# Programming OS/2 PM in Objective C
Version 0.5

Thomas Baier
baier@ci.tuwien.ac.at

August 26, 1994

# 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 runtime 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 manual describes an Objective C class library currently under development to simplify OS/2 PM programming. All you need to use it is the EMX port of GCC.

## 1.1 System Requirements

I assume, you have EMX/GCC (0.8h) installed on your system. To install the library you need a disk drive formatted using HPFS.

To simplify the task of designing the user interface for your programs, it's also recommended to use a dialog editor of some kind. The dialog editor must be capable of writing `.rc` files, which can be compiled with `rc.exe`, the Resource Compiler, or ready to use `.res` files (compiled resources, binary resources). You can find a dialog editor for example in the *IBM OS/2 developer's toolkit*. I also tried to use the *Guidelines* development tool to create `.rc` files, after manually patching the generated files, even this can be used.

## 1.2 Installation

To install the class library you have to unpack the compressed archive files `pm.zip`[1], `db.zip`, `header.zip` and `samples.zip`.

Change the current working directory to the root directory of the HPFS drive, where you want to install the library to and type

```
unzip pm
unzip db
unzip header
unzip samples
```

This automatically creates some directories:

- `\usr\include\objc` contains patched versions of `os2.h` and `os2emx.h`. This files had to be patched to work with Objective C.
- `\usr\include\pm` contains the include files used for this class library.
- `\usr\include\db` contains the include files for a simple database library used in one sample program.
- `\usr\lib` contains the class library for PM programming (`objcpm.a`) and the simple database library (`objcdb.a`).
- `\usr\samples` contains the source code for the sample programs.

[1] use Info ZIP 5.0 or newer

# Chapter 2

# Library License

This libraries are distributed as Shareware. To become a registered user fill in the registration form in the file `register.txt` and send it to me (the address can be found in `register.txt`).

After registration you are automatically registered for all following versions of the library until the major version number increases. That means by registering this version of the library together with the PM class library (PM library: version 0.5; DB library: version 0.3) you are automatically registered for all future versions of the PM and DB libraries including version 1.0.

Starting at version 1.1 of the PM or DB library you have to register newly at a special update price.

Support the Shareware distribution concept and register if you like this library and want to use it in your own applications. Future Shareware releases of this library depend heavily on the will of users to register. So, if no one registers this library, surely no further effort will be made in adding functionality to the libraries.

As a registered user you are allowed to write applications using these two libraries and distribute them at whatever price you think of.

Before registering you are allowed to test this library package as much as you like for a trial period of 30 days after first installing this package. You are not allowed to sell any of the applications written using this package if you have not registered it.

If you continue using the library package after the trial period of 30 days and don't register, that's an act of software-piracy. May your bad conscious haunt you till the end of your days ;-)

Think of the cheap pricing for this powerful library package and register. Future versions will include some tools to make life easier for programmers (just look at the NEXTSTEP development environment. Some kind of Project Builder or Interface Builder would look fine for OS/2 systems). But future Shareware-releases of this software heavily depend on the number of registrations made.

## 2.1 Distribution

This program is Shareware. Feel free to distribute the whole and unmodified package to anyone. You are not allowed to change any of the files part of the package before distributing, you only are allowed to distribute the package as a whole, including all files you received with it.

You are allowed to charge a small amount of money for the physical act of transferring the library. This amount of money must not exceed twice the cost of the storage medium. So, if you for example use floppy disks at a price of 10 ATS[1] each, you are allowed to charge at most 20 ATS for copying the disk. That makes a total of 30 ATS (including packaging).

If you don't like these distribution restrictions, don't distribute the program.

It's a shame to see some vendors "selling" Public Domain or Shareware programs at a price of 80 ATS per disk (3,5" HD disks are sold at a price between 5 and 10 ATS). Especially those vendors are not allowed to distribute the library package at their normal copying costs. So, change your pricing policy, or just don't distribute this library package.

If you're not sure, whether you are allowed to distribute the package, contact me. Any vendors who want to distribute registered versions of the library should do the same.

## 2.2 Warranty, bug reports, support

Well, as you might have thought, there's ABSOLUTELY NO WARRANTY for this library package.

If you find any bugs in the library or want me to make improvements, drop a short E-Mail message to me at `baier@ci.tuwien.ac.at`.

If you are a registered user of the application you will be notified via E-Mail (Internet) – if possible – when new versions of the library are released. If you have any questions concerning the use of the library, working around some special problems,...send me an E-Mail message, I'll try to do my best and answer your question.

[1] ATS is *Austrian Schillings*

# Part I

# Tutorial

# Chapter 3

# 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

## 3.1 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 */
}
```

The above example shows the necessary steps, a program has to go through to be run under 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 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. 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 part of code:

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

## 3.2 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;

  mainWindow = WinCreateStdWindow (HWND_DESKTOP,
                                   WS_VISIBLE,
                                   &createFlags,
                                   (PSZ) NEWCLASSNAME,
                                   (PSZ) "",
                                   0L,
                                   NULLHANDLE,
                                   1000,
                                   &clientWindow);
```

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

  WinDestroyWindow (mainWindow);

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

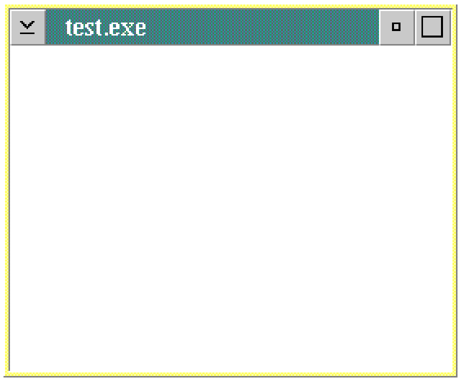


Figure 3.1: Sample application "test.exe"

The following source code illustrates how much simpler the PM class library is to use than "normal" OS/2 PM API functions.

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

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

  [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 `StdWindow`. The OS/2 window identifier is 1000, the window is created with a resizable border.

Calling the method `makeKeyAndOrderFront:` shows the window.

Figure 3.1 shows the window created by this simple piece of source code.

## 3.3 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>`.

After installation of the library, these include files can be found in the directories `\usr\include\pm` respectively `\usr\include\objc`.

If you encounter problems compiling any of the samples, check, if the file `\emx\include\objc\TypedStream.h` exists. 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 `\emx\include\objc\typedstr.h` to `\emx\include\objc\TypedStream.h`.

## 3.4 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 4.7 on page 28.

# Chapter 4

# 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**.

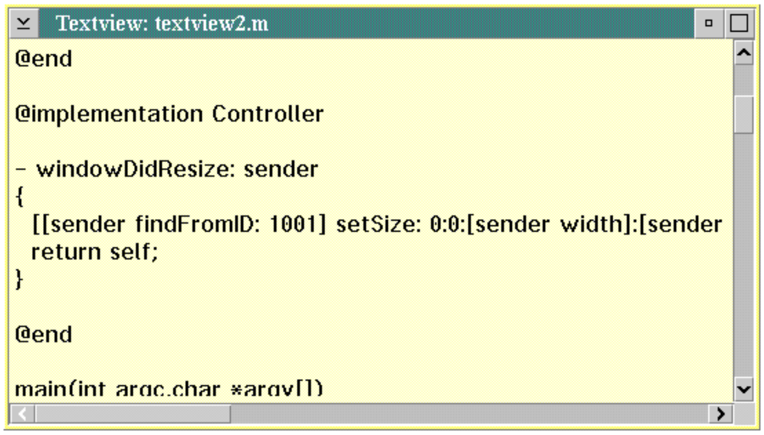


Figure 4.1: "Textview" application displaying it's own source code

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

## 4.1 Parts of the program

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

### 4.1.1 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:

```
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 it's contents are read to `contents`.

Following this part of 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 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`).

After initializing this, the main window is shown and the size of the MLE window is adapted to the size of the main window, to fill it's 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 code also sets the text displayed in the MLE window to be the buffer area `contents`.

## 4.2 Message loop

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

## 4.3 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 it's child windows, in our case, the MLE window.

## 4.4 Compilation

To compile this application, store the code shown in the following subsection to 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 -lobjc
emxbind -ep textview.exe
```

### 4.4.1 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]
          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 this program you will see, that the main window is resizable, but the MLE window inside the window remains the same size, whatever size it's parent window is.

The rest of this chapter shows how an object can be automatically notified, when the main window resizes, to adapt the size of the MLE window.

## 4.5 Delegate objects

One of the main advantages of Objective C compared to most other object-oriented programming languages is the possibility to check at runtime, if an object implements a specific method. This provides a simple way for objects to send messages to other objects, if these messages can be processed, to notify 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'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.

## 4.6 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.

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

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

`setSize::::` takes the coordinates of the lower left corner of the window (the first and second parameters) relative to it's 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 it's 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**[1], **self** is returned on successful completion of the method.

The following section shows the modified source code of **textview.m**, stored in **\usr\samples\textview** with the name **textview2.m**.

### 4.6.1 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

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

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

```
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 */
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 */
}
```

## 4.7 Sample makefiles

In the directory `\usr\samples\makefile` you can find a sample `makefile` together with the used make-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 (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 *~
```

# Chapter 5

# 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.

## 5.1 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
```


Figure 5.1: Simple menu for “Textview”

```
    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 5.1.

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, don't forget to specify the name of the binary resource file (just like any other object file).

When creating the main window of Textview, binary or `FCF_MENU` with the given flag `FCF_SIZEBORDER`. 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 creation. It must be the same as the identifier specified in the resource definition for the menu (in our case, it's `1000`).

## 5.2 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 it's parent window, which it runs modal for, is not possible untill the dialog window gets closed again (*dismissed*.

## 5.3 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.

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.

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.

## 5.4 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.

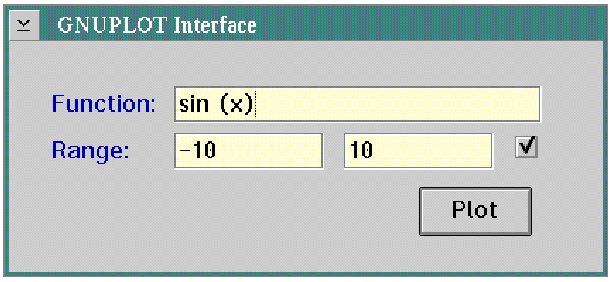


Figure 5.2: Simple PM interface to "Gnuplot"

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 5.2 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.

### 5.4.1 "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 it's 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`.

### 5.4.2 "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 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;
}
```

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

### 5.4.3 "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
```

### 5.4.4 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
    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 dialog `IDD_MAIN`. This template is normally written by a dialog editor.

```
#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` is also normally created by the used 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 4.7 at page 28 to realized, how to fill in these templates.

```
# 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 *~
```

# Part II

# Reference Manual

# Chapter 6

# Overview

This part describes all classes within the library, their instance variables and methods.

Figure 6.1 on page 43 shows all classes implemented in this library and their inheritence hierarchy.

At first an alphabetically listed overview of all classes with their instance variables and all supported methods. This was written in the style of an Objective C Interface declaration.

## 6.1 ActionWindow

```
@interface ActionWindow : Window
{
  List *commandBindings;
}

- init;
- free;
- bindCommand: (ULONG) command withObject: anObject
     selector: (SEL) aSel;
- findCommandBinding: (ULONG) command;
- (HRESULT) execCommand: (ULONG) command;

@end
```

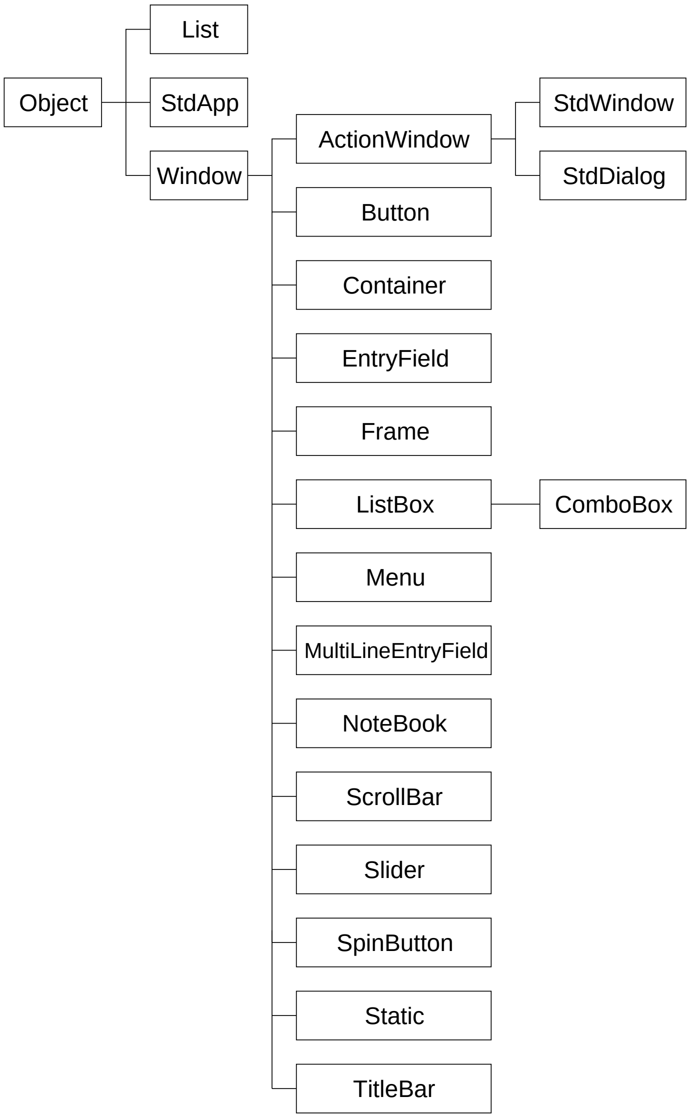


Figure 6.1: Inheritance hierarchy in Presentation Manager Class library

## 6.2 Button

```
@interface Button : Window
{
}

- initWithId: (USHORT) anId andFlags: (ULONG) flags
          in: (Window *) parent;
- clickdown;
- clickup;
- (USHORT) checked;
- (BOOL) highlighted;
- check;
- checkIndeterminate;
- uncheck;
```

## 6.3 ComboBox

```
@interface ComboBox : ListBox
{
}
```

## 6.4 Container

```
@interface Container : Window
{
}
@end
```

## 6.5 EntryField

```
@interface EntryField : Window <Selection>
{
}

- initWithId: (USHORT) anId andFlags: (ULONG) flags
          in: (Window *) parent;
- clearSelection;
- copySelection;
```

```
- cutSelection;
- pasteSelection;
- (BOOL) changed;
- (BOOL) readOnly;
- setReadOnly;
- setReadWrite;
- setTextLimit: (SHORT) limit;

@end
```

## 6.6 Frame

```
@interface Frame : Window
{
}

@end
```

## 6.7 List

```
@interface List : Object
{
  ULONG  key;
  void  *data;
  List  *next;
}

- init: (ULONG) aKey data: (void *) aData;
- free;
- insert: (List *) element;
- (int) compare: (List *) elem1 with: (List *) elem2;
- find: (ULONG) aKey;
- setKey: (ULONG) aKey;
- setData: (void *) aData;
- setNext: (List *) element;
- (ULONG)  key;
- (void *) data;
- next;

@end
```

## 6.8 ListBox

```
@interface ListBox : Window
{
}

- initWithId: (USHORT) anId andFlags: (ULONG) flags
          in: (Window *) parent;
- insertItem: (SHORT) pos text: (char *) buffer;
- (SHORT) count;
- (SHORT) selected;
- (SHORT) itemTextLength: (SHORT) pos;
- (char *) item: (SHORT) pos text: (char *) buffer;
- selectItem: (SHORT) pos;
- deleteItem: (SHORT) pos;
- deleteAll;
@end
```

## 6.9 Menu

```
@interface Menu : Window
{
}

@end
```

## 6.10 MultiLineEntryField

```
@interface MultiLineEntryField : Window
{
}
- initWithId: (USHORT) anId andFlags: (ULONG) flags
          in: (Window *) parent;
@end
```

## 6.11 NoteBook

```
@interface NoteBook : Window
{
```

```
}

@end
```

## 6.12 ScrollBar

```
@interface ScrollBar : Window
{
}

@end
```

## 6.13 Slider

```
@interface Slider : Window
{
}

@end
```

## 6.14 SpinButton

```
@interface SpinButton : Window
{
}

@end
```

## 6.15 Static

```
@interface Static : Window
{
}

@end
```

## 6.16 StdApp

```
@interface StdApp : Object
{
  HAB   hab;
  HMQ   hmq;
}

- init;
- free;
- run;
- (HAB) hab;

@end
```

## 6.17 StdDialog

```
@interface StdDialog : ActionWindow
{
  id       delegate;
  ULONG    result;
}

- initWithId: (ULONG) anId;
- loadMenu;
- free;
- delegate;
- setDelegate: aDelegate;
- (ULONG) result;
- makeKeyAndOrderFront: sender;
- runModalFor: sender;
- (MRESULT) handleMessage: (ULONG) msg
            withParams: (MPARAM) mp1 and: (MPARAM) mp2;

@end
```

## 6.18 StdWindow

```
@interface StdWindow : ActionWindow
{
  HWND        frame;
```

```
  id         delegate;
}

- initWithId: (ULONG) anId;
- initWithId: (ULONG) anId andFlags: (ULONG) flags;
- free;

- setSize: (USHORT) x : (USHORT) y : (USHORT) w : (USHORT) h;

- (HWND) frame;
- delegate;
- setDelegate: aDelegate;

- setTitle: (char *) aTitle;

- makeKeyAndOrderFront: sender;
- performClose: sender;

- (MRESULT) handleMessage: (ULONG) msg
            withParams: (MPARAM) mp1 and: (MPARAM) mp2;

@end
```

## 6.19 TitleBar

```
@interface TitleBar : Window
{
}

@end
```

## 6.20 ValueSet

```
@interface ValueSet : Window
{
}

@end
```

## 6.21 Window

```
@interface Window : Object
{
  HWND    window;
  Window *child;
  Window *sibling;
}

- init;
- associate: (HWND) hwnd;
- free;
- createObjects;
- insertChild: aChild;
- insertSibling: aSibling;
- findFromID: (USHORT) anId;
- findFromHWND: (HWND) aHwnd;
- (char *) text: (char *) buffer;
- (int) textLength;
- setText: (char *) buffer;
- setSize: (USHORT) x : (USHORT) y : (USHORT) w : (USHORT) h;
- size: (PSWP) aSize;
- (USHORT) width;
- (USHORT) height;
- (USHORT) xoffset;
- (USHORT) yoffset;
- (HWND) window;
- (USHORT) pmId;
- enable;
- disable;
- activate;
- deactivate;
- (MRESULT) handleMessage: (ULONG) msg
            withParams: (MPARAM) mp1 and: (MPARAM) mp2;

@end
```

# Chapter 7

# Classes

This chapter describes all variables and methods of the classes implemented in this library.

The description consists of three to five parts:

1. The name of the class and the precessing inheritance hierarchy
2. A short description of the class and it's proposed usage
3. A list of all instance variables and their use
4. All *newly* implemented class and instance methods and their description
5. Methods of a *delegate object* – if it exists – which get called at certain times

The list of instance variables is omitted if there are none of them defined but those inherited from the superclass.

If a class doesn't support delegate objects the corresponding section in the class description is omitted.

If no return type of some method is specified, the return type defaults to **`id`**, a generic pointer to an *Objective C object*.

Methods returning an **`id`** value normally return **`self`**, which is a pointer to the object itself on successful completion, **`nil`** otherwise.

## 7.1 ActionWindow

Inherits from: WINDOW : OBJECT

Class description:

`ActionWindow` is the common superclass for `StdWindow` and `StdDialog`. This class implements the ability to bind command messages to methods in other objects.

Everytime a command message occurs in a `StdWindow` or `StdDialog` the Event-Handler searches for a command binding and – if found – executes the corresponding *Action* in the *Target* object.

Instance Variables:

**List * commandBindings;**

This variable stores a list of all command bindings set up for a certain instance of `ActionWindow` or one of its successors.

Methods:

**- init;**

The instance method `init` initializes the instance variable `commandBindings` to `nil`.

**- free;**

`free` frees the memory allocated for the list of command bindings.

**- bindCommand: (ULONG) command withObject: anObject selector: (SEL) aSel;**

`bindCommand: withObject: selector:` sets up a new command binding. `command` is the command identifier, which normally is the identifier of the sender of the command (Pushbutton, Menuitem, ...). `anObject` is the *Target*, `aSel` the selector[1] of the *Action*.

An Action must be of the form `nameOfMethod: sender`. Only these methods can be called by `execCommand`. Actions should return `nil` on successful execution, a non-`nil` value otherwise.

**- findCommandBinding: (ULONG) command;**

[1] The selector of a method can be queried via `@selector (...)`

This method is used for checking, if a command binding for `command` has been set up previously. `findCommandBinding:` returns `nil`, if no command binding for `command` has been set up, a non-`nil` value otherwise.

**- (MRESULT) execCommand: (ULONG) command;**

`execCommand:` searches for the command binding for `command` and executes the corresponding *Action* in the set up *Target*, if one was found.

## 7.2 Button

Inherits from: WINDOW : OBJECT

Class description:

The Objective C class `Button` represents a special type of a `Window`. Instances of this class are normally associated with PM Windows of class `WC_BUTTON`. The instance methods can be used to set the state of a Button (to simulate a User Action to the Button) or to query the Button's state if it is a *Radiobutton*, a *Checkbox* or a *Tri-State Button*.

Setting and querying the text displayed in the Button can be done using `setText:` and `text:`.

Support for displaying icons instead of a text on a Button is currently not implemented when creating a Button Object "from Scratch", which means by not using a definition for this object in a OS/2 Resource File.

Methods:

**- initWithId: (USHORT) anId andFlags: (ULONG) flags in: (Window *) parent;**

Using this Initializer the Programmer can create a new Button in an existing parent window. `anId` is the PM id of the button to be created, `flags` specify the creation flags for the Button control (`BS_xxxx` and `WS_xxxx` constants). `parent` is the parent window of the newly created Button, which normally is either an instance of `StdDialog` or `StdWindow`.

After creation of the Button the size can be set via `setSize::::` and the text to be displayed via `setText:`.

Association to an existing PM Button Window should be done by using `associate:`.

A newly created Button Object is not automatically inserted as a child window of it's parent. Use `[parent insertChild: button]` where `parent` is the parent window and `button` is the newly created Button Object.

| Flag | Description |
|---|---|
| BS_PUSHBUTTON | The created Button will be a Pushbutton. |
| BS_CHECKBOX | The Button will be a Checkbox. |
| BS_AUTOCHECKBOX | The Button will be an AutoCheckbox, this one toggles it's state every time the user clicks on the Button. |
| BS_RADIOBUTTON | The Button will be a Radiobutton. In contrast to Checkboxes, a dot appears if the Button is checked. |
| BS_AUTORADIOBUTTON | In addition to a normal Radiobutton an AutoRadiobutton automatically unchecks all other Radiobuttons in the same group if it is checked. |
| BS_3STATE | A Tri-state Button has an additional check state, which is called *indeterminate*. |
| BS_AUTO3STATE | same as AutoCheckbox, but Tri-state Button. |
| BS_USERBUTTON | The button created will be an application-defined button. It has to be drawn by the application when a `BN_PAIN` message is received by the parent window. |

Table 7.1: Main Button styles used to define the type of Button


Figure 7.1: This figure shows (from left to right) the following Buttons types: *Pushbutton*, *Radiobutton*, *Checkbox* and *Tri-state Button*.

The following table list all possible `BS_xxxx` styles and a short description of these.

First the primary Button styles, which define the type of the Button. One of these must be given. All other style options in the following tables can be combined with one of the primary style via logical OR. Tables 7.1 (page 54), 7.2 (page 55), 7.3 (page 55) and 7.4 (page 55).

Figure 7.1 on page 54 shows the look of the main Button styles.

**- clickdown;**

By calling this method a click down with the left mouse button is simulated for this Button.

**- clickup;**

| Flag | Description |
|---|---|
| BS_NOCURSORSELECT | The Radiobutton is not selected when it is given the focus from keyboard actions. |

Table 7.2: Button styles which can be combined with an AutoRadiobutton

| Flag | Description |
|---|---|
| BS_HELP | Instead of posting a command message (`WM_COMMAND`), a help message is posted (`WM_HELP`). |
| BS_SYSCOMMAND | When this style is set, a `WM_SYSCOMMAND` message is posted instead of a command message (`WM_COMMAND`). |
| BS_NOBORDER | The Pushbutton doesn't have a drawn border. |

Table 7.3: Button styles which can be combined with a Pushbutton

`clickup` simulates – as a counterpart to `clickdown` – a release of the left mouse button when the mouse pointer is in the Button ("Click Up").

**- (USHORT) checked;**

`checked` queries the check state of the Button if it is a *Radiobutton*, a *Checkbox* or a *Tri-State Button*.

This method returns `0` if the Button is in unchecked state, `1` when in checked state and `2` when in indeterminate state.

**- (BOOL) highlighted;**

The result of `highlighted` is `TRUE` if the current state of the Button is highlighted, `FALSE` otherwise.

**- check;**

`check` sets the *checked* state of the Button.

**- checkIndeterminate;**

`checkIndeterminate` sets the *indeterminate* state of the Button.

| Flag | Description |
|---|---|
| BS_DEFAULT | Only one Button per window should have this style set. In dialogs this button is automatically pushed whenever the user presses the *Enter* key. |

Table 7.4: Button styles which can be combined with a Pushbutton or a Userbutton

**- uncheck;**

`uncheck` sets the *unchecked* state of the Button.

## 7.3 ComboBox

Inherits from: ListBox : Window : Object

Class description:

`CombobBox` is a class designed to provide an interface to OS/2 PM windows of class `WC_COMBOBOX`.

At the moment no additional functionality to it's superclass *ListBox* has been added. Special support for OS/2 PM Combobox windows will be added in the future.

A `ComboBox` consists of a `EntryField` and a `ListBox`. Access to the text in the `EntryField` is provided via `setText:` and `text:`. The items in the `ListBox` can be accessed by using the inherited methods of the superclass `ListBox`.

## 7.4 Container

Inherits from: Window : Object

Class description:

`Container` is a class designed to provide an interface to OS/2 PM windows of class `WC_CONTAINER`.

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Container windows will be added in the future.

## 7.5 EntryField

Inherits from: Window : Object

Class description:

The class `EntryField` was designed to simplify access to OS/2 PM Entryfield windows. Using the methods implemented for this class the programmer can control all interesting features of this predefined window class.

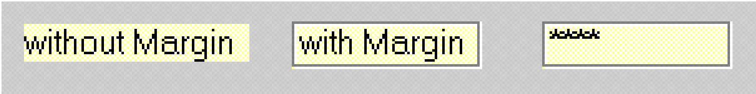


Figure 7.2: In this figure you can see (from left to right) an EntryField without a margin, one with a margin and an EntryField with margin and the style option BS_UNREADABLE

The text typed into the entryfield can be accessed via the inherited methods `setTest:` and `text:`. In future releases of this class library methods for automatically checking typed input will be provided for *integers*, *floating point numbers*,...

By adopting the procotol `Selection` simple access to Clipboard operations as *copy* or *paste* is provided. See also the description of this protocol on page 81.

## Methods:

**- initWithId: (USHORT) anId andFlags: (ULONG) flags in: (Window *) parent;**

By using this Initializer the Programmer can create a new Entryfield in an existing parent window. `anId` is the PM id of the Entryfield to be created, `flags` specify the creation flags for the Button control (`ES_xxxx` and `WS_xxxx` constants). `parent` is the parent window of the newly created Entryfield, which normally is either an instance of `StdDialog` or `StdWindow`.

After creating the Entryfield the size can be set via `setSize::::` and the text to be displayed via `setText:`. Clearing the text of an Entryfield can be achieved calling `[entryfield setText: ""]`.

Association to an existing PM Entryfield Window should be done by using `associate:`.

A newly created EntryField Object is not automatically inserted as a child window of it's parent. Use `[parent insertChild: entryfield]` where `parent` is the parent window and `entryfield` is the newly created EntryField Object.

Table 7.5 (page 58) shows most of the available `ES_xxxx` flags used at creation of the EntryField.

In addition to these flags there's also another group of flags defining the encoding scheme for the text in the EntryField. These flags are only used when a double-byte encoding scheme is used for text.

| Flag | Description |
|---|---|
| ES_LEFT | The text in the EntryField is left-justified. This style is used when neither ES_LEFT, nor ES_RIGHT nor ES_CENTER is specified. |
| ES_RIGHT | The text in the EntryField is right-justified. |
| ES_CENTER | The text in the EntryField is centered. |
| ES_AUTOSIZE | When this flag is set, the text will be sized to fit in the EntryField. |
| ES_AUTOSCROLL | The text in the EntryField is scrolled to the left or right if it is longer than would fit in the EntryField. |
| ES_MARGIN | A margin is drawn around the EntryField. |
| ES_READONLY | The EntryField will be created in *read-only* mode. |
| ES_UNREADABLE | Every character in the text is displayed as an asterisk. This is useful when querying passwords. |
| ES_COMMAND | This style classifies the EntryField as a command entry field. This style should be applied to at most one EntryField per Dialog or Window. |
| ES_AUTOTAB | When this flag is set, the focus is moved to the next Window when a character is appended to the text. |

Table 7.5: `ES_`xxxx styles used at creation of an EntryField

Figure 7.2 on page 57 shows three possible forms of how an EntryField can look.

**- (BOOL) changed;**

`changed` returns `TRUE` if the text displayed in the EntryField has changed since the last call to this method, `FALSE` otherwise.

**- (BOOL) readOnly;**

By using this method the programmer can query if the EntryField is in *read-only* or in *read-write* mode. When *read-only* no characters can be typed into the EntryField.

This method returns `TRUE` if the EntryField is in *read-only* mode, `FALSE` otherwise (*read-write*).

**- setReadOnly;**

Calling this method activates the *read-only* mode of the EntryField.

**- setReadWrite;**

`setReadWrite` switches the EntryField to *read-write* mode.

- **setTextLimit: (SHORT) limit;**

  By calling `setTextLimit:` the programmer can set the maximum number of characters which can be entered into the EntryField. `limit` is this maximum number of characters.

  When querying the contents of the EntryField via `text:` the maximum number of characters returned is `limit + 1`, including the concluding `'\0x0'` at the end of the string.

## 7.6 Frame

Inherits from: WINDOW : OBJECT

Class description:

`Frame` is a class designed to provide an interface to OS/2 PM windows of class `WC_FRAME` (Frame windows).

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Frame windows will be added in the future.

## 7.7 List

Inherits from: OBJECT

Class description:

Instances of this class are used to store command bindings and associated data. Don't use this class neither by instanciating nor by inheriting from it. This class will be replaced by a more generic list class in the future.

It is only used in the class `ActionWindow`.

## 7.8 ListBox

Inherits from: WINDOW : OBJECT

Class description:

`ListBox` is a class designed to be associated to the OS/2 PM class `WC_LISTBOX`. The class provides methods to give access to the items in the Listbox window.

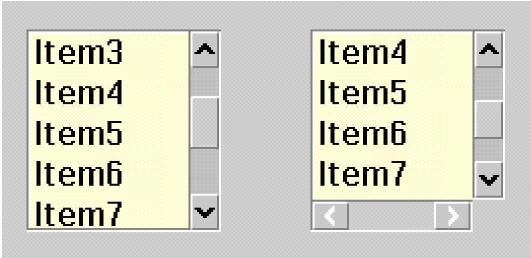


Figure 7.3: Here you can see a standard Listbox (left) and a Listbox window with an additional horizontal Scrollbar.

## Methods:

**- initWithId: (USHORT) anId andFlags: (ULONG) flags in: (Window *) parent;**

`initWithId: andFlags: in:` can be used to create a Listbox window at runtime. The parameters are the same as those used in the appropriate method of the class `Button`.

Figure 7.3 on page 60 shows two forms of Listbox windows. The left is a standard Listbox with only one Scrollbar – a vertical one. The right Listbox also has a horizontal Scrollbar.

How a Listbox window appears depends on what control flags you specify in the parameter `flags`. Table 7.6 shows which control flags are possible and what effect is caused by specififying them. One ore more of the flags can be specified. These flags must be binary or-ed using the `|` operator. If none of them should be used, `0L` should be given as `flags` parameter.

**- insertItem: (SHORT) pos text: (char *) buffer;**

Using this method you can insert a new item into the Listbox. `pos` is the position in the Listbox where the item shall be inserted. If `pos` is `LIT_END`, the item will be inserted as the last item in the Listbox.

`buffer` is the title of the item to be inserted. This string is shown afterwards in the Listbox at the specified position.

The first item in the Listbox is at position `0`, the last at `count - 1`.

**- (SHORT) count;**

`count` returns the number of items which are currently in the Listbox.

| Flag | Description |
|---|---|
| LS_HORZSCROLL | This flags adds a horizontal Scrollbar to the Listbox window, if it is specified at creation. |
| LS_MULTIPLESEL | Normally only one item in the Listbox can be selected once. If this flag is set, multiple selection is enabled. Currently querying the multiple selection is not supported by methods of this class. |
| LS_EXTENDEDSEL | Specifying this flag enables the extended selection user interface of the Listbox window. |
| LS_OWNERDRAW | This flag tells the Listbox not to draw the items itself. Appropriate messages are sent to the owner of the listbox, which has to draw them. |
| LS_NOADJUSTPOS | This flag tells the listbox not to adjust the size and position of the window. If this flag is set, maybe only part of the first or last item shown is drawn. |

Table 7.6: `LS_xxxx` styles used at creation of a Listbox window

**- (SHORT) selected;**

`selected` returns the position of the selected item. If no item is currently selected, a value below `0` is returned.

Multiple selection is currently not supported by this class. If you want to query multiple selection you have to use the appropriate OS/2 API functions, or just wait untill the next version of this library is released.

**- (SHORT) itemTextLength: (SHORT) pos;**

This method returns the length of the item text of the item at position `pos`. Only the number of characters in the item text is returned. Don't forget to allocate an extra character for the `NULL` at the end of the string before querying via `item: text:`.

**- (char *) item: (SHORT) pos text: (char *) buffer;**

`item: text:` copies the item text of the item at position `pos` in the Listbox into the array of characters pointed to by `buffer`. This method assumes, there is enough space in `buffer` to hold all of the item text, including the `NULL` at the end of the text.

This method returns `buffer`.

If `buffer` is `NULL`, a string is allocated via `malloc` to hold all of the item text. This string must be freed by the programmer later using `free`.

**- (SHORT) selectItem: (SHORT) pos;**

Calling this method the specified item at position `pos` will be selected. If `pos` is out of the range of the Listbox items, nothing happens.

**- (SHORT) deleteItem: (SHORT) pos;**

`deleteItem`: deletes the item at position `pos`. If `pos` is out of the range of the Listbox items, no item gets deleted.

Deletion of the currently selected item can be accomplished by sending this message:

```
[listbox deleteItem: [listbox selected]];
```

Here `listbox` is a pointer to the ListBox object.

**- (SHORT) deleteAll;**

`deleteAll` deletes all items in the Listbox.

## 7.9 Menu

Inherits from: WINDOW : OBJECT

Class description:

`Menu` is a class designed to provide an interface to OS/2 PM windows of class `WC_MENU`. Windows of these type are the *Actionbar* or simply whole menus.

The menu items not displayed are no windows on their own. They are created newly before they get displayed (when the menu they are in gets selected).

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Menus will be added in the future.

## 7.10 MultiLineEntryField

Inherits from: WINDOW : OBJECT

Class description:

`MultiLineEntryField` is a class designed to provide an interface to OS/2 PM windows of class `WC_MLE`.

At the moment the only additional functionality to it's superclass *Window* is the initializer `initWithId: andFlags: in:`. Special support for OS/2 PM MLE windows will be added in the future.

| Flag | Description |
|---|---|
| MLS_BORDER | This flag causes a border to be drawn around the MLE window |
| MLS_READONLY | Disable editing in the MLE window (*read-only* mode) |
| MLS_WORDWRAP | Enable word wrap |
| MLS_HSCROLL | Draw a horizontal scroll bar |
| MLS_VSCROLL | Draw a vertical scroll bar |
| MLS_IGNORETAB | If this flag is set, the MLE window ignores pressing the `TAB` key |
| MLS_DISABLEUNDO | Disable the *undo* function of the MLE window. |

Table 7.7: `MLE_xxxx` styles used at creation of a MLE window

The whole text in the MLE can be accessed via `setText:` and `text:`.

Methods:

**- initWithId: (USHORT) anId andFlags: (ULONG) flags in: (Window *) parent;**

Using `initWithId: andFlags: in:` you can create an instance of class `MultiLineEntryField` and an OS/2 PM *MLE window* from scratch. `anId` is the PM identifier of the window, `flags` are the flags specified at creation of the MLE. `parent` represents the parent window of the object, where the MLE shall be inserted.

Table 7.7 lists all possible style flags to be used for instances of this class.

## 7.11 NoteBook

Inherits from: Window : Object

Class description:

`NoteBook` is a class designed to provide an interface to OS/2 PM windows of class `WC_NOTEBOOK`.

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Notebook windows will be added in the future.

## 7.12 ScrollBar

Inherits from: WINDOW : OBJECT

Class description:

`ScrollBar` is a class designed to provide an interface to OS/2 PM windows of class `WC_SCROLLBAR`.

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Scrollbar windows will be added in the future.

## 7.13 Slider

Inherits from: WINDOW : OBJECT

Class description:

`Slider` is a class designed to provide an interface to OS/2 PM windows of class `WC_SLIDER`.

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Slider windows will be added in the future.

## 7.14 SpinButton

Inherits from: WINDOW : OBJECT

Class description:

`SpinButton` is a class designed to provide an interface to OS/2 PM windows of class `WC_SPINBUTTON`.

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Spinbutton windows will be added in the future.

## 7.15 Static

Inherits from: WINDOW : OBJECT

Class description:

`Static` is a class designed to provide an interface to OS/2 PM windows of class `WC_STATIC`. Windows of this class are used for *Labels* or simply *informational messages*.

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Static windows will be added in the future.

## 7.16 StdApp

### Inherits from: OBJECT

### Class description:

This class is used to initialize and free all necessary PM recources needed to run the application.

Every Application written using this library should use exactly one instance of this class.

### Instance Variables:

**HAB hab;**

This variable is used to store the *Handle Anchor Block* of the application. Read-only access to this instance variable is provided via `hab`.

**HMQ hmq;**

`hmq` stores the handle of the *Application Message Queue*. Through this message queue all application-relevant messages are passed to the designated receiver of these messages.

Because there is normally no need for the programmer to have direct access to this message queue, no methods for access to `hmq` are provided.

### Methods:

**- init;**

This is the standard initializer of this class. `init` creates the *Handle Anchor Block* and the *Application Message Queue*. The appropriate handles are stored in `hab` respectively `hmq`.

**- free;**

`free` destroys the *Application Message Queue* and the *Anchor Block*. After calling this method, the program is ready to exit.

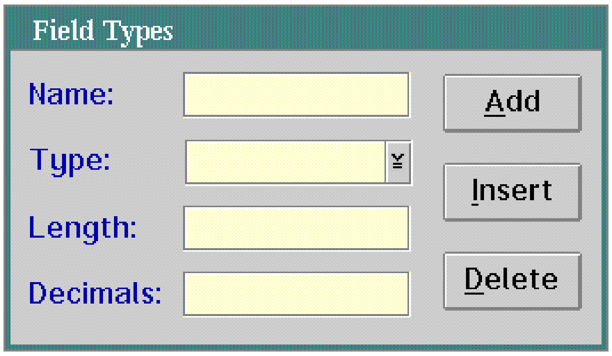


Figure 7.4: This figure shows a simple dialog window containing three *Buttons*, three *Entryfields* and a *drop-down Combobox*.

**- run;**

`run` fetches all messages and posts them to the appropriate receivers. This method exits when a `WM_QUIT` message is received.

**- (HAB) hab;**

`hab` returns the *Handle Anchor Block* of the application.

## 7.17 StdDialog

Inherits from: ActionWindow : Window : Object

Class description:

Instances of this class are used to represent *OS/2 Dialog windows*. At the moment dialogs are loaded from a resource file. This also initializes all controls (`Buttons`, `EntryFields`,...) in the dialog which are defined in the resource file.

Dialogs can be run *modal* for a given window, which means, while the dialog is active, no actions can be processed in the specified parent window, or *not modal*, where dialogs behave just like normal OS/2 PM main windows.

Figure 7.4 shows a simple dialog window.

Instance Variables:

**id delegate;**

`delegate` stores the handle of the *delegate object* of the dialog. Any events not processed by methods of this class are forwarded to the *delegate*.

See also *Methods implemented by the delegate*.

**ULONG result;**

After a dialog is dismissed (closed), the result of the dialog is stored in the instance variable `result`. This result can be queried by using the instance method `result`.

## Methods:

**- initWithId: (ULONG) anId;**

`initWithId`: loads a dialog resource from the main resource file, which is linked into the executable file. `anId` is a key value, which uniquely identifies the dialog to be loaded in the resource file.

This method returns `self` if successful, `nil` otherwise.

**- loadMenu;**

If the loaded dialog shall contain an *Application menu*, the menu must be explicitly loaded from the resource file by calling this method. The menu resource is assumed to have the same resource identifier as the dialog window itself.

`loadMenu` returns `self`.

**- free;**

`free` destroys the PM window and frees all resources allocated previously.

**- (ULONG) result;**

`result` returns the value stored in the instance variable `result`. `result` is set after the dialog gets dismissed.

Therefore calling this method should be done only *after* the dialog has been dismissed.

**- makeKeyAndOrderFront: sender;**

Calling `makeKeyAndOrderFront`: results in the dialog becoming the active window (*key window*), where all PM messages are sent to. It is also brought to the front, if hidden by other windows, or currently invisible.

**- runModalFor: sender;**

`runModalFor:` does the same as the previously described method `makeKeyAndOrderFront:`. In addition, the dialog is run modal for the window specified by `sender`. While the dialog is run, no message processing takes place in the sending window.

`runModalFor:` terminates, when the dialog gets dismissed.

When `sender` is `nil`, the dialog is not run modal for any window, but `runModalFor:` still doesn't terminate while the dialog is not dismissed. This can be used for applications consisting of only a single (or more) dialogs, but no `StdWindow`. In this case, don't call `[application run]`, but `[dialog runModalFor: nil]` (`application` is the current instance of a `StdApp`, `dialog` the dialog to be run instead of a `StdWindow`).

**- (MRESULT) handleMessage: (ULONG) msg withParams: (MPARAM) mp1 and: (MPARAM) mp2;**

`handleMessage: withParams: and:` gets called by the default dialog procedure.

This function evaluates the type of message received and reacts by calling a `delegate` method, if implemented (see "Functions implemented by the delegate").

If the received message is of type `COMMAND` or `SYS_COMMAND`, and a command binding for the command identifier has been set up, the corresponding *Action* in the set up *Target* gets called. (see class `ActionWindow`)

If the corresponding `delegate` function could not be found, the OS/2 default dialog procedure `WinDefDlgProc` is called.

### Methods implemented by the delegate:

**- windowDidMove: sender;**

After a window has been successfully moved, the delegate method `windowDidMove:` gets called.

**- windowDidResize: sender;**

`windowDidResize:` gets called after resizing a dialog. The newly achieved size of the window can be queried by sending the window (`sender`) appropriate messages (`width`, `height`).

**- windowDidResizeFrom: (USHORT) oldX : (USHORT) oldY to: (USHORT) newX : (USHORT) newY : sender;**

`windowDidResizeFrom:: to:::` is just the same as the previously described method `windowDidResize:`. In contrast to this method, `windowDidResizeFrom:: to:::` also sends the old (`oldX`, `oldY`) and new (`newX`, `newY`) width and height of the resized window.

These values can be directly used without querying the width and height of the window via `[sender width]` and `[sender height]`.

It can also be useful for some special purposes to know the width and height of the window before the process of resizing it. These parameters cannot be queried by using any of the methods of `sender`.

**- windowWillClose: sender;**

This function gets called if the `StdDialog` is about to close. If this function returns a non-`nil` value or the `delegate` object doesn't implement this method, the window will be closed.

If – otherwise – the `delegate` returns `nil`, closing the window is stopped and the normal execution of the program continues.

`sender` is a pointer to the sending instance of `StdDialog`.

**- buttonWasPressed: (ULONG) buttonId : sender;**

Everytime a `WM_COMMAND` message is received by `handleMessage: withParams: and:` from a Pushbutton, this message is sent to the delegate of the `StdDialog`.

`buttonId` is the OS/2 PM ID of the Button sending the `WM_COMMAND` message. `sender` is a pointer to the sending instance of `StdDialog`.

This method should return `nil` if the button event could be handled, a non-`nil` value otherwise.

**- menuWasSelected: (ULONG) menuId : sender;**

Analogous to `buttonWasPressed::` this delegate method is called whenever a menu item gets selected by the user.

`menuWasSelected::` should return `nil` if the menu selection could be processed successfully, a non-`nil` value otherwise.

**- commandPosted: (USHORT) origin : sender;**

Every time a command was posted and it could not be processed by `buttonWasPressed::` or `menuWasSelected::`, or if one of these methods or both are not implemented by the window delegate, or the command does not result from a button or a menu item, this delegate method is called.

`commandPosted::` should return `nil`, if the event could be processed successfully, a non-`nil` value otherwise.

**- sysButtonWasPressed: (ULONG) buttonID : sender;**

This method gets called, if a button posts a system command. It should react just alike `buttonWasPressed::`.

**- sysMenuWasSelected: (ULONG) menuId : sender;**

`sysMenuWasSelected::` is the counterpart to `menuWasSelected::`, but this method only gets called, whenever a system menu item was selected.

**- sysCommandPosted: (USHORT) origin : sender;**

`sysCommandPosted::` is called by the window's `handleMessage: withParams: and:` whenever a system command was posted, and neither `sysButtonWasPressed::` and `sysMenuWasSelected::` return `nil`.

It's behaviour should be analogous to `commandPosted::`.

**- (MRESULT) handleMessage: (ULONG) msg withParams: (MPARAM) mp1 and: mp2 : sender;**

Every time an event coult not be handle either by the window itself or by one of the delegate functions, `handleMessage: withParams: and:` gets called. So all types of events can be processed without the need to subclass `StdDialog`.

The return type should always be converted explicitly to type `MRESULT`.

See also the StdDialog build in method `handleMessage: withParams: and:`.

## 7.18 StdWindow

Inherits from: ActionWindow : Window : Object

Class description:

An instance of this class is a simple OS/2 PM Window, consisting of a *frame window* and a *client window*. It is possible to load resources like an *Icon*, a *Menu Bar* or an *Accelerator Table*.

Normally there's only one *StdWindow* in an application, showing and handling the application's Menu Bar and some default informations.

All messages of interest can be captured by an object called the delegate of the window. This object can then react to these messages. Normally there's no need to subclass this class.

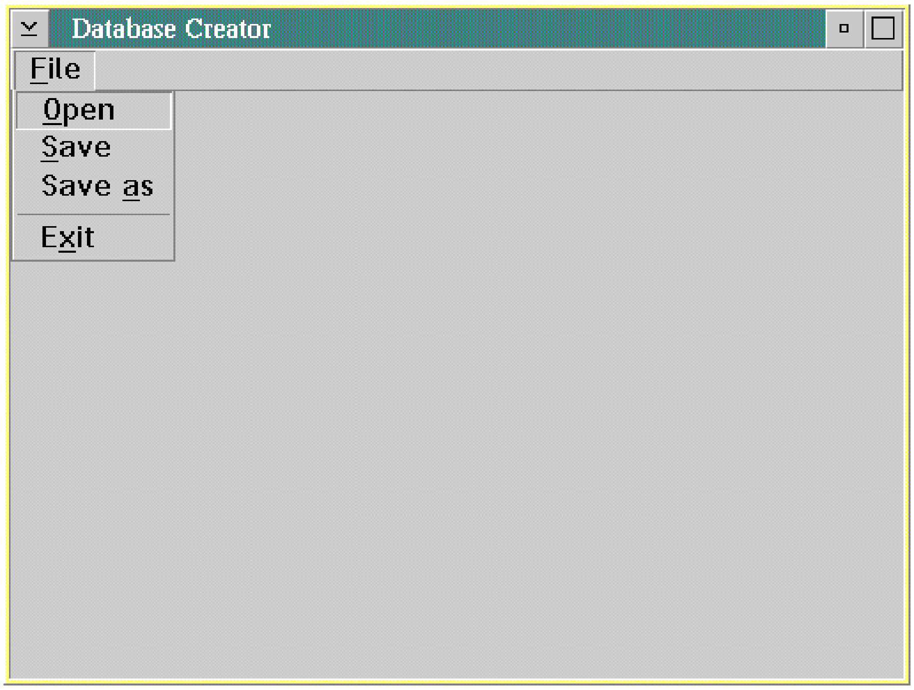


Figure 7.5: This figure shows an instance of the class StdWindow. At creation the flags FCF_MENU, FCF_SIZEBORDER and FCF_ACCELTABLE were specified

.

Figure 7.5 shows a `StdWindow` containing a menu bar.

## Instance Variables:

### HWND frame;

The instance variable `frame` is used to store the window handle of the *frame window*, where the inherited variable `window` is used to store the handle of the *client window*.

## Methods:

### - initWithId: (ULONG) anId;

This method is used to initialize an instance of the class `StdWindow`.

`anId` is the PM identification number of the window.

This method creates the frame window and the client window. The client window is an instance of the OS/2 PM-class `WINDOW_CLASS`. (Note the difference between Objective C classes and OS/2 PM-classes!)

The frame window handle is stored in `frame`, the client window handle in `window`.

The title of the window can be set via `setTitle:`.

### - initWithId: (ULONG) anId andFlags: (ULONG) flags;

This method is used to initialize an instance of the class `StdWindow`. In contrast to `init: id:` you can specify some *frame creation flags* to specify the resources to be loaded.

`flags` can be a combination of `FCF_MENU`, `FCF_ICON` and `FCF_ACCELTABLE`. `FCF_MENU` tells the object, that a Menu Bar should be loaded. The *resource id* of the Menu Bar must match the parameter `anId`. `FCF_ICON` is used to specify an Application Icon to be loaded and shown, whereas `FCF_ACCELTABLE` loads an Accelerator Table.

You should also specify the type of border to be drawn for the window. This can either be `FCF_SIZEBORDER` for a resizable border or `FCF_BORDER` for a normal border. A thin border can be created by specifying `FCF_THINBORDER`.

If you, for example, want to load a Menu Bar and an Icon you have to specify `FCF_MENU | FCF_ICON` as flags.

### - free;

`free` destroys the PM window and frees all resources allocated previously.

**- (HWND) frame;**

`frame` returns the OS/2 PM window handle of the frame window of the `StdWindow`.

**- delegate;**

This function returns a pointer to the current set delegate object of the window.

**- setDelegate: aDelegate;**

`setDelegate:` sets the object `aDelegate` as the delegate object of the window.

**- setTitle: (char *) aTitle;**

Using `setTitle:` you can set the title of the window. This title appears in the `TitleBar` of the window and also in the tasklist.

`aTitle` is the title to be set.

**- makeKeyAndOrderFront: sender;**

Calling `makeKeyAndOrderFront:` results in the `StdWindow` becoming the active window (*key window*), where all PM messages are sent to. It is also brought to the front, if hidden by other windows, or currently invisible.

**- performClose: sender;**

`performClose:` sends an OS/2 PM close message to the window (`WM_CLOSE`), which causes the window to be closed and – normally – the application to terminate.

**- handleMessage: (ULONG) msg withParams: (MPARAM) mp1 and: (MPARAM) mp2;**

`handleMessage: withParams: and:` gets called by the default window procedure for the OS/2 PM-class `WINDOW_CLASS`.

This function evaluates the type of message received and reacts by calling a `delegate` method, if implemented (see "Functions implemented by the delegate").

If the received message is of type `COMMAND` or `SYS_COMMAND`, and a command binding for the command identifier has been set up, the corresponding *Action* in the set up *Target* gets called. (see class `ActionWindow`)

If the corresponding `delegate` function could not be found, `handleMessage: withParams: and:` of its precessor in the class hierarchy is called.

Methods implemented by the delegate:

**- windowDidMove: sender;**

After a window has been successfully moved, the delegate method `windowDidMove:` gets called.

**- windowDidResize: sender;**

`windowDidResize:` gets called after resizing a window. The newly achieved size of the window can be queried by sending the window (`sender`) appropriate messages (`width`, `height`).

**- windowDidResizeFrom: (USHORT) oldX : (USHORT) oldY to: (USHORT) newX : (USHORT) newY : sender;**

`windowDidResizeFrom:: to:::` is just the same as the previously described method `windowDidResize:`. In contrast to this method, `windowDidResizeFrom:: to:::` also sends the old (`oldX`, `oldY`) and new (`newX`, `newY`) width and height of the resized window.

These values can be directly used without querying the width and height of the window via `[sender width]` and `[sender height]`.

It can also be useful for some special purposes to know the width and height of the window before the process of resizing it. These parameters cannot be queried by using any of the methods of `sender`.

**- windowWillClose: sender;**

This function gets called if the `StdWindow` is about to close. If this function returns a non-`nil` value or the `delegate` object doesn't implement this method, the window will be closed.

If – otherwise – the `delegate` returns `nil`, closing the window is stopped and the normal execution of the program continues.

`sender` is a pointer to the sending instance of `StdWindow`.

**- buttonWasPressed: (ULONG) buttonId : sender;**

Everytime a `WM_COMMAND` message is received by `handleMessage: withParams: and:` from a Pushbutton, this message is sent to the delegate of the `StdWindow`.

`buttonId` is the OS/2 PM ID of the Button sending the `WM_COMMAND` message. `sender` is a pointer to the sending instance of `StdWindow`.

This method should return `nil` if the button event could be handled, a non-`nil` value otherwise.

- **menuWasSelected: (ULONG) menuId : sender;**

  Analogous to `buttonWasPressed::` this delegate method is called whenever a menu item gets selected by the user.

  `menuWasSelected::` should return `nil` if the menu selection could be processed successfully, a non-`nil` value otherwise.

- **commandPosted: (USHORT) origin : sender;**

  Every time a command was posted and it could not be processed by `buttonWasPressed::` or `menuWasSelected::`, or if one of these methods or both are not implemented by the window delegate, or the command does not result from a button or a menu item, this delegate method is called.

  `commandPosted::` should return `nil`, if the event could be processed successfully, a non-`nil` value otherwise.

- **sysButtonWasPressed: (ULONG) buttonID : sender;**

  This method gets called, if a button posts a system command. It should react just alike `buttonWasPressed::`.

- **sysMenuWasSelected: (ULONG) menuId : sender;**

  `sysMenuWasSelected::` is the counterpart to `menuWasSelected::`, but this method only gets called, whenever a system menu item was selected.

- **sysCommandPosted: (USHORT) origin : sender;**

  `sysCommandPosted::` is called by the window's `handleMessage: withParams: and:` whenever a system command was posted, and neither `sysButtonWasPressed::` and `sysMenuWasSelected::` return `nil`.

  It's behaviour should be analogous to `commandPosted::`.

- **(MRESULT) handleMessage: (ULONG) msg withParams: (MPARAM) mp1 and: mp2 : sender;**

  Every time an event coult not be handle either by the window itself or by one of the delegate functions, `handleMessage: withParams: and:` gets called. So all types of events can be processed without the need to subclass `StdWindow`.

  The return type should always be converted explicitly to type `MRESULT`.

  See also the StdWindow build in method `handleMessage: withParams: and:`.

## 7.19 TitleBar

Inherits from: WINDOW : OBJECT

Class description:

`Container` is a class designed to provide an interface to OS/2 PM windows of class `WC_TITLEBAR`.

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Titlebar windows will be added in the future.

## 7.20 ValueSet

Inherits from: WINDOW : OBJECT

Class description:

`ValueSet` is a class designed to provide an interface to OS/2 PM windows of class `WC_VALUESET`.

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Valueset windows will be added in the future.

## 7.21 Window

Inherits from: OBJECT

Class description:

`Window` is an abstract superclass for all classes representing some kind of window (e.g. an Entryfield, a StdWindow or a Dialog).

This class should never be instantiated. It doesn't provide enough functionality to be really useful. It can be compared to the Objective C root class `Object`, it's the root class for all PM windows.

Only PM Windows with minimal functionality should be associated directly with instances of this class (e.g. Static Texts, Pushbuttons, ...).

Instance Variables:

**HWND window;**

`window` is an OS/2 PM window handle. It stores the handle of the PM window associated with an instance of this class.

**Window * child;**

This variable points to the first *child window* of this window.

**Window * sibling;**

`sibling` points to the first *sibling window* of this window.

## Methods:

**- init;**

This method initializes the instance variables to default values, which means it sets `window` to `NULLHANDLE`. `init` returns `self`.

**- associate: (HWND) hwnd;**

This instance method is used to associate an already existing Presentation Manager Window (Pushbutton, ...) with an instance of the class `Window`.

The only parameter `hwnd` is the window handle of the OS/2 PM window.

By using this method the programmer can create an Objective C Object without creating a PM window. After associating a PM window with a window Object, window data can be set and queried and manipulation can be done by using instance methods.

**- free;**

`free` frees all resources allocated by this object. `free` returns `self`.

`free` does *not* destroy an associated window using the OS/2 API function `WinDestroyWindow`.

If *child windows* or *sibling windows* exist, they are freed before this window.

**- createObjects;**

`createObjects` searches if any PM child windows of this window exist, and then creates appropriate Objective C objects for each of them and inserts them in the window hierarchy of this window as child windows.

This method is maily used after loading a `StdDialog` from a resource file to build the complete object hierarchy.

**- insertChild: aChild;**

`insertChild:` inserts `aChild` as a child into the window hierarchy of this window. `aChild` must be an instance of `Window` or one of its subclasses.

**- insertSibling: aSibling;**

`insertSibling:` inserts `aSibling` as a child into the window hierarchy of this window. `aSibling` must be an instance of `Window` or one of its subclasses.

**- findFromID: (USHORT) anId;**

`findFromID:` returns a pointer to an Objective C window identified by its OS/2 identifier `anId`, if there's a window identified by `anId` beyond the children of this window.

**- findFromHWND: (HWND) aHwnd;**

`findFromHWND:` returns a pointer to an Objective C window identified by its OS/2 window handle `aHwnd`, if there's a window identified by `aHwnd` beyond the children of this window.

**- (char *) text: (char *) buffer;**

By using `text:` the *Window Text* of the associated PM window can be queried. If `buffer` is `NULL`, enough memory to hold the window text is allocated via `malloc` and can be freed later by the application program using `free`.

The window text is copied into `buffer`, which must be large enough to hold all of the text, and buffer, or a pointer to the newly allocated area is returned.

The length of the window text can be queried via `textLength`.

**- (int) textLength;**

This method returns the number of characters the window text consists of. Don't forget to allocate an extra byte for the *End-of-String*-character before using `text:`.

**- setText: (char *) buffer;**

`setText:` is used to set the window text to a new string. This string is stored in `buffer`.

**- setSize: (USHORT) x : (USHORT) y : (USHORT) w : (USHORT) h;**

The instance method `setSize::::` is used for resizing a PM window by the application program. The parameters `x` and `y` represent the lower left corner of the window relative to its parent, `w` and `h` the width and the height of the window.

- **size: (PSWP) aSize;**

  `size` : fills the `SWP-structure aSize` with the appropriate values by querying this window's instance variables.

- **(USHORT) width;**

  `width` returns the width of the window in pixels.

- **(USHORT) height;**

  `height` returns the height of the window in pixels.

- **(USHORT) xoffset;**

  `xoffset` returns the horizontal offset of the lower left corner of the window from the lower left corner of the desktop in pixels.

- **(USHORT) yoffset;**

  `yoffset` returns the vertical offset of the lower left corner of the window from the lower left corner of the desktop in pixels.

- **(HWND) window;**

  This method returns the handle of the Presentation Manager window associated with this window object. If no PM window is associated with this object, `NULLHANDLE` is returned.

- **(USHORT) pmId;**

  `pmId` returns the OS/2 PM identification key of the window.

- **enable;**

  `enable` (re-) enables this window. Message processing for this window continues after receiving this message, if the window was previously in *disabled* state.

- **disable;**

  `disable` disables this window. No message processing is done by this window before *re-enabling* the window by using `enable`.

- **activate;**

  `activate` activates the window.

- **deactivate;**

`deactivate` deactivates the window.

**- (MRESULT) handleMessage: (ULONG) msg withParams: (MPARAM) mp1 and: (MPARAM) mp2;**

`handleMessage: withParams: and:` gets called by the *default Window procedure* for the OS/2 PM-class `WINDOW_CLASS` if a message was sent to this window. This function only reacts to `WM_ERASEBACKGROUND`. If this message is received, `TRUE` is returned, otherwise the result of the default window procedure (`WinDefWindowProc`).

The result should always be converted explicitly to the PM type `MRESULT`.

# Chapter 8

# Protocols

This chapter describes all available protocols. This descriptions consists of two parts,

1. The name of the protocol and a list of all classes which adopt it
2. A list of all methods declared and a short description of these

## 8.1 Selection

Adopted by: EntryField

Protocol description:

This protocol is used to declare all OS/2 Clipboard functions which can be used by the implemented Window classes.

**- clearSelection;**

`clearSelection` clears the current Selection of items in the object which adopts this protocol.

**- copySelection;**

Using `copySelection` the selected items are copied into the system clipboard. The items themselves remain unchanged.

**- cutSelection;**

`cutSelection` works alike a combination of `copySelection` and `clearSelection`. The selected items are copied into the system clipboard and they are deleted from the source window.

**- pasteSelection;**

When calling `pasteSelection` all selected Items in the system clipboard are pasted into the object implementing this method.

# 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.

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.

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

# Appendix B

# Future of this Library

In the near future I plan to extend most of the classes to what the PM API provides – and some tricky methods more. I just recently tried writing a completely new window class derived from `Window` and it seems to work fine.

It's also necessary to write better documentation for the classes themselves, at most the tutorial is quite short at this time.

I'm currently working on some kind of a *Project-Builder* like development environment, as known from NEXTSTEP. That will a point-and-click environment to quickly assemble applications, generate makefiles, compile and link and so on.

Next I'd like to provide a simple program for creating Objective C classes in a graphical way and then writing the appropriate interface and also skeletons of the implementation files.

If you have any good ideas, what should be included to build a really usable development environment, drop me an E-Mail message at `baier@ci.tuwien.ac.at`.

If anyone is working on a dialog editor, just let me know. Using Objective C as the main programming language makes it possible to create the command bindings using point-and-click actions and then store them in a Objective C typed stream.

# Appendix C

# List of Tables

# Appendix D

# List of Figures

# Index