OS/2 PM and database library

Version 0.6

Reference Manual

February 1995

Thomas Baier
baier@ci.tuwien.ac.at

Abstract

This manual is a reference manual for the *Objective C class library* for OS/2 PM and database programming.

Here all necessary information concerning the classes provided by the library can be found.

If you are searching for specific information concerning

- *Installation* ··· Read the Installation Manual.
- *Basics of Application development* ··· Read the appropriate sections in the Tutorial. There you can find a gentle introduction into using this library package for developing OS/2 PM applications.
- *Classes and Methods provided by the library* ··· You can find special information about the provided classes and methods in this manual, the Reference Manual.
- *The Database Builder Utility* ··· Read the appropriate sections in the Application Programming Tools Manual.
- *Literature* ··· Look in the Literature section of this Manual.

Contents

Part I

PM programming classes

Chapter 1

Overview

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

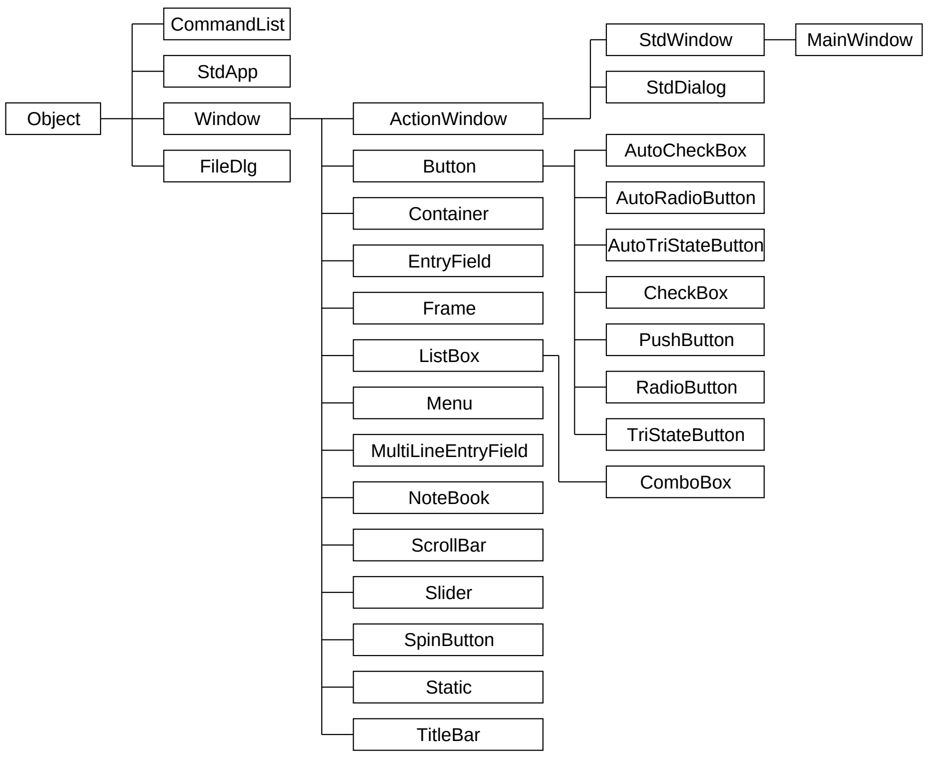

Figure 1.1: Inheritance hierarchy in Presentation Manager Class library

Figure 1.1 on page 11 shows all classes implemented in this library and their inheritence hierarchy.

In the beginning you will be shown 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.

General information about PM programming can be found in [2] and [1]

1.5 Button

```
@interface Button : Window
{
}

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

@end
```

1.6 CheckBox

```
@interface CheckBox : Button
{
}

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

@end
```

1.7 ComboBox

```
@interface ComboBox : ListBox
{
}

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

@end
```

1.8 CommandList

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

```
- (CONTAINER_MINIREC *) nextSelected;
- (BOOL) recordIsSelected;

- invalidateRecord;
- invalidateSelectedRecords;

- hideRecord : sender;
- hideSelectedRecords : sender;
- hideNotSelectedRecords : sender;
- showRecord : sender;
- showAllRecords : sender;
- (BOOL) recordIsHidden;

- (ULONG) columns;

- (FIELDINFO *) firstColumn;
- (FIELDINFO *) lastColumn;
- (FIELDINFO *) nextColumn;
- (FIELDINFO *) previousColumn;

- (char *) columnTitle;
- (ULONG) columnTitleAttributes;
- (ULONG) columnDataAttributes;

- hideColumn : sender;
- showColumn : sender;
- showAllColumns : sender;
- (BOOL) columnIsHidden;

- invalidateColumns;
- setColumnTitleAttributes: (ULONG) attr;
- setColumnDataAttributes: (ULONG) attr;

- select;
- deselect;
- selectAll: sender;
- deselectAll: sender;

- sort: (ULONG) column;

@end
```

1.10 EntryField

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

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

- clearSelection;
- initWithId: (ULONG) 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
```

1.14 MainWindow

```
@interface MainWindow : StdWindow
{
}

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

@end
```

1.15 Menu

```
@interface Menu : Window
{
}

- enableItem: (USHORT) identifier;
- disableItem: (USHORT) identifier;

@end
```

1.16 MultiLineEntryField

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

1.21 Slider

```
@interface Slider : Window
{
}

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

@end
```

1.22 SpinButton

```
@interface SpinButton : Window
{
}

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

@end
```

1.23 Static

```
@interface Static : Window
{
}

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

@end
```

1.24 StdApp

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

- init;
- free;
- run;

- (HAB) hab;

@end
```

```
- makeKeyAndOrderFront: sender;
- performClose: sender;

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

@end
```

1.27 TitleBar

```
@interface TitleBar : Window
{
}

@end
```

1.28 TriStateButton

```
@interface TriStateButton : Button
{
}

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

@end
```

1.29 ValueSet

```
@interface ValueSet : Window
{
}

@end
```

1.30 Window

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

- init;
- associate: (HWND) hwnd;
- free;
```

Chapter 2

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.

All methods having just one parameter called `sender` can be used as targets for command/action bindings. If not specified explicitly, `sender` is ignored. Normally, this parameter should be a pointer to the sending object which can be obtained by `self`.

2.1 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.

2.3 AutoRadioButton

Inherits from: BUTTON : WINDOW : OBJECT

Class description:

The class `AutoRadioButton` is a subclass of `Button`. It's only purpose is to simplify creating a PM Button window for a special purpose.

For a short description of an instance of this class see table 2.1 on the following page. Figure 2.1 on page 27 shows an instance of this class. See the description of the class `Button` for access methods.

Methods:

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

This method initializes a newly created instance of `AutoRadioButton`. Using this class and method is similar to creating a Button object while specifying the flag `BS_AUTORADIOBUTTON`.

2.4 AutoTriStateButton

Inherits from: BUTTON : WINDOW : OBJECT

Class description:

The class `AutoTriStateButton` is a subclass of `Button`. It's only purpose is to simplify creating a PM Button window for a special purpose.

For a short description of an instance of this class see table 2.1 on the following page. Figure 2.1 on page 27 shows an instance of this class. See the description of the class `Button` for access methods.

Methods:

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

This method initializes a newly created instance of `AutoTriStateButton`. Using this class and method is similar to creating a Button object while specifying the flag `BS_AUTO3STATE`.

2.5 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.

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 2.3: Button styles which can be combined with a Pushbutton

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

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 2.1 (page 26), 2.2 (page 26), 2.3 (page 26) and 2.4 (page 28).

Figure 2.1 on page 27 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;

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

- (ULONG) 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.

- uncheck;

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

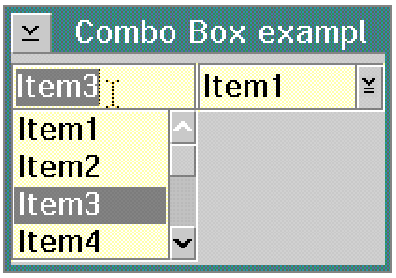

Figure 2.2: Combo Box controls (simple and dropdown)

As in all other `initWithId: andFlags: in:` methods, `parent` is a pointer to the parent window, which must be an instance of `Window` or one of it's subclasses.

Figure 2.2 on the next page shows two combo boxes. The left one was created using `CBS_SIMPLE`, the right one using the flag `CBS_DROPDOWN`. Using `CBS_DROPDOWNLIST` would have the same apperance as the right combo box.

2.8 CommandList

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`.

2.9 Container

Inherits from: WINDOW : OBJECT

Class description:

The OS/2 container class is one of the most flexible user interface classes provided by the OS/2 API. Because of its flexibility, there's a lot of work to do for the application programmer, before he can use a container object in his programs successfully. The Objective C class `Container` was created to simplify the handling of the API functions concerning this PM class.

Information concerning this specific PM class and how to use it in standard C programs can be found in [3]. This series of articles is part of EDM/2 and can be obtained via anonymous ftp from *hobbes.nmsu.edu* and many other sites which provide OS/2 software.

Instances of this class are designed to store Objective C objects as data. So anything you can encapsulate in an Objective C class definition can be stored in container objects.

Don't forget to specify `CCS_MINIRECORDCORE` at initialization. Only this style is supported at the moment.

When using detail's view, the objects inserted must conform to the protocol `ContainerItem`.

Instance Variables:

- addColumn: (char *) aTitle;

When in detail's view, the container displays data in multiple columns. This method can be used to add a new column at run-time. `aTitle` is the title text of the column which is shown if this feature is enabled.

- insertObject: anObject;

`insertObject:` inserts a new object into the container. `anObject` must be an allocated and initialized Objective C object. The title text of the newly inserted object is empty, the application uses the clock pointer (`SPTR_WAIT`) as the icon to be displayed.

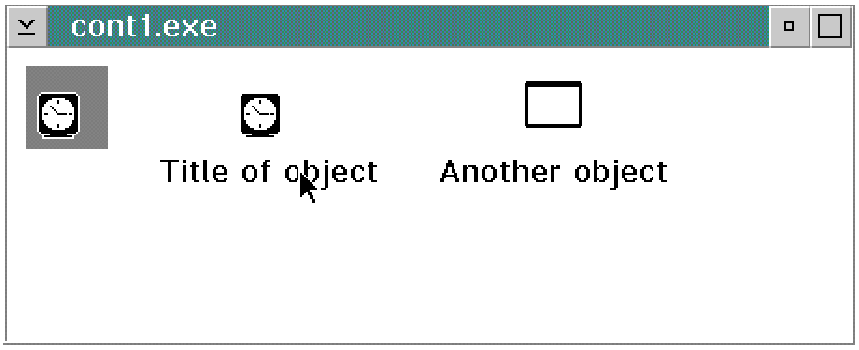

Figure 2.3: Container window in icon view showing objects created with the different insertObject: methods

This method is preferred for inserting new objects into a container when in detail's view, where it doesn't matter, which icon or title is displayed.

Figure 2.3 on the facing page shows a cotainer window in icon view. The leftmost object was inserted using `insertObject:`.

When in one of the other container views, consider using `insertObject: withTitle:` or `insertObject: withTitle: andIcon:`.

- insertObject: anObject withTitle: (const char *) aTitle;

Just as with `insertObject:` a new object is inserted into the container. `aTitle` specifies the title text to be displayed. This method should be used if the container is in text view. The clock pointer is used as icon.

Figure 2.3 on the page before shows a cotainer window in icon view. The middle object was inserted using `insertObject: withTitle:`.

- insertObject: anObject withTitle: (const char *) aTitle andIcon: (ULONG) anIcon;

Using `insertObject: withTitle: andIcon:` the application programmer has full control over the data displayed in any of the views. `anObject` is a pointer to the Objective C object, which shall be inserted into the container.

The title text of the item is specified via `aTitle`, a resource handle of the icon (as can be queried using `WinQuerySysPointer ()`) is specified in `anIcon`.

Use this method for inserting new objects when in icon view or tree view, of if you plan to change the view of the container at run-time to one of these.

Figure 2.3 on the preceding page shows a cotainer window in icon view. The object to the right was inserted using `insertObject: withTitle: andIcon:`.

- arrange;

Figure 2.4 on the next page shows a container window in detail's view. It holds three records, each of them consisting of two columns. The second record is selected. Note, that the second column of the second record holds a multi-line entry.

- (ULONG) records;

`records` queries the number of items currently stored in the container. The number of items (*records*) is returned.

- object;

This method returns a pointer to the object of the current record. The current record is set using the methods `firstRecord`, `lastRecord`, `nextRecord`, `previousRecord`, `firstSelected` and `nextSelected`.

- (CONTAINER_MINIREC *) firstRecord;

This method retrieves the first record in the container. A pointer to the data struction `CONTAINER_MINIREC`, which is used to store the data internally, is returned. The data of the record can be accessed using `object`. It is stored in `recordBuffer`.

If you want to visit every record, consider using the following piece of code:

```
.
.
  if ([container firstRecord]) {
    do {
      /* specific manipulations  */
      /* for each record go here */
    } while ([container nextRecord]);
  }
.
.
```

Here, the container object is assumed to be stored in a variable called `container`. The specific code for manipulating or querying each record is put in the `do` loop.

`firstRecord` returns a pointer to the `CONTAINER_MINIREC` structure of the first record, if none exists, `NULL` is returned.

- (CONTAINER_MINIREC *) lastRecord;

In contrast to `firstRecord`, this method searches for the last record stored in the container. If none is found, `NULL` is returned, otherwise a pointer to the `CONTAINER_MINIREC` structure of the last record.

Visiting all records, starting at the end can be accomplished using this piece of code.

```
.
.
  if ([container lastRecord]) {
    do {
      /* specific manipulations  */
      /* for each record go here */
    } while ([container previousRecord]);
  }
.
.
```

The variables have the same meaning as shown before in `firstRecord`.

Calling `hideRecord:` causes the current record to be hidden. The record is not deleted, it only will not be displayed by the container.

This method automatically calls `invalidateRecord`.

If you want to hide more than one record, consider using `hideSelectedRecords:` or `hideNotSelectedRecords:` for performance issues. Hiding each record using `hideRecord:` is painfully slow.

- hideSelectedRecords: sender;

This method hides all selected records and thereafter causing them to be invalidated.

If you want to hide some records which are currently not selected, walk through the list of records using `firstRecord` and `nextRecord` and select or deselect some of the records using `select` or `deselect`. Afterwards call `hideSelectedRecords:`.

Because this methods accepts a parameter `sender` it can be used as target of a command/action binding. The parameter `sender` itself is ignored.

- hideNotSelectedRecords: sender;

In contrast to `hideSelectedRecords:` this method hides all records which are *not selected.*

`sender` is ignored.

- showRecord: sender;

This method is the counterpart to `hideRecord:`. It shows the current record, if it was previously hidden.

- showAllRecords: sender;

`showAllRecords:` changes the state of all hidden records to visible again and displays them.

- (BOOL) recordIsHidden;

If the current record is hidden, this method returns `YES`, otherwise `NO` is returned.

- (ULONG) columns;

`columns` is used to query the total number of columns currently existing in detail's view.

- (FIELDINFO *) firstColumn;

As `firstRecord` sets the current record to the first record, this method affects the internal buffer variable `columnBuffer`. The first column is put into this variable and `columnBuffer` is returned.

`columnBuffer` stores a pointer to the `FIELDINFO` structure of the current column.

If there are no columns, `NULL` is returned.

- (FIELDINFO *) lastColumn;

In contrast to `firstColumn`, `lastColumn` queries information about the last column stored in the container. If no columns exist, `NULL` is returned.

- (FIELDINFO *) nextColumn;

This method queries information about the next column. The search must have been initialized using `firstColumn`.

This part of code can be used to query and modify information for all existing columns, starting at the first one:

Flag	**Description**
CFA_BITMAPORICON	The data stored is a bitmap or icon.
CFA_STRING	The data stored is a text string. This is the only datatype currently supported. Don't specify one of the other flags!
CFA_ULONG	Data is an unsigned long integer.
CFA_DATE	Data is a date structure.
CFA_TIME	Data is a time structure.

Table 2.10: Flags specifying the data type of a field in a container

Flag	**Description**
CFA_SEPARATOR	Draw a vertical separator line to the right of the current column.
CFA_HORZSEPARATOR	Draw a horizontal separator line underneath the column title.
CFA_OWNER	Enable ownerdraw for this field. This is currently not supported by this class.
CFA_INVISIBLE	If this flag is set, the column is not visible.
CFA_FIREADONLY	This sets the data displayed to read-only mode.

Table 2.11: Miscellaneous flags specified for container column data.

`columnTitle` returns a pointer to the `NULL`-terminated title string of the current column.

- (ULONG) columnTitleAttributes;

Every column stored in the container has two different attribute settings, one for the title data, and one for the data itself.

`columnTitleAttributes` is used to query the title attribute settings of the current column. These settings can be modified using `setColumnTitleAttributes`.

Table 2.8 on the preceding page shows the attributes which can be set or queried for the title data. Additionally, the appearance can be modified using the alignment flags shown in table 2.9 on the page before.

- (ULONG) columnDataAttributes;

Every column stored in the container has two different attribute settings, one for the title data, and one for the data itself.

`columnDataAttributes` is used to query the data attribute settings of the current column. These settings can be modified using `setColumnDataAttributes`.

Each column can display data of a specific data type. The data type is specified using the flags shown in table 2.10.

Table 2.9 on the page before shows all possible flags used for alignment of the data, table 2.11 shows all other possible flags which can be specified for column data.

- hideColumn: sender;

Using `hideColumn:` causes the current column to be hidden in the future. The data inside this column is preserved and can be restored using `showColumn:`

- showColumn: sender;

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 2.12: `ES_xxxx` styles used at creation of an EntryField

Calling this method causes the records to be sorted by column number `column`. Do not use this method if your records are not prepared to hold different columns as needed in detail's view.

The sorting is simply done in ascending order by the ASCII values of the data represented as strings.

2.10 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.

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 71.

Methods:

- initWithId: (ULONG) 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`.

To enable the users of your programs to perform simple file selection as customary in OS/2 PM applications, the class `FileDlg` is provided. This dialog lets the user select a single file on one of the disks, supporting this action with various features shown in figure 2.6 on the facing page.

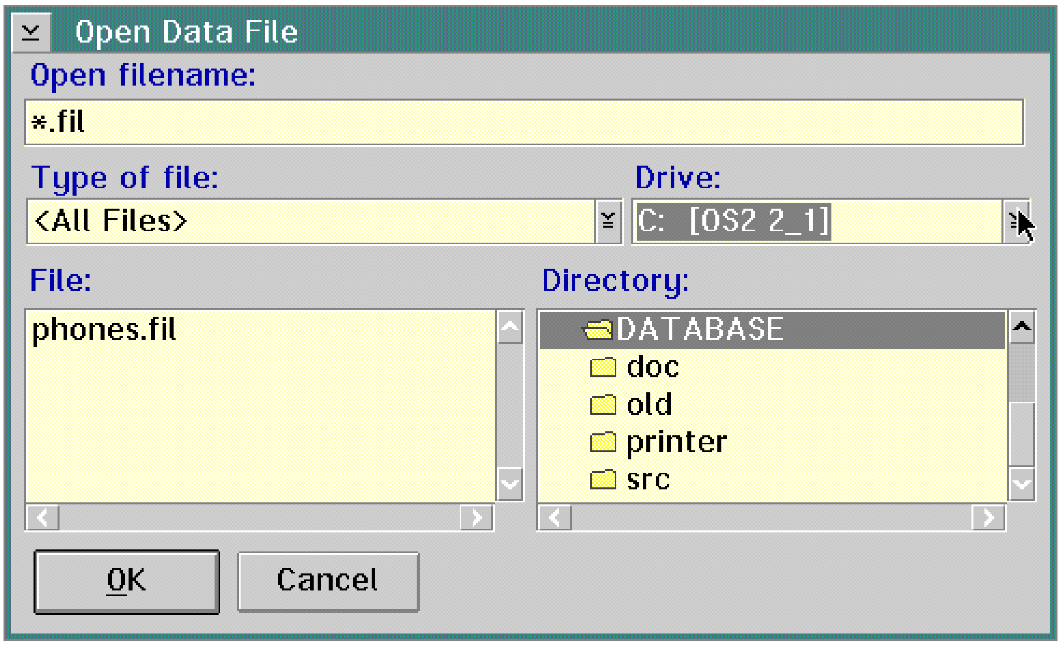

Figure 2.6: Sample file dialog

A file dialog can be created either for opening a file (see `initForOpen: withFilter:`) or for saving a new file (see `initForSaveAs: withFilter:`).

The dialog provides user interface objects to select the *disk drive*, the *directory* and the *filename* of the file to be opened or saved.

Depending on the initializing method either a `Open` or a `Save` button is present to close the dialog.

Instance Variables:

FILEDLG fileDlg;

This variable is used to store the structure used to create and display a file dialog. The result of a file dialog after closing is also stored here. There is normally no use to access this structure directly.

Methods:

- init;

This method initializes the structure `FILEDLG` to zeros and calls the `init` method of it's superclass.

- initForOpen: (const char *) aTitle withFilter: (char *) aFilter;

This method initializes a dialog for opening a file. The dialog is always centered in it's parent window, which is specified as a parameter to `runModalFor:`.

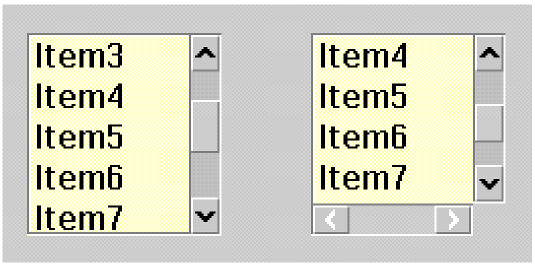

Figure 2.7: Here you can see a standard Listbox (left) and a Listbox window with an additional horizontal Scrollbar.

Every dialog *must* either contain `FDS_OPEN_DIALOG` or `FDS_SAVEAS_DIALOG`. Otherwise an error code is returned when calling `runModalFor:`.

Other flags, which can be specified are shown in table 2.13 on the preceding page.

- setOKTitle: (const char *) aTitle;

Using `setOKTitle:` the application can set the title string of the *OK*-Button displayed in the file dialog. Normally the title string is either `Open` or `Save`.

- (ULONG) runModalFor: sender;

`runModalFor:` displays the file dialog and runs it as a modal dialog box in respect to `sender`, which must be an instance of `Window` or one of its subclasses.

On successful completion this method returns `DID_OK`, otherwise `DID_CANCEL` is returned.

- (char *) fileName;

After successful execution of the file dialog, this method returns a pointer to the filename which was selected.

This method only returns a valid string after successful execution using `runModalFor:`.

2.12 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.

2.13 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.

- (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.

2.14 MainWindow

Inherits from: StdWindow : ActionWindow : Window : Object

Class description:

This class is only provided for simplification purposes. It extends it's superclass `StdWindow` only by means of specifying default flags when using the methods `initWithId:` and `initWithId: andFlags:`.

By default, the flags

- FCF_MINMAX
- FCF_SHELLPOSITION
- FCF_SYSMENU
- FCF_TASKLIST
- FCF_TITLEBAR

are specified at creation.

Methods:

`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.

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

Methods:

- initWithId: (ULONG) 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 2.15 lists all possible style flags to be used for instances of this class.

2.17 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.

2.18 PushButton

Inherits from: BUTTON : WINDOW : OBJECT

Class description:

The class `PushButton` is a subclass of `Button`. It's only purpose is to simplify creating a PM Button window for a special purpose.

For a short description of an instance of this class see table 2.1 on page 26. Figure 2.1 on page 27 shows an instance of this class. See the description of the class `Button` for access methods.

Methods:

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

This method initializes a newly created instance of `PushButton`. Using this class and method is similar to creating a Button object while specifying the flag `BS_PUSHBUTTON`.

2.19 RadioButton

Inherits from: BUTTON : WINDOW : OBJECT

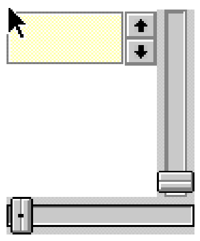

Figure 2.9: This figure shows a horizontal and a vertical slider and a spinbutton

This method is used to initialize a newly created instance of `ScrollBar`.

The PM identifier of the window is specified with `anId`, the parent window, in which the scrollbar shall be displayed is set via `parent`.

`flags` is used to specify, what kind of scrollbar shall be created. See table 2.16 for more information on the flags.

Figure 2.8 on the next page shows examples of a *horizontal scrollbar* (left) and a *vertical scrollbar* (right).

- (SHORT) position;

This method returns the current scrollbar position. This position is always in the range of [lowerBound;upperBound].

- (SHORT) lowerBound;

Return the lower bound of the scrollbar range.

- (SHORT) upperBound;

Return the upper bound of the scrollbar range.

- setPosition: (SHORT) position;

`setPosition:` sets the current position of the slider in respect to the upper and lower bounds.

- setScrollBar: (SHORT) position withBounds: (SHORT) lower : (SHORT) upper;

This method sets the slider position and the upper and lower bounds of the scrollbar. The slider position must be in the range of [lower;upper].

- setThumbSizeForVisible: (SHORT) visible of: (SHORT) all;

Using `setThumbSizeForVisible: of:` the size of the slider is adjusted to match `visible` visible items of a total of `all` items.

2.21 Slider

Inherits from: Window : Object

Class description:

A Slider object is used to let the user visually choose a discrete value in a specified range.

Currently, only creation of a Slider object is supported.

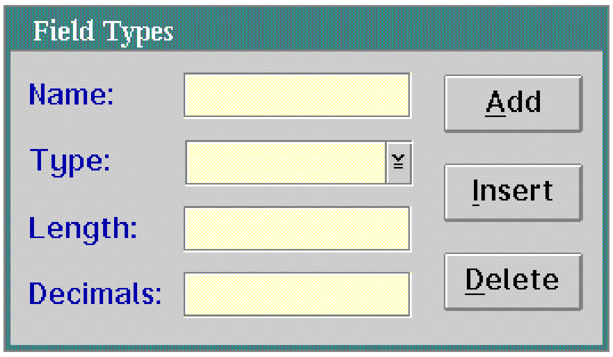

Figure 2.10: This figure shows a simple dialog window containing three *Buttons*, three *Entryfields* and a *drop-down Combobox*.

2.24 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.

- run;

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

- free;

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

- delegate;

This method returns the *delegate* object of this dialog.

- setDelegate: aDelegate;

Using `setDelegate:`, the object specified with `aDelegate` is set the delegate object of this dialog. See the appropriate section of this class description for the methods which can be processed by the delegate object.

- (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`).

- dismiss: sender;

Calling `dismiss:` causes the dialog – either running modal or not modal – to be dismissed. The instance variable `running` is reset to `NO` and the dialog is hidden from display.

- (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:

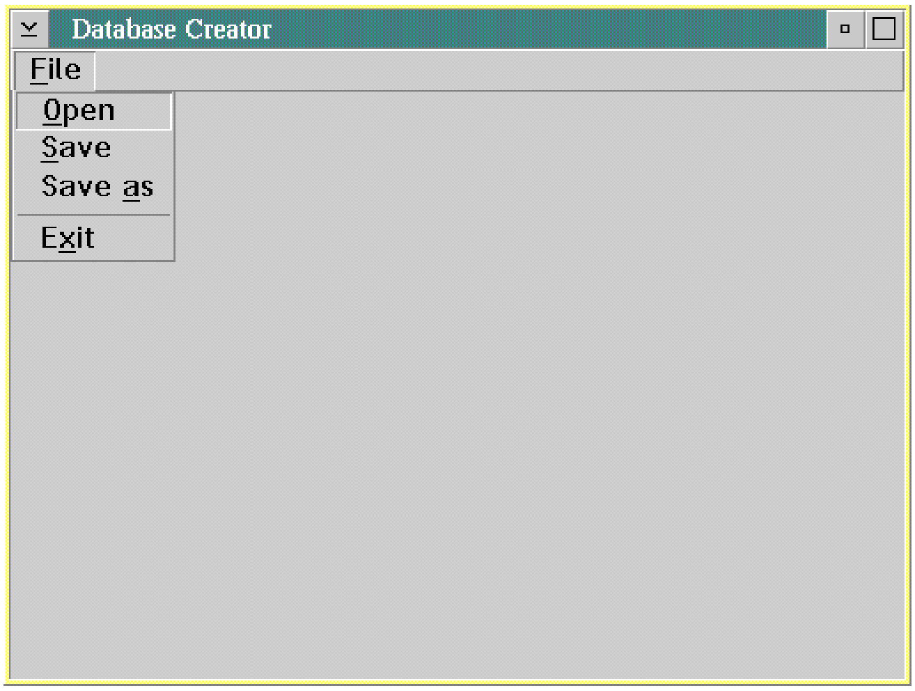

Figure 2.11: This figure shows an instance of the class `StdWindow`. At creation of the window, the flags `FCF_MENU`, `FCF_SIZEBORDER` and `FCF_ACCELTABLE` were specified

.

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:`.

2.26 StdWindow

Inherits from: ActionWindow : Window : Object `free` destroys the PM window and frees all resources allocated previously.

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

`setSize::::` sets the size of the window to `(w/h)` and its position to `(x/y)`.

- setRect: (LONG) w : (LONG) h;

Just as with `setSize::::` `setRect:` is used to set the size of the window. In contrast to `setSize::::` the position of the window is not changed.

The size of the window is set to `(w/h)`.

- (LONG) framexoffset;

`framexoffset` returns the horizontal offset of the frame window relative to the lower left corner of its parent window (normally the desktop).

- (LONG) frameyoffset;

This method returns the vertical offset of the frame window relative to its parent window.

- (LONG) framewidth;

`framewidth` returns the width of the frame window.

- (LONG) frameheight;

`frameheigth` returns the heigth of the frame window.

- (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;

`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:`.

2.27 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.

2.28 TriStateButton

Inherits from: BUTTON : WINDOW : OBJECT

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: (ULONG) 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`.

- 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.

- invalidate;

Calling `invalidate` causes the display area occupied by the window to be invalidated. As a consequence of this, the window is redrawn.

- show;

If the window was previously hidden (either by using the `hide`-method or by not specifying `WS_VISIBLE` at creation time, the window object is shown.

If the window is already visible, this method has no effect.

- hide;

`hide` hides the window object. It can be made visible again using `show`.

- (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 3

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

3.1 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.

Part II

Database programming classes

Chapter 4

Overview

This part of the reference manual describes the classes provided by the database library. Figure 4.1 shows all classes implemented by this library.

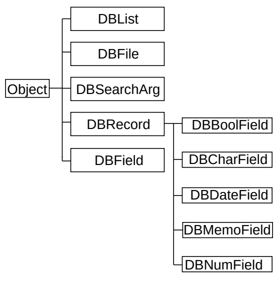

Figure 4.1: Inheritance hierarchy in Database Class library

Before using any of the classes in one of your source code files, include `<db/db.h>`. The object files must be linked with `objcdb.a`. This can be accomplished by specifying the linker option `-lobjcdb` in addition to `-lobjc`.

An introduction into using this classes can be found in the tutorial.

Methods and classes not listed here should *not* be used by the application programmer.

4.1 DBBoolField

```
@interface DBBoolField : DBField
{
}

@end
```

4.2 DBCharField

```
@interface DBCharField : DBField
```

```
   long      currentRecord;
}

- init:(char *) fileName;
- create: (char *) fileName withFields: (int) count
    list: (DBFIELD *) fields;
- free;

- field: (int) fieldNumber;
- (int) fieldCount;

- readRecord: (long) offset;
- writeRecord: (long) offset;
- (long) currentRecord;
- (BOOL) deleted;

- append;
- replace;
- delete;
- undelete;
- clear;

- (BOOL) findFirst;
- (BOOL) findNext;

- (void *) copyBuffer;
- (void *) copyBufferTo: (void *) aBuffer;
- setBuffer: (void *) aBuffer;

- (long) recordCount;

@end
```

4.6 DBList

```
@interface DBList : Object
{
  DBRecord *firstRecord;
  DBFile   *database;
  int       count;
}

- init;
- initForDatabase: (DBFile *) aDatabase;
- free;

- insertRecord: (DBRecord *) aRecord;
- insertRecord: (DBRecord *) aRecord at: (int) index;
- deleteRecordAt: (int) index;
- findRecordAt: (int) index;

- fetchAllRecords;
- setNext: (DBRecord *) aRecord;
- next;

@end
```

Chapter 5

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

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

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.

5.1 DBBoolField

Inherits from: DBFIELD : OBJECT

Class description:

`DBBoolField` is a a special class for handling of fields storing boolean values.

At the moment, no additional functionality to its superclass `DBField` is provided.

5.2 DBCharField

Inherits from: DBFIELD : OBJECT

Class description:

`DBCharField` is a a special class for handling of fields storing string values.

At the moment, no additional functionality to its superclass `DBField` is provided.

Methods:

- initWithName: (char *) aName andLength: (char) aLength andDecimals: (char) someDecimals;

`initWithName: andLength: andDecimals` is used to initialize a `DBField` object. The first parameter, `aName`, should be a `NULL`-terminated string and represents the name of the database field.

`aLength` specifies the total length of the data stored in the field in bytes. `someDecimals` represents the number of decimals stored.

If `someDecimals` is greater than `0`, the total length of the numeric value stored is `aLength - 1 - someDecimals` digits before the comma, and `someDecimals` decimals.

- free;

Free the storage allocated for this object and all following database fields, which are stored in `next`.

- setData: (void *) aPointer;

Using this method, the pointer to the data area for this field in the internal record buffer can be set. Without using this method, `data` is initialized to `NULL`.

- (char *) data;

This method returns the pointer to the data area used to store the data for this field in the internal record buffer.

- add: (DBField *) newField;

Using `add:` the initialized object `newField` is appended to the list of fields.

- next;

`next` returns a pointer to the next field or `nil` if this one is the last.

- setString: (char *) aString;

This method is used to modify the data in the internal record buffer.

`aString` is a `NULL`-terminated string representing the data to be stored. The data is copied into the record buffer.

- (char *) string;

`string` returns the data currently stored in the internal record buffer for this field as a `NULL`-terminated string.

5.5 DBFile

Inherits from: Object

Class description:

- create: (char *) fileName withFields: (int) count list: (DBFIELD *) fields;

The method previously described (`init:`) can only be used to open an existing database file. When you want to create a new database file, you have to specify all necessary header information to write a new template database.

`fileName` is the name of the database file, which shall be created. If this file already exists, it is overwritten.

`count` specifies the number of fields the database shall contain.

The information what fields shall be stored is passed in the `fields` parameter. `fields` is an array of structures of type `DBFIELD`. This type is defined in `<db/dbtypes.h>`.

You must fill in the following fields

- `name` ··· must be a `NULL`-terminated string of uppercase letters (for compatibility reasons) with at most 11 characters (including the terminating `NULL` character). This is the name of the field.

  Use `toupper ()` to convert the name to uppercase letter.
- `type` ··· specifies the type of data stored in this field. This can be `DB_FLD_CHAR` for character fields (strings), `DB_FLD_NUM` for numeric fields (with and without decimals), `DB_FLD_LOGIC` for fields storing logic values (boolean values), or `DB_FLD_DATE` for fields storing dates.

  `DB_FLD_MEMO` should *not* be used because handling of memo fields is not implemented now.
- `data_ptr` ··· is the pointer to the data for this field in the record buffer. This is calculated at creation of the database. You must initialize this variable to `NULL`.
- `length` ··· is the length of the field data in bytes. The datatype used for this variable is `unsigned char`.

  Remember to count all characters to be stored. When using numeric data with decimals, `length` is `decimals + 1 +` *number of digits before comma.*

  When using a date field, `length` *must* be `8`.

  Character data is stored without a terminating `NULL` character.
- `dec_point` ··· is an `unsigned int` designed to hold the number of decimals for numeric fields. For all other field types this must be set to `0`.

So creating a simple database with two fields, where the first is designed to store a string with at most 10 characters and a numeric field with 5 digits before and 2 after the comma would look like this:

```
.
.
DBFile  *newDatabase = [DBFile alloc];
DBFIELD  fieldinfo[2];

/* set information for first field */
strcpy (fieldinfo[0].name,"FIELD 1");
fieldinfo[0].type = DB_FLD_CHAR;
fieldinfo[0].data_ptr = NULL;
fieldinfo[0].length = 10;
fieldinfo[0].dec_point = 0;

/* set information for second field */
strcpy (fieldinfo[1].name,"FIELD 2");
fieldinfo[1].type = DB_FLD_NUM;
fieldinfo[1].data_ptr = NULL;
```

Write the record in the internal record buffer to the database file. A new record is appended.

This method is equal to using `[database writeRecord: [database recordCount]]`.

- replace;

Replace the current record in the database file with the data in the internal record buffer.

This is equal to using `[database writeRecord: [database currentRecord]]`.

- delete;

Mark the current record as deleted. The information is *not* written to the database automatically. Don't forget to call `replace` after modifying the record.

- undelete;

Mark the current record as not deleted. The information is *not* written to the database automatically. Don't forget to call `replace` after modifying the record.

- clear;

`clear` clears the record buffer. All values are reset to their defaults. You should always call this method before setting up the record buffer to append a new record.

- (BOOL) findFirst;

To access all records in the database, you should use `findFirst` and `findNext`.

`findFirst` searches for the first not deleted record in the database and returns `YES` on success.

If no active record could be found, `NO` is returned.

Visiting all records in a database can be accomplished using the following piece of source code

```
.
.
if ([database findFirst])
  do {
    .
    . /* do modifications to records */
    .
  } while (![database findNext]);
.
.
```

Here, `database` is assumed to be a pointer to a successfully initialized `DBFile` object.

- (BOOL) findNext;

`findNext` searches for the next record in the database file which is not marked as deleted.

The search *must* have been initially started using `findFirst`.

- (void *) copyBuffer;

`copyBuffer` creates a new buffer area in memory. This buffer area is filled with the contents of the internal record buffer. The newly created buffer area is returned.

This buffer area has to be freed again after using it with `free ()`.

- (void *) copyBufferTo: (void *) aBuffer;

This method copies the internal record buffer to a new buffer area pointed to by `aBuffer`. After completion, `aBuffer` is returned.

This method appends a new record `aRecord` to the end of the record list. `aRecord` must be a valid instance of `DBRecord`.

- insertRecord: (DBRecord *) aRecord at: (int) index;

This method inserts a new record `aRecord` into the record list. `aRecord` must be a valid instance of `DBRecord`.

`index` is the index in the record list, where the record gets inserted.

Enumeration starts at `0`.

- deleteRecordAt: (int) index;

Delete the record at position `index`. Enumeration of the records starts at `0` and ends at `[dblist count] - 1`.

This does not mark the record as deleted. This only deletes the record from the list. All modifications to this record are lost.

- findRecordAt: (int) index;

Return a pointer to the `DBRecord` object of the record a position `index`. Enumeration of the records starts at `0` and ends at `[dblist count] - 1`.

The record returned can be modified with the appropriate methods of `DBRecord`.

- fetchAllRecords;

Using `findFirst` and `findNext` all active records are retrieved into this object.

- setDatabase: (DBFile *) aDatabase;

Associated the database object `aDatabase` with the record list.

- (DBFile *) database;

This returns the associated instance of `DBFile`.

- (int) count;

`count` returns the number of records stored in the database file.

5.7 DBMemoField

Inherits from: DBFIELD : OBJECT

Class description:

`DBMemoField` is a a special class for handling of fields storing multiple lines of text.

Memo fields are currently not supported.

At the moment, no additional functionality to its superclass `DBField` is provided.

5.8 DBNumField

Inherits from: DBFIELD : OBJECT

Class description:

- findAt: (int) index;

This method returns the record object a index `index` in the linked list of records. If `index` is `0` `self` is returned.

If the index is out of range, this method returns `nil`.

Normally, you should not use this method.

- copyToDB;

`copyToDB` copyies the record buffer of this object to the internal record buffer of the database object.

The number of the record in the database file is *not* set. So don't try saving the data using `[database replace]`.

- copyFromDB;

`copyFromDB` retrieves the data stored in the internal record buffer of the database object to the record buffer of this object.

- replace;

This method copies the record buffer to the database object and replaces the record associated with this object on disk.

- setChanged: (BOOL) value;

Using `setChanged:` you can modify the value of the instance variable `changed` by hand. `changed` is set to `value`.

- (BOOL) changed;

The method `changed` returns the value stored in the instance variable `changed`.

Normally, `YES` is returned, if the record got modified, otherwise `NO` is returned.

- (long) recNo;

`recNo` returns the number of the record in the database file, which is associated with this object.

- setNext: (DBRecord *) aRecord;

Using `setNext:` the initialized `aRecord` is set as the next element in the linked list of records.

- next;

`next` returns a pointer to the next record in the linked list.

Bibliography

[1] *Introduction to PM Programming.* Salomon, Larry Jr. Article series starting in EDM/2 Vol. 1, issue 7.

[2] *OS/2 Version 2.0, Volume 4: Application Development.* IBM International Technical Support Center, Boca Raton. IBM Document Number GG24-3774-00.

[3] *Programming the Container Control.* Salomon, Larry Jr. Article series in EDM/2 Vol. 1, issues 3–5.

List of Tables

List of Figures

Index