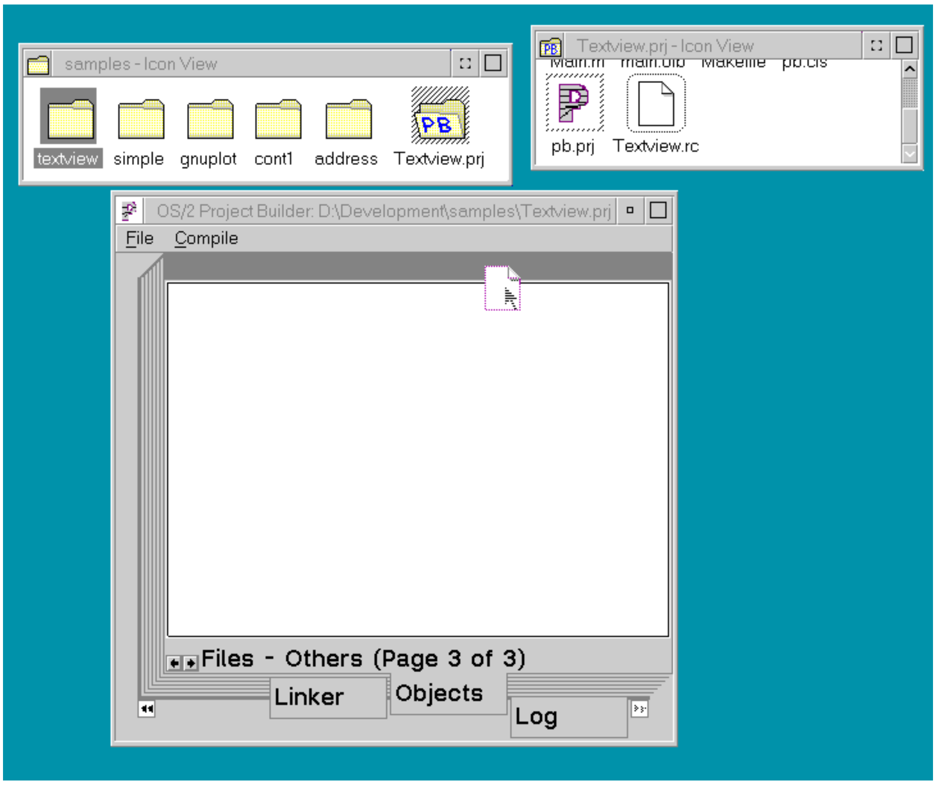


Figure 9.5: Adding a resource script to the project.

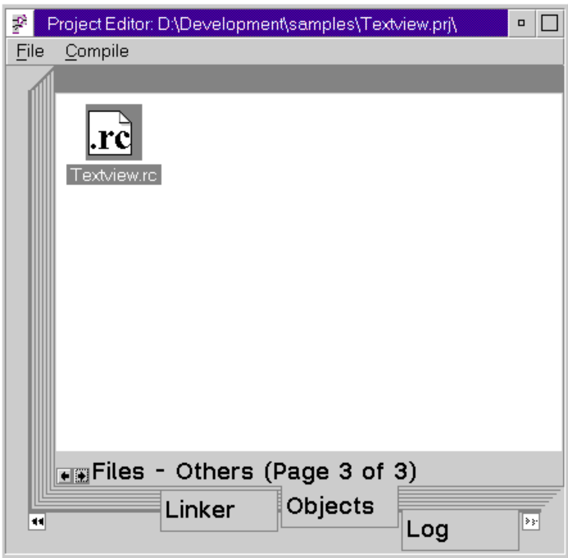


Figure 9.6: Objects page reserved for resource scripts and C source code files after adding a resource script.

cause this file to be linked with the object files—only "`Main.o`" at the moment—when starting compilation.

In the next section creating a new *Objective C class* will be discussed as well as creating a skeleton source code (implementation and header file) for this class.

9.2.3 Creating a Class Template and Generating Source Code

In addition to the "standard features" any project managemant tool should offer—such as managing the files a project consists of—the Project Editor offers help in writing the source code files of your program itself. The classes used by your application must be defined in some central place to be known by the other development tools, i.e, the Interface Editor.

At the moment you can only define the constructs supported by the Interface Editor program, enabling you to graphically create *outlet variables* and *action methods.* After doing so the classes can be used by the graphical development tools, e.g. you can create instances of these classes and connect them to other objects. Additionally the Project Editor application relieves you from the annoying work of creating Objective C interface files and implementation files. A skeleton source code is generated for you automatically and you only have to fill in the method templates. Of course, the newly created class is automatically added to the list of classes managed internally and handled correctly when creating a new "`Makefile`".

9.2.4 How the Constructs are Translated to Source Code

When creating a class you can specify a list of *outlets* and *actions* to be used as the instance variables respectively instance methods of the class. This section will discuss how these structures are translated into source code when generating the interface and implementation files.

Inserting an outlet called `anOutlet` will result in a declaration line in the variables section of the interface of the class:

```
  id anOutlet;
```

As actions are simple instance methods having one parameter of type `(id)`, the translation of an action `-anAction:` yields

```
-anAction: sender;
```

in the header file (interface file) and in an implementation template of

```
-anAction: sender
{
  return self;
}
```

in the implementation file of the class. The only work you will have to do now is to load the implementation file into some text editor and fill in the method template.

9.2.5 Creating the Class Template

In the case of our sample application, only one class which will be called `Controller` is required. The class should be derived directly from `Object`. Two connections to other objects, to the main window of the application and to the multi-line entry field are needed, resulting in two outlets we will call `mainWindow` and `mleWindow`. In addition to that, a single action method to be called from the menu item *Open...*, must be defined. This method will be called `-loadFile:`.

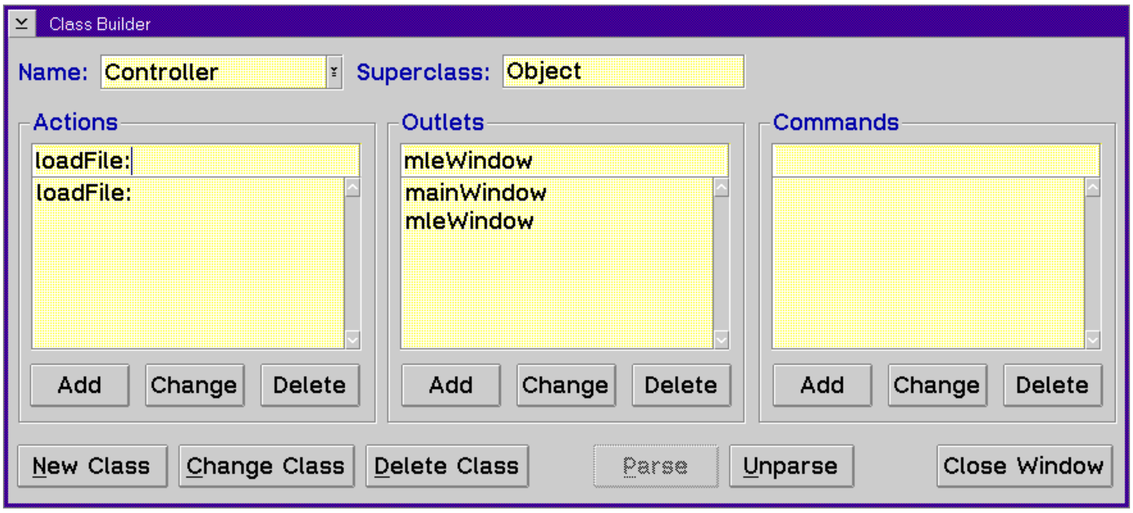


Figure 9.7: Dialog for editing the properties of newly create classes

To create the new class select the menu item *Edit classes* from the *File* menu of the project editor. This will cause a dialog window as depicted in Figure 9.7 to be displayed.

First, select the entry field labeled with *Name* and enter the name of the new class. In our case, just enter `Controller` into this field. Next, you will enter the name of the superclass into the corresponding field, *Superclass*. Because our class will be derived directly from `Object`, it is not necessary to enter the name of the superclass.

After the names of the class and of its superclass have been specified, click on the leftmost button at the bottom of the dialog window shown in Figure 9.7 which is labelled with *New Class*. This will create a new class template in memory. Now you can edit the list of *Actions* and the list of *Outlets*[2].

To insert the single action method, enter the name of the method (including the colon at the end of the method name) in the entry field on top of the corresponding list box. So, just enter `-loadFile:` and depress *Add* below the list box.

The two outlets are inserted alike, by just specifying every outlet's name in the entry field on top of the *Outlets* list box followed by clicking on the *Add* button in the middle section of the dialog window.

After entering the two outlet variables called `mainWindow` and `mleWindow` all information for the newly created class has been set. Normally, you will now create an interface file—containing the class declaration—and a skeleton implementation file. This is achieved by clicking on *Unparse*.

When creating these files for the first time, you will be asked by the Project Editor program if you want to insert the new class into the project. Answering this question (see Figure 9.8) by clicking on *Yes* will cause the interface file "`Controller.h`" to be inserted into the *Headers* page of the *Files* section of the project notebook. Additionally, the class implementation file "`Controller.m`" is added to the list of classes (page *Classes* of section *Files*). When creating a new "`Makefile`", the implementation file is added to the list of files to be compiled and linked to the application program.

Unparse will always create an implementation file only containing method templates. When implementation and/or header files do already exist, you will be warned of this fact by the Project

[2]Normally you should not bother about the third list labelled with *Commands*. This special list is explained in detail in the *Application Programming Tools Manual*.

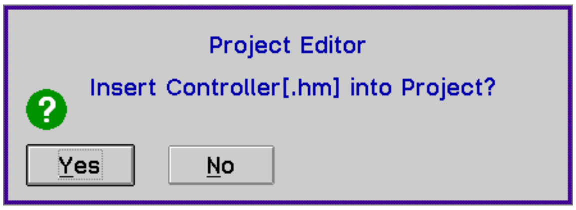


Figure 9.8: The Project Editor application will ask you if the newly created class should be inserted into the project.

Editor application, but if you decide to overwrite these files, all changes you have made to them previously will be lost.

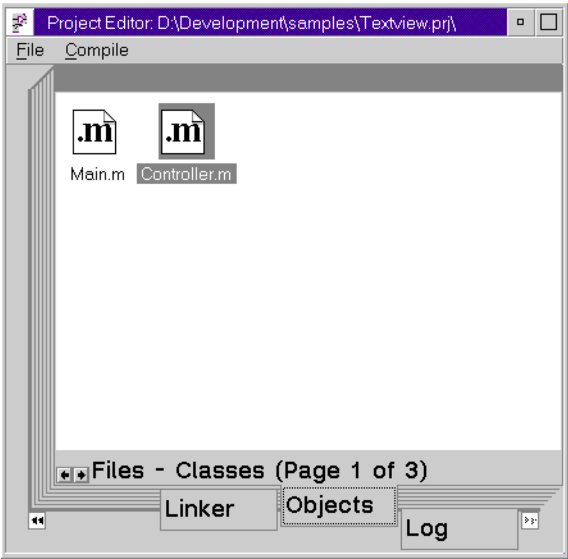


Figure 9.9: Page *Classes* of the *Files* section of the project notebook after creating and inserting a new class called `Controller`

After creating the new class and unparsing the interface declaration and the implementation file you can close the class editor and return to the project notebook. Figure 9.9 shows how the *Classes* page of the *Files* section of the project notebook will look after inserting the newly created class `Controller`—including the interface file "`Controller.h`" and the implementation "`Controller.m`".

Before discussing the design of the user interface of the application and using the newly defined class `Controller` in the project, the automatically generated interface file ("`Controller.h`", Section 9.2.6) and the class implementation file ("`Controller.m`", see Section 9.2.7) are presented.

9.2.6 Interface File of the Class "Controller"

The header file starts with a comment block. After some necessary include statements (the *root header files* for the three libraries are automatically included) the interface declaration for the class follows.

```
/* -----------------------------------------------------------------
```

```
        Project:

        Objective-C interface file for the class Controller

        This file was automatically generated by Project Editor
        COPYRIGHT (C), 1995, Thomas Baier

        Unregistered version

        COPYRIGHT (C), 1996
        ALL RIGHTS RESERVED.

        Date:                           Rev:
        Sat Jan 06 14:33:21 1996


 */

#ifndef _CONTROLLER_H_
#define _CONTROLLER_H_

/*====================================================================
                        Interface of class Controller
====================================================================*/
#include <pm/pm.h>
#include <db/db.h>
#include <util/util.h>

@interface Controller : Object
{
  id      mainWindow;
  id      mleWindow;
}

/* ------------------------- Initialize ------------------------- */

/* ---------------------------- Free ---------------------------- */

/* ---------- Methods for access to Instance Variables ---------- */

/* ----------------------- Public methods ----------------------- */

-loadFile: sender;

/* ---------------------- Private methods ----------------------- */

/* --------------------- Archiving methods ---------------------- */

@end

#endif
```

In the beginning, the instance variables—only the outlets at the moment—are declared, then the declaration of the action methods follows.

The methods are divided into six categories,

1. *Initialize* ... methods for initialization of an object. Normally the names of the methods placed here will have the prefix `-init`.

2. *Free* ... methods for deleting an instance of the class and freeing all resources occupied by the object.

3. *Methods for access to Instance Variables* ... variable access methods. As you should never access instance variables directly from outside the implementation of the object's class, you must provide methods to *read* the information stored and to *update* the information if this is desired.

4. *Public Methods* ... these are methods normally called from other objects; methods of this kind are sometimes called the *services* of the class (or object). Actions defined with the Project Editor application will be placed here.

5. *Private Methods* ... here you should put methods only called by other methods of the object itself. In addition to that, the declarations of delegate methods can be put here. Everything not really "public", i.e., not referenced directly from source code you write for your project, should go here.

6. *Archiving methods* ... the only methods you should put in this section are those designed for writing the object to a stream (`-write:`) or restoring an object from a stream (`-read:`). The only other methods fitting here are the ones also concerned with archiving, e.g. `-awake` or `-awakeFromInterfaceFile`.

The implementation file is organized alike. The skeleton implementation file produced by the Project Editor application is discussed in the next section.

9.2.7 Implementation Skeleton for "Controller"

Just as the interface file, the class implementation starts with a comment block. Thereafter the interface file—"`Controller.h`" in our case—is included.

The declaration of the instance variables is repeated then. Take care to modify this declaration in both the interface file and the implementation file when adding, deleting or renaming instance variables. These actions will also normally require the class information stored in the project file to be updated, but only if action methods or outlet instance variables are concerned.

```
/* -------------------------------------------------------------------

        Project:

        Objective-C source file for the class Controller

        This file was automatically generated by Project Editor
        COPYRIGHT (C), 1995, Thomas Baier

        Unregistered version

        COPYRIGHT (C), 1996
        ALL RIGHTS RESERVED.

        Date:                           Rev:
        Sat Jan 06 14:33:21 1996

------------------------------------------------------------------- */
```

```
/* ------------------------ Include files ------------------------- */

/* ------------------------  Classes used  ------------------------ */


/* -------------------------  Externals  -------------------------- */


/* --------------------------  Defines  --------------------------- */


/* ----------------------  Class variables  ----------------------- */


/*====================================================================
                     Implementation of class Controller
====================================================================*/
#include "Controller.h"

@implementation Controller : Object
{
  id      mainWindow;
  id      mleWindow;
}

/*====================================================================
                              Initialize
====================================================================*/


/*====================================================================
                                 Free
====================================================================*/


/*====================================================================
               Methods for access to Instance Variables
====================================================================*/


/*====================================================================
                            Public methods
====================================================================*/

/*--------------------------------------------------------------------
|-loadFile: sender|

Return self.
--------------------------------------------------------------------*/
-loadFile: sender
{
  return self;
}


/*====================================================================
                           Private methods
```

```
======================================================================*/


/*=====================================================================
                          Archiving methods
======================================================================*/

@end
```

Every action method is preceded by a comment block suitable for putting in required information, as for example a short description of the purpose of the method and the return value.

```
/*---------------------------------------------------------------------
|-loadFile: sender|

Return self.
---------------------------------------------------------------------*/
-loadFile: sender
{
  return self;
}
```

By default, all action methods will return a pointer to the instance of the class itself (`self`). This convention should be followed, if the method exits without error. On error, `nil` should be returned.

New methods should be inserted in the appropriate places in the implementation file. Never forget to update the header file after adding, removing or changing the name of a method or instance variable. In case of adding a new action or outlet, you will also have to change the information stored in the project data.

After the class was created, save the project data and recreate the project's "`Makefile`".

9.3 Creating the User Interface

For the moment you can put aside the Project Editor and relax a bit. Up to now a new project was created from scratch—from a project template in other words—a resource script was added and a new class was defined. The project now consists some files,

1. "`PB.prj`" and "`PB.cls`" storing the project data and the custom classes defined by the user.
2. The "`Makefile`" containing rules on how to compile and link the source code files together to form an executable program and the source code files themselves,
3. "`Textview.rc`", a resource definition file storing information about the application menu requested by the program,
4. "`Main.m`", the source code file containing the `main()` function. This file should normally not be edited by the application programmer.
5. "`Controller.h`" and "`Controller.m`" are the Objective C interface file respectively the class implementation file for the newly created class `Controller`.
6. In addition to these files, there is also a file called "`main.oib`". This file is the primary user interface definition. This section will deal with editing this file and putting the *core* of the program into it.

Figure 9.10: Main user interface definition file "`main.oib`".

Figure 9.10 shows the icon of the user interface file. It can be found in the project folder you created at the beginning.

To start editing the user interface definition, just start the Interface Editor application by double-clicking on the icon of "`main.oib`".

After some seconds, the Interface Editor will be loaded and the windows will be displayed, revealing a user interface similar to the one depicted in Figure 9.11.

In the following the term *interface file* will not be used to refer to file containing an Objective C class interface declaration (a header file); instead of this interface file will be a shortcut for *user interface file*, the kind of file you will create by using the Interface Editor application. For every interface file the Interface Editor program will display a separate window—also containing a menu bar—which will be called the *main window of the interface file.*

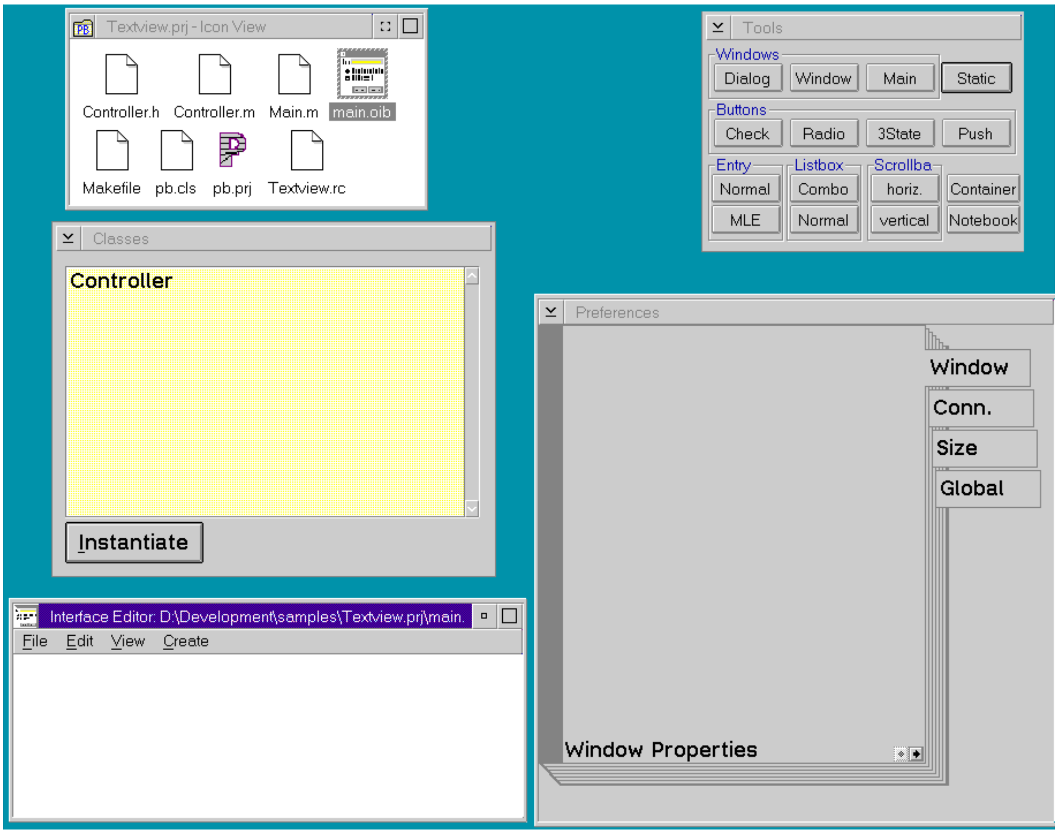


Figure 9.11: User interface of the Interface Editor application. At the bottom the main window displaying the objects currently stored in the interface file (left) and the preferences window (right) are displayed. Above you will find the tools palette providing easy and fast access to most of the user interface objects supported by the program (upper right corner) and a window displaying a list of classes currently known by the application.

When starting the Interface Editor for the first time, only the main window will be displayed. It is recommended to make the other windows visible by choosing the appropriate menu items from the *View* menu. For simple and fast access to the most-used functions, at least the preferences window and the tools palette should be displayed automatically.

In contrast to most other OS/2 applications, the desktop itself will be your work area. Although you might not be accustomed to directly editing your windows on the desktop, this approach has some advantages over putting everything inside the display area of the main window of an application.

As you will remember, the user interface of the application to be created will consist of a simple (main) window having a menu bar and a multi-line entry field covering the whole display area of this window object. Additionally, a single instance of the `Controller` class will be used to respond to user actions.

To create the necessary objects and establish some connections between them, the Interface Editor application will be used. The rest of this section will discuss the look and the use of the windows depicted in Figure 9.11 while doing the work.

9.3.1 Creating Objects—Using the "Class List"

Our journey through the windows of Interface Editor will start with a window labelled with *Class List*. This simple dialog window consists of a single listbox control and a push button. Figure 9.12 shows how the window should look at the moment.

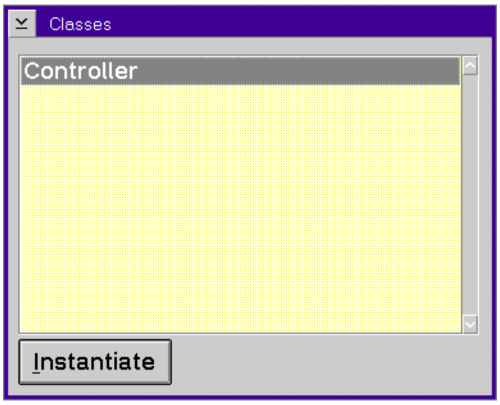


Figure 9.12: *Class List* window displaying the only custom class known by the application at the moment.

In the listbox control all custom classes currently known by the Interface Editor application are shown. When loading an interface file, the program will try to load a class definition file ("`PB.cls`") from the directory the interface file is loaded from.

Using the push button *Instantiate*, you can create instances of the currently selected class in your interface file. Because one instance of the class `Controller` is needed in our application, just click once on the button. As a consequence of this, you can see that a new icon is displayed in the main window of the Interface Editor application (see Figure 9.15 on Page 76).

The class definition file is read in automatically when loading an interface file. If you change the class definition by, e.g., adding a new class or changing classes, you will have to initiate the process of updating manually by choosing the menu item *File/Reload Classes*.

9.3.2 Creating User Interface Objects—The "Tools" Palette

The last section was mainly concerned with instantiating user-defined classes. It was mentioned that every object created from a user-defined class is represented by an object in the main window of the Interface Editor program. Every object displayed there consists of an object symbol and an object title.

While the icon of the object indicates the type of object, e.g. an instance of a user-defined class or a dialog window, the title of the object is a unique identifier for the object itself. These objects are stored in an instance of the class `InterfaceFile` which is provided by the PM library. The objects themselves—or to be more exact, pointers to the associated Objective C objects—can be queried by the application program at run-time using the appropriate instance methods of `InterfaceFile` (e.g. `-objectWithTitle:`) and the title of the object.

Figure 9.13: Object Types currently supported by Interface Editor

Figure 9.13 shows the icons of the object types currently supported by the development environment. Starting with the object displayed at the left margin, you can see a *user-defined object* labelled with `Controller`, three different user interface objects and two objects representing instances of some other often-used classes in an application. Instances of user-defined classes can be created using the *Classes* dialog window discussed in the previous section.

The user interface objects are a *dialog window* (an instance of `StdDialog`) having the title `Dialog`, a *window object*—which is an instance of the class `StdWindow`—labelled with `Window` and an instance of `MainWindow` having the same name as the class it is an instance of (*main window object*). These objects can be created by clicking on one of the push buttons *Dialog*, *Window* or *Main* in the *Windows* section of the *Tools Palette* (see Figure 9.14). This section will be concerned with creating these windows and other user interface control elements.

The other two objects are created by choosing the appropriate menu items from the *Create* menu located in the main window of an interface file as displayed in Interface Editor. The second object from the right is an instance of the *Help* class, the rightmost object represents a *Printer* object.

In future versions creating instances of many other non-user interface classes, as for example from classes provided by database library, will be supported.

Other user interface objects than dialog, standard and main windows, like push buttons or entry fields, are always part of one of the window objects shown in Figure 9.13 and therefore not displayed separately in the main window of the interface file.

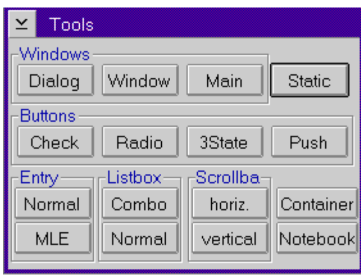


Figure 9.14: The Tools Palette of the Interface Editor application. From here you can choose the user interface objects you want to add to an interface file.

Just alike any program for creating and editing dialogs Interface Editor assists the programmer in designing the user interface of his programs in a very simple way. One of the main advantages over a "normal" dialog editor, creating instances of user defined classes, was already discussed. The more advanced features as, e.g., setting up connections between objects will be dealt with in the next section. Here only the basic creation of the user interface of a program—in our case for the "Textview" application—will be shown.

9.3.3 Creating a Window Object

As you will remember, the user interface of our application will only consist of a simple `MainWindow` containing a multi-line entry field and a menu bar displaying a pull down menu as defined in the resource script "`Textview.rc`". To create an instance of `MainWindow` you must use the *Tools Palette*. If this window (see Figure 9.14) is not already displayed on your desktop, choose *Tools* from the *View* menu.

Figure 9.15: Main window of the Interface Editor application. Two objects have been inserted, an instance of a class called *Controller* and a window object labelled with *Main Window*.

Now, simply click on the push button labelled with *Main* in the topmost row of buttons in the dialog window. Immediately, an empty window object will be displayed on the desktop and a new object is created in the main window of the interface file. This window should now look like the window shown in Figure 9.15.

Before speaking about customization of this main window the multi-line entry field will be created.

9.3.4 Inserting a Multi-Line Entry Field into the Window

To insert the multi-line entry field into the main window just click on the *MLE* button in the *Tools Palette*. Any user interface object—except dialog windows, standard windows and main windows—are inserted into the currently selected window object. To determine which object is currently selected, look at the icons representing the objects in the main window of the interface file. You will notice that exactly one object is selected, making it the target window of every insertion of user interface elements (if it is a window object).

You can either change the current selection by clicking with mouse button one—which normally is the left mouse button—on the icon representing the the object or you can alternatively select the window object itself if it is currently displayed on the desktop.

For more information on selection techniques and manipulating the ordinary object properties see the *Application Programming Tools Manual*.

9.3.5 Editing the Object Properties

Up to now a main window has been created with a multi-line entry field in its display area. In the previous chapters creating a window object required to specify some information at initialization, e.g. the PM identifier and some flags determining the look and feel of the object. As the main window and the multi-line entry field were not created from source code, there must be some other way, some graphical way to set the object properties.

To edit the properties of the main window just select it and then have a look at the *Preferences* window normally displayed in the bottom right corner of the desktop (choose *View/Preferences* if the window is not already displayed on the screen).

In this window a notebook control is displayed providing some (editable) information about the currently selected object. Every time you select another object (or user interface control in a window) the display area of the notebook pages is updated to reflect the correct set of properties and the actual settings for the currently selected object. The first preferences page for a main window is shown in Figure 9.16.

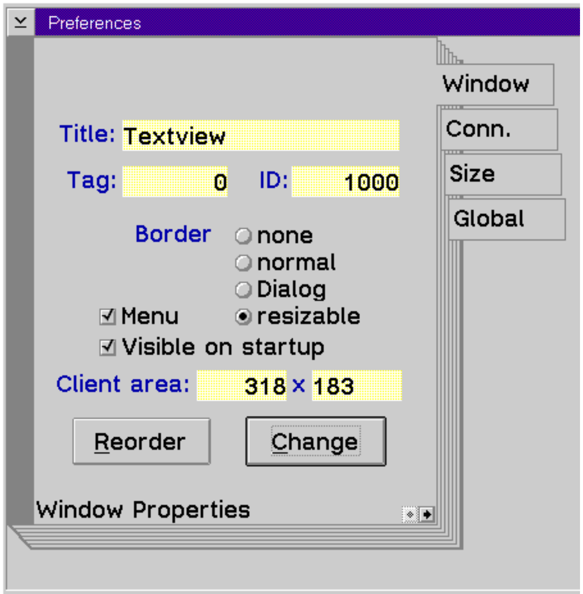


Figure 9.16: First Page of the Window Preferences of a Main Window. This page contains controls for the most commonly used properties of such an object.

The following description assumes that the main window created previously is selected. On the first page of the notebook in the preferences window you can find some generic information about the selected object, such as the text displayed in the title bar of the window (*Title*), its PM id (*ID*) and some flags.

For the moment, just change the title text (in the *Preferences* window, not in the main window of the interface file) to `Textview` and its PM id to `1000`. After doing so, check the boxes placed on the left sides of *Menu* and *Visible on startup*. Then click on *Change* in the lower right corner of the notebook page to actually perform the changes.

These actions (the dialog as shown after doing the changes is depicted in Figure 9.16) will cause the window title to be changed to `Textview` and the PM resource identifier to `1000`. In addition the flag `FCF_MENU` is set and *Visible on startup* causes the window to be displayed automatically when the interface file is loaded by the application (normally when the application is started). Most of the changes made in the *Preferences* window are displayed immediately. Only those settings requiring access to a resource file (as it is the case for menu resources at the moment) will only be activated when the application program has been compiled, linked and is being started.

Next click on the small arrow at the bottom of the page pointing to the right side to switch to the next notebook page, the *secondary window preferences*. On this page, as you can see in Figure 9.17, you can change most of the other flags normally specified when calling `-initWithId:andFlags:` "manually" by simply checking or unchecking some of the boxes left to the description

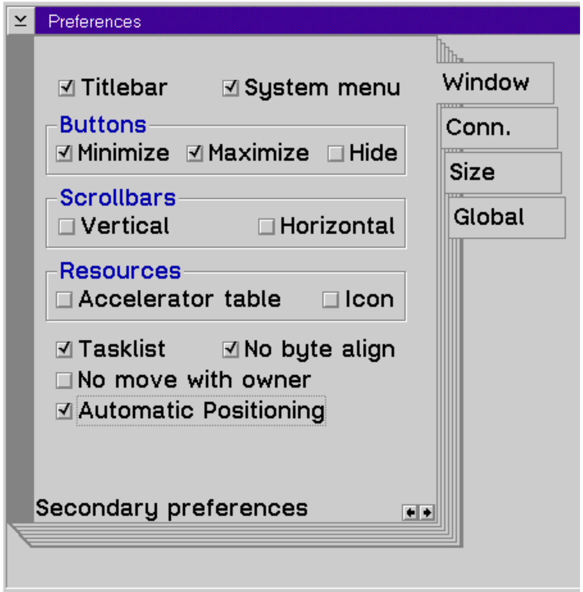


Figure 9.17: Secondary Window Preferences Page of a Main Window

In addition to the flags set automatically you should check the boxes next to *Tasklist* and *Automatic Positioning*. It should be clear what effect setting these flags will have on the window object.

For a more detailed description of the available flags see the *Application Programming Tools Manual* and the *Reference Manual*.

After correctly setting up the properties of the main window, the multi-line entry field must be dealt with. To display the preferences of this object, click once with button one on the object as displayed inside the main window. You will notice that a thin border is drawn around the object indicating that now this object is the selected user interface control in its parent window.

Immediately, a new *Window Properties* page is displayed in the *Preferences* window already reflecting the current settings of the multi-line entry field.

You should make sure that the check box *read-only* is in checked state, so that the settings of the multi-line entry field are the same as shown in the window in Figure 9.18. Additionally, you should add horizontal and vertical scrollbars by activation the appropriate check boxes.

You will have noticed that you can find a button labelled *Change* at the bottom of some of the preferences pages. You are only required to depress this button when editing the data in one of the entry fields on a preferences page. Changes to a check box or a radio button are immediately performed on the window object without requiring any further user action.

Now the user whole user interface is created, as you would expect when using a normal dialog editor. In the next section you will learn about making connections between the objects in an interface file and how this can simplify your life when creating OS/2 programs.

9.3.6 Objective C—Connecting Objects

In previous chapters the design philosophy of the library package was introduced. Every user action is normally processed by one of the standard PM classes in the beginning. If an object

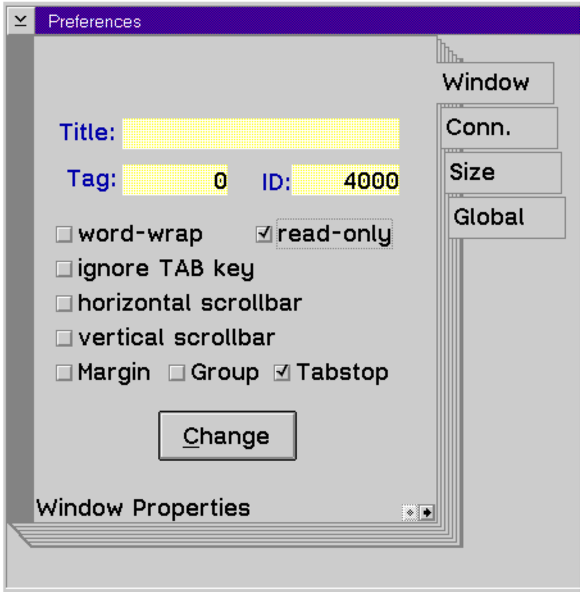


Figure 9.18: *Window Properties* Page of a Multi-Line Entry Field.

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.

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.

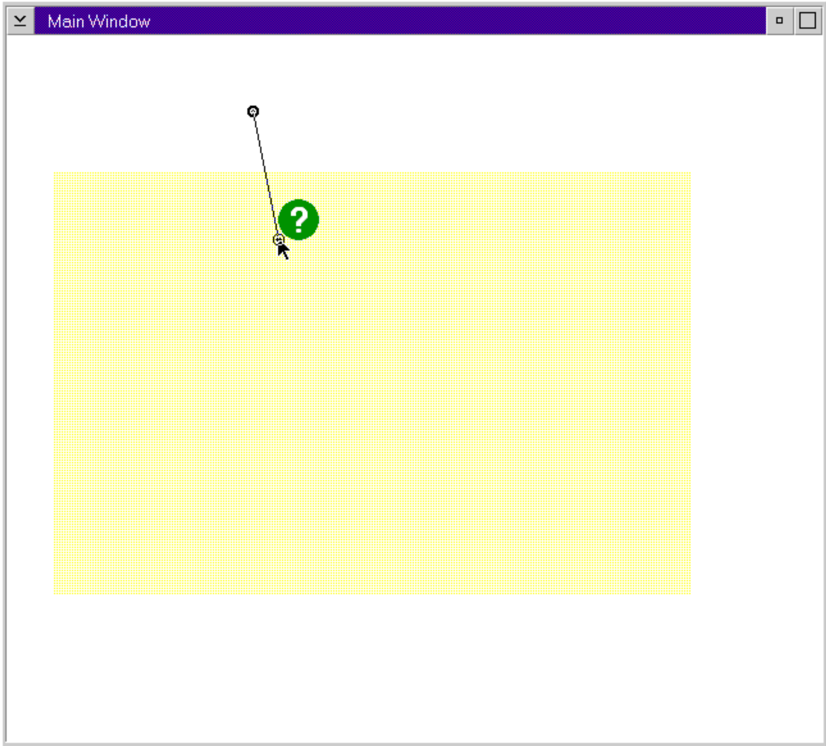


Figure 9.19: Connecting the Main Window with the Multi-line Entry Field

In addition to implementing a delegate method called `-windowDidResize:` you can use the special outlet instance variable `clientWindow` to cause a window object to always fill the complete client area of a main window. To automatically adapt the size of the multi-line entry field to the size of its parent window (the "Textview" main window in our case), make an outlet connection from the main window to the multi-line entry field using the outlet `clientWindow`. Figure 9.19 shows how connecting objects will look.

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 9.20.

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.

The list box will display all available outlet instance variables and commands. You can differ

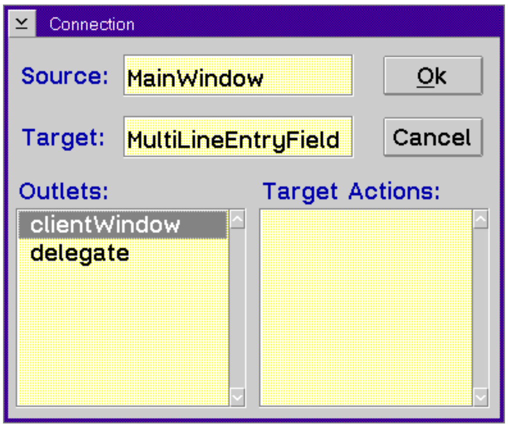


Figure 9.20: Dialog for connecting the Main Window of the "Textview" application with the Multi-line Entry Field

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. Deleting a connection is achieved by using the *Preferences* window. All connections are displayed on a page labelled with *Object Connections* (the tab text of this page is *Conn.*). Using the *Disconnect* button displayed there you can delete a connection. This is described in more detail in the *Application Programming Tools Manual*.

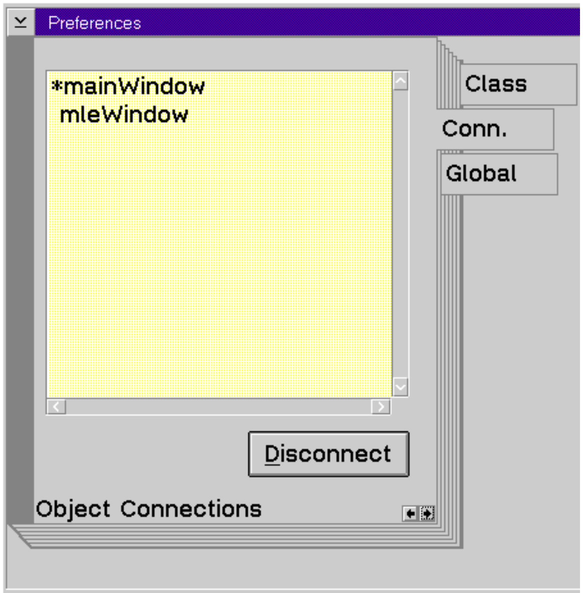


Figure 9.21: *Object Connections* page of the *Preferences* window for the *Controller* object after making the first outlet connection

In addition to the single outlet connection already established, the two outlet instance variables of the `Controller` object must be initialized. Make an outlet connection for the variable `mainWindow` from *Controller* to the main window and another connection for `mleWindow` from the same source object to the multi-line entry field.

To get the right sense for these object connections, have a look at Figure 9.21. There the *Object Connections* page for the *Controller* object after making the first outlet connection is shown. You can see that the name of one outlet instance variable is preceded by a `*`, marking it as being the source of a connection.

After connecting the object, the work with the Interface Editor program is finished, you can now save the data by using *File/Save* and exit the program. Now we will return to the Project Editor and start editing the source code.

9.4 Editing the Source Code of the Program

You should have saved your interface file by now and eventually closed the Interface Editor program—you will not need it any more when finishing this rewrite of "Textview".

After bringing the Project Editor to the front again, you can see two file icons on the *Classes* page of the project notebook. These icons—as depicted in Figure 9.22—represent the two implementation files part of the project, the file containing the `main()` function, "`Main.m`", and the implementation of our *Controller* class, "`Controller.m`".

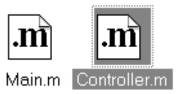


Figure 9.22: Icons in the *Classes* Page of the Project Editor Project Notebook

As the main project source code file is suitable for most applications you will develop, including "Textview", we will only have to edit the class implementation file of *Controller*. To start editing, simply double-click on the icon of "`Controller.m`" in the Project Editor application. The designated text editor for class implementation files—see the *Application Programming Tools Manual* for information on customizing the external programs—will be started and the implementation file "`Controller.m`" is loaded automatically.

The only method currently defined for *Controller* is the action method `-loadFile:`. The method template created by the Project Editor automatically must be filled in to provide the expected functionality. As the implementation of this method was already discussed when describing the "Textview" application for the first time, no further explanation is required. The source code should be modified to look like this:

```
-loadFile: sender
{
  FileDlg *fileDlg = [[FileDlg alloc] initForOpen: "Open File..."
                      withFilter: "*.*"];
  if (([fileDlg runModalFor: mainWindow]) && ([fileDlg result] == DID_OK))
    [self readAndDisplayFile: [fileDlg fileName]];

  [fileDlg free];

  return self;
}
```

The `-readAndDisplayFile:` method was already discussed previously. Because it is only used internally by objects of the class `Controller`, the source code should be placed in the section reserved for *private methods*.

```
-readAndDisplayFile: (char *) fileName
{
  char        *title;
  FILE        *inputFile;
  struct stat  statbuffer;
  char        *contents;

  if (stat (fileName,&statbuffer) < 0) { /* check file */

    [mleWindow setText: ""];
    [mainWindow setTitle: "Textview: no file loaded"];

    return nil;
  }

  /*
   * open file and read contents to buffer
   */
  inputFile = fopen (fileName,"r"); /* open text file read-only */

  contents = (char *) malloc (statbuffer.st_size + 1); /* allocate buffer */
  fread (contents,statbuffer.st_size,1,inputFile); /* read contents of file */

  /*
   * calculate title of window and set it
   */
  title = (char *) malloc (11 + /* this is the length of "Textview: " +
                                   NULL */
                           strlen (fileName)); /* allocate buffer
                                                  for title */
  sprintf (title,"Textview: %s",fileName); /* fill title buffer */
  [mleWindow setText: contents]; /* display contents of file */
  [mainWindow setTitle: title];

  free (title);
  free (contents);

  fclose (inputFile); /* close file */

  return self;
}
```

Defining a new method requires—just for the sake of a good programming practice—to insert a method declaration in the appropriate header file. To do this, double-click on the icon representing "`Controller.h`" in the *Headers* section of the project notebook and add

```
-readAndDisplayFile: (char *) fileName;
```

to the interface declaration of the class `Controller` (also put the declaration in the *private methods* section).

What is still to do now is to bind the menu items to the appropriate objects and methods. At the moment there is no way to bind menu items graphically to action methods, therefore your application must issue some `-bindCommand:withObject:selector:` statements in an appropriate place in the source code itself. Remember, that this limitation does only apply to binding commands originating from menu items. Commands initiated by e.g. button controls can be dealt with by just setting up a command-action binding using the Interface Editor application.

Normally, you would place setting up these bindings into the initialization method of the `Controller` class. The problem is, that at the time the `-init` method is processed, the outlet bindings originating from the object itself are not initialized. Therefore there must be a method called *after* completely initializing an object and all in- and outgoing command-action and outlet bindings. This method is called `-awakeFromInterfaceFile`. For every object created from a user-defined class using the Interface Editor, this method is called—if implemented—after loading and creating it from an `InterfaceFile`. The only thing you must do to perform a custom initialization for an object *after* all bindings have been set up correctly, is to write an `-awakeFromInterfaceFile` method for your classes.

In the case of `Controller`, the method will simply bind the menu items *Open...* and *Exit* with the appropriate methods of the target objects. An implementation—this time to be put into the section reserved for the *Archiving methods* at the end of the class implementation file—will just consist of two calls to `-bindCommand:withObject:selector:`:

```
-awakeFromInterfaceFile
{
  [mainWindow bindCommand: 2001 withObject: self
               selector: @selector (loadFile:)];
  [mainWindow bindCommand: 2002 withObject: mainWindow
               selector: @selector (performClose:)];

  return self;
}
```

Just as before, put the prototype declaration of the method,

```
-awakeFromInterfaceFile;
```

into the header file "`Controller.h`".

After these changes to the interface and the implementation of the class have been made, everything is done and we can use Project Editor to compile and start the program.

9.5 Compiling and Linking the Application

To compile and link the application first have a look at Figure 9.23. There you can see the Project Editor application and one of its menus. In the compile menu you can find some menu items all concerned with compiling and starting the application you are developing.

Using

- *Build* you simply compile and link the application. This is equivalent to typing "`make`" on the command line. The object files are created with full optimization on (compiler flag `-O2`).
- *Debug* is used to build an executable file including debug code. You can use `gdb` to debug your application. Always keep in mind, that debugging of PM applications requires `gdb` to be started in a full-screen session.

  I experienced some troubles when using normal *GNU make* for compiling using *debug*. My version complains about too many open files. I will try to find out, if my version is just a bit out of date, or there is a bug I can fix.
- *Clean* will remove all object files, compiled resource files and the executable file. Additionally backup files as created by *GNU Emacs* (ending with a) will be deleted. This is recommended after compilation was stopped using
- *Abort*. This menu item will stop compilation, or, in case your program is already started, will terminate the program.

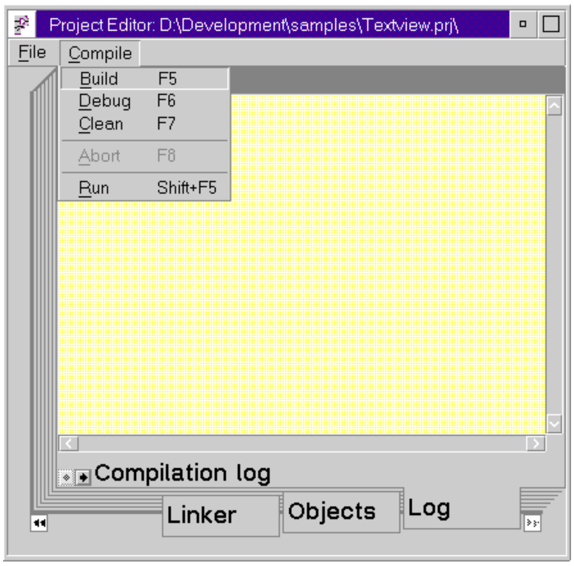


Figure 9.23: Project Editor Application with the *Compile* Menu being open. To compile and link the project, just choose *Build*, to start the program, click on *Run*

- *Run* is used to start the application. If necessary, *Build* is executed automatically before starting the executable file.

When compiling, the *Log Page* of the project data is automatically shown. Here error and status messages are displayed while compiling and linking. Figure 9.24 shows this *Log Page* after compiling the current "Textview" project.

9.6 Creating an Application—Step by Step

In theory, you should now be able to start a project on your own and utilize the visual development tools shipped with the library package. To provide some help, here the necessary steps for creating and editing a project are repeated once more. You can see this as some kind of *recipe* for creating your own applications.

1. Create a new project by simply dragging a project template to your work directory.
2. Open the newly created project folder and start the Project Editor by double-clicking on the project icon ("`PB.prj`").
3. Now edit the settings on the *Linker Page* of the project notebook. You must at least set the name of the executable file and check whether all library files required by your application are specified.
4. Open the *Class Editor* and create the custom classes you intend to use. *Unparse* a source code template for every of the new classes and then save the project.
5. Now create a new "`Makefile`" and start editing the main interface file "`main.oib`" by simply double-clicking on its icon in the project folder.
6. Create the user interface elements you want to use and the instances of the user-defined classes your application will require.

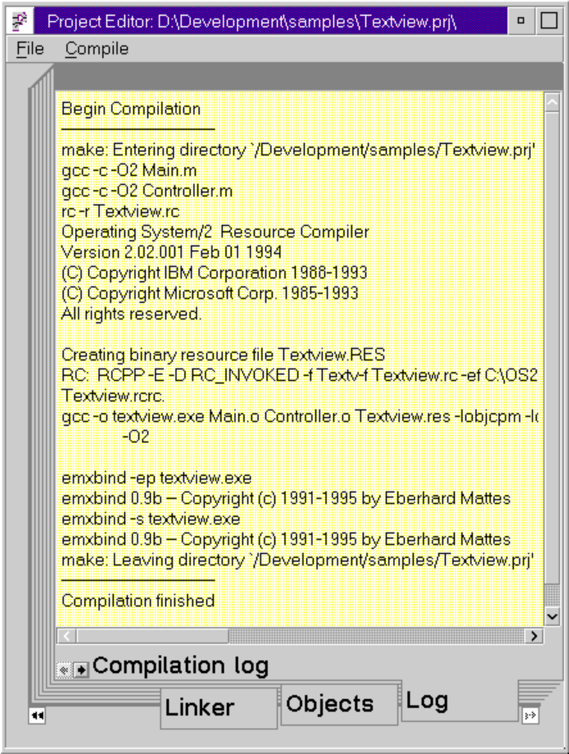


Figure 9.24: Log Page of the Project Editor Program after successful compilation of the Project. You can now simply choose *Compile/Run* to start the program you created.

7. Then set up all command-action and outlet bindings and save the interface file to disk.

8. In the end, edit the source code and fill in the method templates. Add new methods, save the files to disk and choose *Compile/Build* from the menu bar of the Project Editor.

9. After compilation succeeded, you can start and test your application by choosing *Compile/Run* from the menu bar.

More information about the tools can be found in the *Application Programming Tools Manual*. The libraries are fully documented in the *Reference Manual*. Also have a look at the sample programs in `\usr\samples`. There you can find additional information about programming and design. To find out about using Project Editor and Interface Editor together with the database library, check `\usr\samples\DBView.prj` and carefully examine the source code. It's really simple, you will see.

Chapter 10

Debugging Objective C Programs

No matter how good the development system is, you are using—that's not to say, this one is perfect—or how good and experienced you are in developing applications, you will of course produce erroneous source code in the beginning. This chapter will try to mention some tricks supporting you in finding these errors.

First, let me say, that in case of an application not performing as expected by you, the most useful tool to localize the errors is a debugger. The *emx* development tools contain a very good—even if it is not that simple and intuitive to use as you would expect for an OS/2 program—debugger called *gdb*, the *GNU Debugger*. But for most simple cases, using *gdb* is just a bit too much.

10.1 Using "Console"

When your program just silently[1] crashes, in most cases you just made some simple error the Objective C runtime library complains of.

In the current version of the Objective C runtime library, the library functions will print a short description of the error which occured to *stderr* and stop the program.

The only problem is that normally *stderr* is connected with the output area of a shell window, and therefore such error messages can't be seen when writing PM applications. The only possibility to view such messages is to redirect *stderr* to some file or device other than the standard output device.

If you started your application from the Project Editor, *stdout* as well as *stderr* will be connected to the *Log Page* of Project Editor. There you can see the messages printed by the runtime library.

So, if for example you will see something like

```
error: InterfaceFile (instance)
InterfaceFile does not recognize unknownMethod
Abnormal program termination
core dumped
```

this means that your program tried to call a method not recognized by the receiving object. In this special case, the error occured with an instance of `InterfaceFile`. The program tried to call the method `-unknownMethod`, which `InterfaceFile` cannot respond to. Therefore the runtime library printed the error message to *stderr* and exited the program.

[1]silently means "without OS/2 putting some message box onto the desktop informing you of some `SYS****` error"

Although this does not show where in your program the error was made, you will surely be able to localize the error by just thinking on where and how you used instances of the class issuing the error.

For applications not started from within the Project Editor, a small program is provided, which just displays everything sent to a special file called "`\pipe\console`". After starting the program *Console*, this (virtual) file is created and you can write text data to it. To display the messages printed to *stderr* by your program in the *Console* window, you will have to *redirect stderr* to "`\pipe\console`". Starting an application called "`test.exe`" and redirecting *stderr* can be accomplished by typing

```
test 2> \pipe\console
```

in the directory "`test.exe`" resides.

Redirecting *stderr* can also provide useful when your application program wants to print some debugging information. To write strings or other data to the *Console* window after *stderr* has been redirected to `\pipe\console`, use

```
fprintf (stderr,...);
```

in your programs.

For more complex programs or errors, you are not able to localize within some seconds, you should use the debugger *gdb*.

10.2 Using "gdb"

To be able to use *gdb* to debug your applications, you must create *debugging code* while compiling and linking your program. This is done automatically when you specify the option `-g` on the `gcc` command line used for compiling and linking.

To preserve debugging information, you may not *strip* the symbol table from the executable file (`emxbind -s *`) after linking. Choosing *Compile/Debug* from the menu bar of the Project Editor will compile all *modified* source code files and create an executable file suited for debugging called "`debug.exe`". If you experience problems, i.e., the files you want to debug are not automatically recompiled, choose *Compile/Clean* before using *Compile/Debug*.

After the executable file including debug information has been created, start *gdb* in an OS/2 full-screen session[2] by typing

```
gdb debug.exe
```

gdb does not speak Objective C at the moment, that means, the debugger does not know the syntax of the Objective C language. Therefore, you have to debug your applications just as if they were written in plain C.

The methods of your classes are represented as normal C functions. The names of the functions are constructed from

- the type of the method, which is either an instance or a factory (class) method,
- the name of the class the method belongs to, and
- the name of the method itself.

[2]You *must* start *gdb* in an OS/2 full-screen session to debug PM applications, otherwise your system will hang while debugging and you will have to reboot

Every name is of the form

```
_<type>_<name of class>__<name of method>
```

where `<type>` is either `i` for an instance method or `c` for a factory method. In the method name, every colon (`:`) is replaced by an underscore (`_`). To clarify this form of notation, a simple example will be presented:

To set a breakpoint in the method `-loadFile:` of the `Controller` class you would use the `b` (break) command of *gdb*. So, simply type

```
b _i_Controller__loadFile_
```

at the *gdb* command prompt. For a class method called `+create:andInsertInto:`, you would have to type

```
b _c_Controller__create_andInsertInto_
```

Instance variables are stored in some kind of `struct`. `self` is a pointer to this structure, so you could print the instance variable `text` of the current object by typing

```
p self->text
```

Which object is the current object just depends on the function you are debugging.

More information on debugging Objective C programs can be found in the Objective C FAQ.

Appendix A

Literature

If you are searching for good books about the programming language Objective C itself, and you have access to any machine running NEXTSTEP, try reading the according sections of the NEXTDEVELOPER manual pages. An easy to understand document about Objective C and its rootclass can be found there.

Another good introduction into Objective C is a book by Brad J. Cox, who specified the language itself, *Object-Oriented Programming, An Evolutionary Approach*, second edition. Addison Wesley, 1991.

German users should look for a copy of the December 1994 issue of the magazine *iX*. You can find a (mostly) good overview of Objective C and the implementation found with GCC.

Some information concerning Objective C or special questions towards using this language can be found in `comp.lang.objective-c`. An FAQ on Objective C in general and on the GNU implementation of this language is posted regularly. General information in the world wide web can be found at

- `http://www.marble.com/~dekorte/dekorte/Objective-C` (Objective C "home page"),
- `http://www.geom.umn.edu/software/w3kit/overview/objective-c.html` (a short introduction into Objective C) and many other places in the web.

Work on an Objective C class library providing some Smalltalk-like classes is being done now. Check for an OS/2 port of `libobjects` on Hobbes[1]. This library provides many classes simplifying handling of object storage (e.g. List classes, HashTable,...).

At the moment, it is recommended to read some documentation about PM programming. The documentation for this toolbox and the classes themselves are not as complete, as they will be in the near future. Nevertheless, they are quite usable to create some simple—and by capturing some OS/2 PM messages—also more complex Presentation Manager applications. To find out more about "pure PM programming" get the issues of the Electronic Developer's Magazine, which also can be found on Hobbes. Most information ever needed for PM programming can be in the documentation accompanying the *IBM OS/2 developer's toolkit*. An introduction into PM programming, its concepts and the API functions provided by OS/2 can also be found in one of the Redbooks, *OS/2 Version 2.0; Volume 4: Application Development*. This book can be found in *.INF*-format on Hobbes.

Before sending any questions to me, be sure to read all of this manual and the reference manual. Also have a look at the sample programs, which can be found in `\usr\samples`. Some of the samples stored there are not described in this manual, but can contain some information, you might need.

[1] anonymous ftp server `hobbes.nmsu.edu`