OS/2 PM and database library

Version 0.6

Tutorial

January 1995

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 simple database library is provided.

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*.

When 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 library* ··· You can find special information about the provided classes and methods in the Reference Manual.
- *The Database Builder Utility* ··· Read the appropriate sections in the Application Programming Tools Manual.
- *Literature* ··· Look in the Literature section of this Manual.

Contents

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 used object oriented programming language today is *C++*.

Because of the mostly static binding and it's 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 it's features, such as dynamic binding, messaging,... it is better suited for developing complex applications using a graphical user interface with PM.

There's another object oriented programming language, which is as easy to learn as pure C (because it's not much more than C itself), but supports dynamic binding just alike Smalltalk. 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. All two ports of GCC, the EMX port, and the native port called GCC/2 support this language.

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, 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.

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 library. As you will see in 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 doing any *real work* the program must do some initialization, which means it has to allocate all necessary resources to run, it has to *register* itself at PM.

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

So, let's 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,
  .
  .
  [application run];
  .
  .
  .
  [application free];
}
```

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 an object.

2.2 A simple application

O.K. to show a complete PM application I'll show you a program that just creates a standard window, waits until this window gets closed by the user and then terminates. At first, again, the standard C version, 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;
```

In addition to inititializing an application object, the main window is created as an instance of `StdWindow`. 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>` and `<objc/os2emx.h>`. The patches are based on the files `\emx\include\os2.h` and `\emx\include\os2emx.h` from *emx0.8h*.

When using the Database library, you have to include `<db/db.h>`.

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

If you encounter problems compiling any of the samples, check, if the file `TypedStream.h` exists in `\emx\include\objc`. This file is part of the EMX port of GCC. After installing a new GCC version, I found out, this file had been renamed to `\emx\include\objc\typedstr.h` to match the FAT file name conventions. So the include file could not be found by the Interface declaration file for the `Object` class. Just rename `typedstr.h` to `TypedStream.h` in the directory `\emx\include\objc`.

2.4 Compilation

To compile programs using the PM class library just link the executable file with the class library file *and* the Objective C runtime library.

If you save the above example in a file called `test.m`, type the following to produce an executable PM application called `text.exe`:

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

After linking and setting the application type you can strip all debug symbols off the executable file by using the `-s` option of `emxbind`. `emxbind -s test.exe` strips all debug information.

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 24.

Chapter 3

A simple File-Browser

This chapter describes a simple application, which does something useful. It's 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`.

The window should be resizable and it's 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 calling the program, 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 file to the buffer area.
- If all is o.k., create the application instance and a window. Insert a *multi line entryfield* into the window, where the text will be displayed.
- load the text buffer in memory to the display area of the multi line entryfield.

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:

```
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 as follows:

```
/*
 * create app instance and window,
 * create MLE for text display
 */
application = [[StdApp alloc] init]; /* initialize
                                        application
                                        object */
window = [[StdWindow alloc] initWithId: 1000
                              andFlags: FCF_SIZEBORDER];
             /* 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 standard 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`).

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 = [[StdWindow alloc] initWithId: 1000
                                andFlags: FCF_SIZEBORDER];
              /* create main window */

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

  mle = [[MultiLineEntryField alloc]
```

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

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

Delegation is used by some objects in this library – not as many as there will be soon, but at least the two classes `StdWindow` and `StdDialog`, both representing some kind of main window, make use of it.

Using the method `setDelegate:` you can assign a special object, implementing some *delegate functions*, as the delegate object of an instance of `StdWindow` 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 it's 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.

```
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 = [[StdWindow alloc] initWithId: 1000
                              andFlags: FCF_SIZEBORDER];
                                 /* 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 */
```

```
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 *~
```

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. Application then can 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.

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...* · · · to open and display a textfile
- *Exit* · · · to close the application window and exit

The definition of these menu items are as follows:

```
MENU 1000
{
  SUBMENU      "~File",                     2000
  {
    MENUITEM   "Open...",                   2001
    MENUITEM                                SEPARATOR
    MENUITEM   "Exit",                      2002
  }
}
```

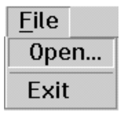


Figure 4.1: Simple menu for "Textview"

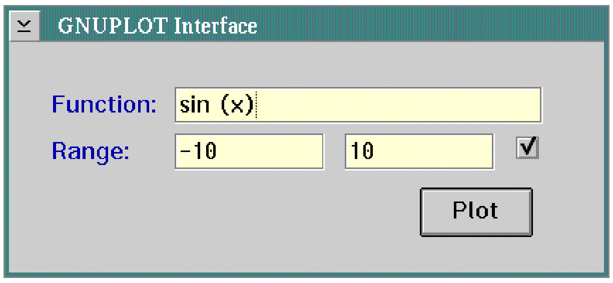


Figure 4.2: Simple PM interface to "Gnuplot"

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 entryfields, a checkbox and a pushbutton.

The first entryfield 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 entryfields and the checkbox are computed and the function is plotted.

Figure 4.2 shows how the dialog looks.

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 does the reading from the entryfields and 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()
{
  [super init];
  gnuplot = popen ("gnuplot.exe","w");
  return self;
}
```

`init` first initializes it's 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 it's instance by calling the `free` method of it's superclass.

The following source code for the method `plot:` is a bit more complicated. Using the `findFromID:` method of `sender`, pointers to the entryfield 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;
}

@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 above file 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`.

```
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
```

Chapter 5

Using the database library

In contrast to the previous chapters, this chapter does not describe how to write Presentation Manager programs. Here, a simple overview of the database library is presented. To simplify things, the first application making use of the database features has no PM interface. The next chapter describes how to integrate both libraries, for PM and 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. This can be accomplished by specifying `-lobjcdb` when creating the program with *gcc*.

5.2 Accessing a DBase III file

As 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 very small database files, at least indexing will be provided soon, to grant real fast, non-linear access to all records stored in the data files.

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 carefull 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;

   .
   .
   .
```

```
#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] string],
                         [[myDBFile field:1] string]);
  }

  [myDBFile free];
}
```

Using `readRecord:` all records are read and by using the `string` 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 -lobjc
```

As you may have noticed, if you compiled and started the application, the name and the phone number of Michael is also printed. It seams the library does not notice, when a record is marked as deleted.

Using `readRecord:` the application program can access all stored records, even those, which have been marked as deleted. There's a method 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 would be printed, which are not marked as deleted. Look at `dbtest2.m` to see the whole source code including this modification.

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

This example just reads each 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 `FALSE`, if there's no active record in the whole database.

`findNext` then searches for the next appearance of an active record in the file. If no more active records are in the database, `FALSE` is returned.

Rewriting the application `dbtest2.m` to `dbtest3.m` utilizing these methods would look like this:

Chapter 6

Modifying data

As shown in the last chapter, opening a database file and reading records from them is quite simple. Some pages ago, I mentioned, that the database file is opened for reading *and* writing. This chapter describes 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
- mark deleted records as active

For this purpose, a new class is defined, called AddressDatabase, which can handle all these functions als instance methods. The interface definition 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 it has one instance variable, called `database`, which holds the `DBFile` instance used for all operations.

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

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 clear enough, what these simple methods are doing, so no more explanation has to be done.

6.2 Printing all records in the database file

printInfo is used to read all active records and print their contents to stdout. This is accomplished by using the following code:

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

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

```
  }

  [database delete];
  [database replace];

  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] string],
      [[database field:1] string]);
    }
  }

  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];
  [database replace];

  return self;
}
```

```
}
```

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

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

Note that this application does very little error-checking. It's really not written as an end-user application, but to demonstrate the usage of the diverse methods provided by the `DBFile` class.

As you can see, it's 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 shows how to combine the PM library with 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 using DBase III files. In addition to this the program provides a framework for an address database maybe useful to some people.

The application will store records with a format specified in table 7.1.

The main window of the application will contain a single listbox, only displaying the first field of each 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 next page shows the main window of the address database displaying some records.

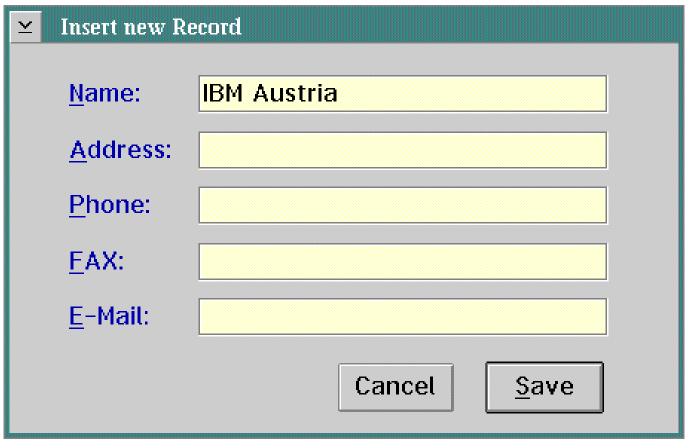
7.2 Application menu

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 called *Exit*.

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



Name: IBM Austria
Address:
Phone:
FAX:
E-Mail:

Figure 7.2: Dialog window to add a new record

In addition to the 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 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 48 for an example main window as created and filled by this method.

The `Controller` object used in the application is created and initialized by the `main ()` function of the program. This function also creates the main window 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 differ only in the dialog title. Therefor 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 these are initialized.

```
- 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 lower 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 deletes 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 findRecordAt: selected] recNo]];

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

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

`info:` is used to display a dialog window previously allocated (`infoRecord`) and fill the entry fields with the data associated with the selected record.

The source code for the *interesting* parts of the implementation is shown below:

```
- info: sender
{
  .
  .
```

```
    free (faxBuffer);
    free (emailBuffer);
  }

  return self;
}
```

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

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.

So the lines in `replace:` copying the record buffer and replacing the contents of the record, including the modification of the item in the listbox must be changed to

```
    [database append];
    [recordList insertRecord: [[DBRecord alloc] initForDatabase: database]];

    [nameListBox insertItem: LIT_END text: nameBuffer];
```

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

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. Don't forget to update all storage structures (record list, data file, listbox) every time you modify one of the records, or add or delete one.

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

7.4.4 Delegate methods

The first of the delegate methods is called `closeApp:`. It is called by the menu item `Exit` to terminate the application. The implementation of this method is shown below. Because of it's simplicity, there is no need to describe what is done here.

```
- closeApp: sender
{
  WinPostMsg ([sender window],WM_CLOSE,0L,0L);
  return self;
}
```

As shown in section 3.6 on page 21, the following simple method is called every time the size of the main window is changed by the user. It's 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;
}
```

Chapter 8

Using the Container Class

One of the most interesting window classes provided by PM 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 forms, as icons, as text, as a combination of both, or similar to multi-column listboxes, 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 section, starting at page 60 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 must be encapsulated in Objective C objects.

The information presented in this section assumes, you already have a simple PM application including a main window. The container, which will be used here, will fill the whole client part 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 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];
```

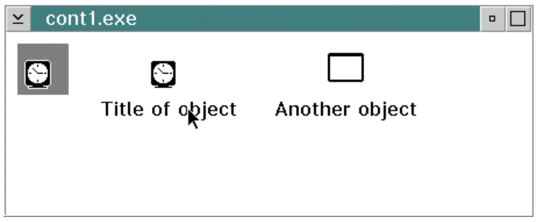


Figure 8.1: First container sample "cont1.exe"

```
  [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];
```

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 detail's 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 detail's 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 not, you must manually change the data stored for *all* records already stored in the container object. As normally, you should know, what columns you need, already 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 *Objective C objects.* But how does the container window know, what data shall be displayed in each 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.

`setFieldData: (ULONG field) withString: (char *) aString)` is used to set a string to field number `field`. Field numbering starts at `0`. Don't 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 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];
```

`free` must be implemented, if any dynamic allocation is used by the object. Otherwise the default method inherited from the superclass provides 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, the container window sends a message asking the program, if the new text (length is given in `cnrEditData->cbText` should be copied. 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 don't 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.

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 it's 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.

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. This library provides many classes simplifying handling of object storage (e.g. List classes, HashTable,...).

At the moment, it's 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 comples 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, it's 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.

List of Figures