The ButtonFace Library

Version 1.0

Software & Documentation:
Harald Albrecht
Institut für Geometrie und Praktische Mathematik
Rheinisch-Westfälische Technische Hochschule Aachen
Bundesrepublik Deutschland
albrecht@igpm.rwth-aachen.de

Proofreading:
Chris Marotz
Sterling Software ITD
Bellevue, NE
chris_marotz@itd.sterling.com

June 1994

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One great advantage of so called "Graphical User Interfaces" – or GUIs for short – is the graphical component. This can be traditional graphics like bar and pie charts as well as mathematical plots. But this really isn't the whole story. The term "graphical" instead often refers to the use of tiny pictures to represent operations or objects like trash cans or other useful things. These tiny pictures are also known as "icons"[1]. The intention is to aid or speed up the user's recognition process by bypassing the step of text recognition.

Whereas the use of icons as program symbols is very common today, most Motif applications heavily rely on the textual representation of information. Thus they often look like frontends developed some centuries ago to be used with those noisy ancient teletypes. Push buttons especially needed a polish-up. This is where the *"ButtonFace Library"* comes in. It particularly focuses on pictural push buttons and labels as well as on message or notification dialogs. The *"ButtonFace Library"* provides all you need to easily create such push buttons displaying pictures. The library itself is built on top of Arnaud Le Hors' famous X pixmap library[2] so you should have at least some understanding of Xpm pixmaps. The *"ButtonFace Library"* also helps saving resources on X servers. If you create more than one push button or label displaying the same picture then only one pixmap is created (assuming that all buttons are displayed on the same display and use the same background color). This sharing of pixmaps therefore helps reducing the "resource hunger" of applications.

Enough words for now. Now let us see how to use the *"ButtonFace Library"*. In the following you get a thoroughly introduction.

But before you start you should take a look at the `Makefile` and edit it in order to adjust it accordingly so it reflects your platform's configuration. Especially pay attention to the settings of **`CCOMP`**, **`INCLUDE_XPM`** and **`LIB_XPM`**. The first setting determines which compiler to use and how to put it into ANSI mode. (Sorry – no more mess with the old K&R style!) Luckily, we live in a time where you can get at least one ANSI compliant C compiler for nearly every platform: `gcc` (even better: it's (nearly) for free!)[3]. The settings of **`INCLUDE_XPM`**

[1] *One should recall that the term "icon" has its origin in sacred art. It originally designate one particular kind of paintings with religous themes.*

[2] *Tested with 3.2g as well as with 3.4. You'll find the Xpm library on many ftp sites around the world.*

[3] *People often believe that things for free have no quality. In the last ten years I've had enough opportunities to gain more and more experience with that "high quality commercial software". To be honest,* **ISO 9000** *(QA) is just a bad joke concerning today's commercial software... or just another way of employing*

and `LIB_XPM` specify where to find the include file of Arnaud Le Hors' Xpm library as well as the `libXpm.a` itself. The source of the Xpm library is not included in this package so you have to get it from somewhere else[4].

On some platforms you may also edit in the `Makefile` the `AR` and the `RANLIB` settings for creating and indexing libraries.

The *"ButtonFace Library"* itself consists of three object modules put together into one library file with the name `libButtFace.a`. It will be created automatically whenever you compile one of the following demo progams. But you can also create it at every time by doing a `make libButtFace.a`.

1.1 First steps

The easiest way to learn more about the facilities of the *"ButtonFace Library"* is to take a look at some examples. The first one is a fairly simple version of the well known exercise "my first X application". This time it simply displays a push button with a pixmap on its face instead of the usual "hello world!" message.

To keep this first program simple we don't create our own button face pixmap but instead use one of the predefined buttons. Below there is the required piece of code. In addition you'll find this example as the file `xhello1.c`. If you want to give it a try, run the command `make xhello1`. This will compile and link `xhello1` (provided that you've adjusted the `Makefile` accordingly as mentioned above).

```
/*
 * xhello1.c -- A (modified) "My First Program"
 *
 */

#include <Xm/Xm.h>
#include <pushbuttons.h>

void ExitCB(Widget w, XtPointer pClientData, XtPointer pCallData)
{
    exit(0);
} /* ExitCB */

int main(Cardinal argc, char **argv)
{
    static char  *Fallback[] = {
        "*.background: Gray75",
        "*.fontList: -*-helvetica-bold-r-normal-*-*-120-*-*-*-*-iso8859-1",
        NULL};
    XtAppContext AppCtx;
    Widget       TopLevel, Button;
```

more people and keep money floating around.

[4] *You can find it at* `ftp.x.org` *(*`198.112.44.100`*) in* `/contrib/libraries/xpm-3.4`*x*`.tar.gz`

```
    TopLevel = XtVaAppInitialize(&AppCtx, "XHello", NULL, 0,
                                 &argc, argv, Fallback, NULL);
    Button = XmCreateStandardPushButton(TopLevel, "ok", XmPUSHBUTTON_OK,
                                        NULL, 0);
    XtAddCallback(Button, XmNactivateCallback,
                  (XtCallbackProc) ExitCB, NULL);
    XtManageChild(Button);
    XtRealizeWidget(TopLevel);
    XtAppMainLoop(AppCtx);
    return 0;
} /* main */

/* End of xhello1.c */
```

With most of the code you should already be familiar with. The only new affair is a call to the function `XmCreateStandardPushButton()`. Don't even try to look it up in the OSF's documentation. It's a new convenience function exported by the *"ButtonFace Library"*. As you may already have guessed a call to this function creates a push button with a picture on its face.

The library comes with a set of predefined pictures. You simply specify which one to use in the third parameter of the call to `XmCreateStandardPushButton()`. In our case this is `XmPUSHBUTTON_OK` – a constant defined in `pushbuttons.h`. As with most convenience functions, you may supply additional resource settings in the fourth and fifth parameter. A more detailed oversight of all functions exported by the *"ButtonFace Library"* can be found in the next chapter.

constant name	*button description*
`XmPUSHBUTTON_NONE`	only three colorful question marks
`XmPUSHBUTTON_OK`	Okay: accept something
`XmPUSHBUTTON_CANCEL`	Cancel: cancel operation
`XmPUSHBUTTON_ABORT`	Abort: abort operation
`XmPUSHBUTTON_RETRY`	Retry: try to retry the last operation
`XmPUSHBUTTON_IGNORE`	Ignore: ignore your X application
`XmPUSHBUTTON_YES`	Yes: accept something
`XmPUSHBUTTON_NO`	No: deny something
`XmPUSHBUTTON_HELP`	Help me!

Table 1.1: Available predefined push buttons

It is the design goal of the *"ButtonFace Library"* to be as nearly invisible as possible. (To the programmer of course. The user however shall be impressed as much as possible by the outer appearance of your famous Motif application.) That's why the *"ButtonFace Library"* shares Motif's `XmXXXX` name space[5].

In addition to push buttons showing fancy pictures the *"ButtonFace Library"* provides this functionality with Motif's label widgets too. The convenience function `XmCreate-`

[5] *Hopefully the OSF don't doom me therefore. Anyway, the author has surely attracted the OSF's anger with his own* `XmComboBoxWidgetClass`*...*

`StandardLabel` can be used for this task whenever you want to use one of the predefined pictures. They are mainly used with message boxes notifying users of errors or other messages. These message boxes are covered in a later section. Table 1.2 shows the available pixmaps for label widgets.

constant name	*label description*
`XmLABELFACE_INFO`	An "*i*" like the well-known information icon
`XmLABELFACE_QUESTION`	A large question mark
`XmLABELFACE_EXCLAMATION`	A large exclamation mark
`XmLABELFACE_HAND`	A large hand indicating "no, no, no!"
`XmLABELFACE_TIME`	My reference to the spanish artist Dalí. A smearing clock indicating that a time consuming process is undergoing.

Table 1.2: Available predefined label pictures

1.2 Crafting your own picture buttons

Although you will probably use in almost every application one or more of the predefined push button pixmaps, you also may want to create your own picture buttons. Before we take a look at the second example program, here are a few clues about crafting your own pixmaps.

During the last year while I'm using pictural push buttons in my own Motif applications, the following guidelines have been proven to be proper.

The pixmaps to be used with picture push buttons should be 64×40 pixels tall. This has been proven to look good on a large variety of screen resolutions ranging from 800×600 on tiny 13" displays up to 1280×1024 on big 20" tubes. Whereas on tall displays such push buttons remain easily recognizeable they do not appear oversized with small resolutions.

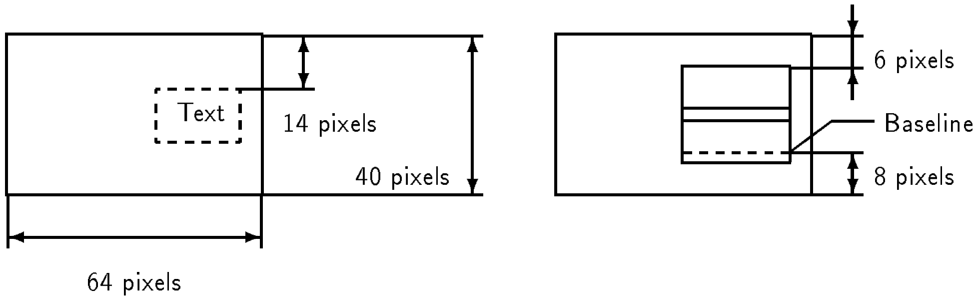

Figure 1.1: Design rules for placing text

The font to be used for writing text next to the button's picture should be a 14 pixel tall narrow Helvetica font[6]. All predefined push buttons of the "*ButtonFace Library*" already use that font. It is important not to mix different fonts (and font sizes) for the push buttons

[6] *also known as* `-*-helvetica-medium-r-narrow-*-14-*-*-*-*-*-*`

within the same application – this will give your customers no good impression. Try to keep the text as short as possible (you'll have not much spare room in the button). To achieve best results you may have to edit the text by hand to smooth it somewhat. Especially the letters "O", "Q" and "C" need to be polished up so they become more readable.

If you need to display two lines of text on the push button place the rectangle which surrounds the first line's text 6 pixels off the pixmap's top border. The baseline of the second line of text then has to be placed 8 pixels off the bottom line. Always remember to leave a space of at least 2 pixels to the right side of the text string. This room will be needed later when crafting the armed pixmap.

And now some rules for crafting the pixmap which is displayed whenever the push button is in its armed state. In order to get the impression that the button looks pressed in, the text has to be shifted by 2 pixels to the right as well as to the bottom. The picture may stay in its place, but in this case the shadow under the picture should grow to give the user the impression of pushing in the button. For this, take a look at the predefined pixmaps to see how all these design rules work together. You'll find the pixmaps in the `./pixmaps/eng/` directory.

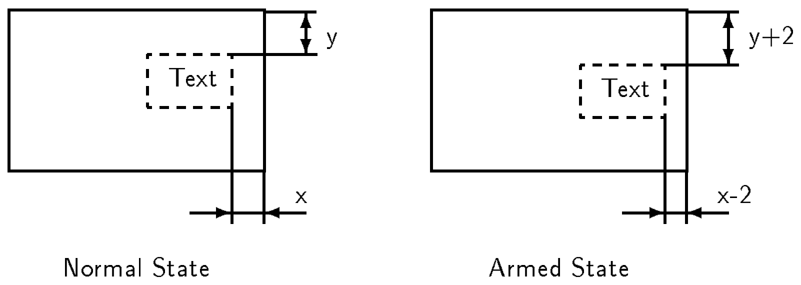

Figure 1.2: More design rules for placing text

As the background color you should use the special color "none". This results in pixmaps using the push button's background for every pixel with the color "none". Not every painting program is capable of handling this special color. In this case, use a distinctive color as the background when crafting the pixmap and save it as a xpm file. Then edit the xpm file and replace the corresponding color entry with this entry: "*pixelchar* `s None c None`". In the following example I've used a color with the RGB code of "#424242424242" as the background. Thus the header of your pixmap file may look like this

```
/* XPM */
static char * QuitPB_xpm[] = {
"64 40 6 1 0 0",
"       c #424242424242",
".      c black",
```

The fourth line specifying the color for all background pixels has to be replaced (the first blank in that line defines all spaces in the following pixmap's ASCII representation to be background pixels). In almost every case the background color is represented by a space.

```
/* XPM */
```

```
static char * QuitPB_xpm[] = {
"64 40 6 1 0 0",
"       s None  c None",
".      c black",
```

After all these clues and diagrams, we can now proceed with our second example. This time, the push button should display a "Quit" push button. To create your own push button you'll need at least one pixmap. But in order to achieve better results you should always use two pixmaps – one for the normal state and one for the armed state. Normaly you'll also need a third pixmap which is displayed whenever a button is insensitive. But if you don't supply it, the *"ButtonFace Library"* creates a suitable one for you. The *"ButtonFace Library"* achieves this by stippling the normal state's pixmap with a 50% stipple drawn in the background color. Thus the insensitive pixmap looks like dimmed.

```
/*
 * xhello2.c -- A (modified) "My First Program", now with user defined
 * push button face.
 */

#include <Xm/Xm.h>
#include <pushbuttons.h>

#include "QuitPB.xpm"  /* The pixmaps to be displayed */
#include "QuitPBa.xpm"

void ExitCB(Widget w, XtPointer pClientData, XtPointer pCallData)
{
    exit(0);
} /* ExitCB */

int main(Cardinal argc, char **argv)
{
    static char  *Fallback[] = {
        "*.background: Gray75",
        "*.fontList: -*-helvetica-bold-r-normal-*-*-120-*-*-*-*-iso8859-1",
        NULL};
    XtAppContext AppCtx;
    Widget       TopLevel, Button;

    TopLevel = XtVaAppInitialize(&AppCtx, "XHello", NULL, 0,
                                 &argc, argv, Fallback, NULL);
    Button = XmCreatePixmapPushButton(TopLevel, "ok",
                                        QuitPB_xpm, QuitPBa_xpm, NULL,
                                        NULL, 0);
    XtAddCallback(Button, XmNactivateCallback,
                  (XtCallbackProc) ExitCB, NULL);
    XtManageChild(Button);
    XtRealizeWidget(TopLevel);
    XtAppMainLoop(AppCtx);
    return 0;
```

```
} /* main */

/* End of xhello2.c */
```

This time you have to use `XmCreatePixmapPushButton` instead of `XmCreateStandardPushButton`. In the third, fourth and fifth parameter you can specify which pixmaps to use for the push button's different states: normal, armed and insensitive. The first pixmap must be always given but you can set the fourth and fifth parameter to `NULL` if you don't want to supply your own pixmap. In this case the *"ButtonFace Library"* creates the missing pixmaps based on the normal state's pixmap.

1.3 Message boxes – the easy way

You've surely used many times one of Motif's predefined message dialogs. There are plenty of chances you'll need it: prompting the user before overwriting or erasing her or his files, reporting error conditions and much more. But to be honest: it seems that the OSF didn't spend much time on the message dialog's design. Thus the outer appeareance is rather... *hmmm...* basic.

Sometimes it is difficult to follow the common way of creating and using message dialogs. For example, you have to prompt the user within a function saving the user's data to disk. The usual way is to create a message dialog and then to set up the necessary callbacks. Afterwards leave the function and return to the message loop once again. You can't save the data to disk until the callback for the "Ok" push button has been activated. Thus the save operation (and the save function in C) must be divided into two separate functions: one checking if the file already exists and then prompting the user and another function which actually saves the data. The end result of this is scattered code.

Often it is much more convenient just to call *one* simple function to create the message dialog, waiting for the user's answer and then returning her/his answer. Therefore the *"ButtonFace Library"* provides such a nice convenience function called `XmMessageBox`. You can simply call it "on the fly" without the need to explicitly return to the event dispatching loop. To achive this `XmMessageBox` provides it's own message dispatch loop. In case you're using your own dispatch loop you can supply a pointer to an external event loop as the last parameter to `XmMessageBox`. This external event loop isn't rather a real loop but a function which will be called repeatly by `XmMessageBox` in order to get and dispatch *one* event for every call, then it *must* return. Nevertheless the external event dispatch function is allowed to block if no pending events are waiting. It can also signal `XmMessageBox` that the current message box should be destroyed if it returns `False`.

As an example how to use `XmMessageBox` we'll modify `xhello2.c` to prompt the user before exiting. This is achieved by calling `XmMessageBox` within the exit callback. The following piece of code shows the necessary changes:

```
#include <msgbox.h>
  .
  .
  .
void ExitCB(Widget w, XtPointer pClientData, XtPointer pCallData)
```

```
{
    if ( XmMessageBox(w, "Quit...",
                      "Do you really want to quit?",
                      MB_YESNO | MB_DEFAULT2, NULL) == XmPUSHBUTTON_YES )
        exit(0);
} /* ExitCB */
```

Once again you can give it a try and compile and link `xhello3.c` by doing a `make xhello3`.

But now take a look at the parameters you supply to a call of `XmMessageBox`. The first one is obviously the parent widget for which the message dialog should be created. As for the second and third parameters, you specify the message dialog's title as well as the displayed text. To simplify this, these two strings are simple `C` strings – no mess with Motif's `XmStrings`. In the fourth parameter, you must supply the push buttons to be displayed.

constant name	*description*
`MB_OK`	Just the "Okay" push button
`MB_OKCANCEL`	"Okay" and "Cancel" buttons
`MB_YESNO`	"Yes" and "No" buttons
`MB_YESNOCANCEL`	"Yes", "No" and "Cancel"
`MB_RETRYABORT`	"Retry" and "Abort"
`MB_RETRYABORTCANCEL`	"Retry", "Abort" and "Cancel"

Table 1.3: Available push button combinations for `XmMessageBox`

modifier	*description*
`MB_HELP`	Display an additional "Help" push button.
`MB_DEFAULT1`...`MB_DEFAULT4`	Specifies which button should receive the keyboard focus when the message dialog is popped up. The push buttons are counted left to right. The default value is `Xm_DEFAULT1`.

Table 1.4: Modifiers for `XmMessageBox`

modifier	*description*
`MB_INFOICON`	Displays an *"i"* next to the text.
`MB_HANDICON`	Displays a hand.
`MB_QUESTIONICON`	Displays a question mark
`MB_EXCLAMATIONMARK`	Displays an exclamation mark
`MB_STANDBYICON`	Displays a smearing clock

Table 1.5: More modifiers (icons) for `XmMessageBox`

In addition to these predefined constants you can specify additional modifier flags as shown in table 1.4 by ORing them together with one of the `MB_xxx` constants listed in table 1.3. The `xhello3.c` source code shows an example of this: the "Cancel" button should receive the keyboard focus when the message dialog pops up on the user's display.

You can also specify an icon which is displayed next to the message dialog's text much like Motif's message boxes. Simply OR one of the constants found in table 1.5 to the `flags` parameter in the call to `XmMessageBox`. If you leave it out, the default value will be `MB_INFOICON`.

Please note that the "*ButtonFace Library*" provides the usual keyboard interface. The keys <Key>osfCancel (known as ESC) and <Key>osfActivate (labeled Return or Enter) work as expected. Pressing <Key>osfCancel cancels the message dialog by activating the cancel button if it is present. Whereas <Key>osfActivate triggers the push button which currently holds the keyboard focus. The message dialog also takes part in the `WM_DELETE_WINDOW` protocol. If you close the dialog window by activating "close" in the window's system menu the cancel pushbutton is activated. If the message dialog contains no cancel button, a push button with a similiar intention (like "abort") is used instead.

1.4 Message boxes – the hard way

Sometimes you don't want to use the built-in event dispatch loop and do it the common way instead. This is to first create the message dialog and then install the necessary callbacks on the push buttons. Then pop up the message dialog the usual way with a call to `XtManageChild`. Please note that this is Motif's way of popping up dialogs and not the original Xt Intrinsic's method.

Once more I'll show only the interesting piece of code. Please always carefully check out which parameters you specifiy in the call to `XmMsgBoxGetChild`. (Especially if you later change the set of used push buttons in the call to `XmCreateMsgBox`!) At least creating message dialogs this hard way agrees with the common opinion about treating our environment: just recycle any dialog. This time the message dialog gets created once and is used several times...

```
Widget MessageDialog = NULL;

void ReallyExitCB(Widget w, XtPointer pClientData, XtPointer pCallData)
{
    exit(0);
} /* ReallyExitCB */

void CancelCB(Widget w, XtPointer pClientData, XtPointer pCallData)
{
    XtUnmanageChild(MessageDialog);
} /* CancelCB */


void ExitCB(Widget w, XtPointer pClientData, XtPointer pCallData)
{
    if ( MessageDialog == NULL ) {
        MessageDialog = XmCreateMsgBox(w, "msgbox",
```

```
                            "Quit...",
                            "Do you really want to quit?",
                            MB_YESNO | MB_DEFAULT2, NULL, 0);
        XtAddCallback(
            XmMsgBoxGetChild(MessageDialog, XmPUSHBUTTON_YES),
            XmNactivateCallback,
            (XtCallbackProc) ReallyExitCB, NULL);
        XtAddCallback(
            XmMsgBoxGetChild(MessageDialog, XmPUSHBUTTON_NO),
            XmNactivateCallback,
            (XtCallbackProc) CancelCB, NULL);
    }
    XtManageChild(MessageDialog);
} /* ExitCB */
```

1.5 Internationalization

The *"ButtonFace Library"* is especially designed in such a way that it eases the process of internationalizing a Motif application[7]. You can exchange the pixmaps displayed on push buttons and labels on the fly without recompiling. You just add or replace the necessary entries in the application's resource file – and get a localized application. This way you can compile the source into an application for english speaking people and if you need to use a different language instead use a (different) resource file.

The *"ButtonFace Library"* is designed such that you don't have to supply the pixmaps as additional xpm files. Instead you add the xpm file content's to the resource file with the help of the `xpm2rsc` utility script. A simple example should clear this: imagine an application called `xfavouriteapp` that uses a resource file called `XFavouriteApp`. Within this application you're using a push button with the name "ok". Now you want to switch from english to german. The necessary pixmaps are already stored in the `./pixmaps/ger/` directory.

You then use `xpm2rsc` to add or replace the entries that specify which pixmaps to use:

```
xpm2rsc ./pixmaps/ger/OkayPB.xpm  XFavouriteApp \*.ok.normalFace
xpm2rsc ./pixmaps/ger/OkayPBa.xpm XFavouriteApp \*.ok.armedFace
```

As the first parameter you specify which xpm file has to be added to the resource file given as the second parameter. The third parameter then specifies on which push button you intend to set the pixmap. This can be one of the available three resource indentifiers listed in table 1.6. With the exception of `insensitiveFace`, all the other resource identifiers can also be applied to pictural labels. Don't forget to escape wildcard characters like * or ? so the shell can't do any damage on them.

Please note that these resource identifiers have no corresponding widget resource! Thus you can't use `XtSetValues` or `XtGetValues` on it! Whenever you create a pushbutton with

[7] *No wonder – the author of the "ButtonFace Library" is a "Kraut"... a person who likes unusual letters like ä,ö, ü and ß – and Sauerkraut of course (not to forget the famous bavarian beer!)*

`XmCreateStandardPushButton()` or `XmCreatePixmapPushButton()` the *"ButtonFace Library"* tries to look up the three resource entries in you application's resource file. If it finds a suitable entry it then compiles the given xpm entry into a real pixmap. If you don't supply a pixmap for the armed or insensitive state a suitable pixmap is created by the *"ButtonFace Library"* as already described above.

resource name	*description*
`normalFace`	The push button's normal face.
`armedFace`	Pixmap to be displayed when in armed state.
`insensitiveFace`	Pixmap to be displayed when button is insensitive.

Table 1.6: Available push button resources

You'll find a somewhat larger example program in `demo.c` which not only displays all available predefined push buttons and labels, but also shows how to easily internationalize an application. Compile it with – guess what: `make demo`. If you then run it using `./demo` the message dialogs will show english messages. Localization is then achieved by doing a `make I18N`[8]. This will create a resource file with the name `Demo.ad` in `./pixmaps/ger`. This file then contains the messages and pixmaps needed for a german demo application. To run the demo, enter on the command line:

```
XENVIRONMENT=./pixmaps/ger/Demo.ad; export XENVIRONMENT; \
./demo; unset XENVIRONMENT
```

With a real application the installation procedure should copy the necessary resource file to an appropiate place so users don't need to worry about such things.

The demo application now shows german messages. You should note the special convenience function `XmLoadStringResource` for loading the appropiate message string from a resource file. This way you can hardcode the english text into the application (although this is almost not a very good idea to do so.) But you can also override this with a resource file by specifying an appropriate entry.

In the file `demo.c` you can study how this is done:

```
Caption = XmLoadStringResource(w, "title1", NULL, "Important Help...");
Text = XmLoadStringResource(w, "question1", NULL,
                            "Please stand by while I'm trying\n"
                            "to find the answer to the question\n"
                            "of life, the universe and everything.");
```

The english strings are hardcoded so even in case no application resource file is installed the user gets a readable message. If you want to supply another piece of text then you add the necessary resource entries to the application's resource file. They may look like the following ones (where \344 represents the letter a umlaut $\hat{=}$ "ä"):

[8] *If you don't know what the term I18N refers to, here's the explanation: if you use only the letters I and N of "Internationalization" and count the letters within the I and N you'll count 18 letters. So some geeks abbreviate it as I18N.*

```
*.title1: Wichtige Hilfe...
*.question1: Bitte haben Sie einen Moment Geduld,\n\
    w\344hrend ich die Anwort auf\ndie Frage nach dem Leben, dem\n\
    Universum und dem Rest suche.
```

With this release of the *"ButtonFace Library"* you'll get not only all predefined push buttons with english inscription but also in addition a set of push buttons for german speaking users. You'll find the german push buttons in `./pixmaps/ger/`. Hopefully someone contributes localized pixmaps for other tongues – mainly french, spanish and khehasueli – since I can't do that myself.

Phew – what a long introduction. But now you've hopefully seen how to use the *"ButtonFace Library"* in your own applications. Now it's time to polish them up! Have much fun and enjoy you customers with better looking Motif applications!

Chapter 2

Convenience Functions

This chapter focuses on the available convenience functions. Below you'll find a list of all available functions and their parameters.

2.1 General utilities

`xpm2rsc` <xpm file> <resource file> <resource entry>

This is a tiny `/bin/sh` script relying heavily on `sed`. It reads in the xpm pixmap stored in **xpm file** and adds it to the resource file indicated by **resource file**. The pixmap is then accessible through the resource entry specified as **resource entry**. If such an entry already exists it'll be replaced. Please note that simply concatenating a xpm file to the end of a resource file won't work because xpm files contain bare line feeds. Therefore `xpm2rsc` replaces them with "`\n\`" so xpm files can be included into resource files.

2.2 Convenience functions in `pushbuttons.h`

```
Widget XmCreatePixmapPushButton(Widget Parent, String Name, char **Normal,
        char **Armed, char **Insensitive, ArgList Args, Cardinal ArgCount)
```

Creates a pictural push button from your own supplied pixmaps. As you might have already guessed **Parent** specifies the widget ID of the new push button's parent and **Name** is the instance name. At least in **Normal** you must supply a pointer to an xpm pixmap stored in memory in the usual textual form. If you set **Armed** and/or **Insensitive** to `NULL` the *"ButtonFace Library"* will create suitable pixmaps based on the pixmap given as **Normal**. You may specify additional resource settings in **Args** and **ArgCount**.

```
Widget XmCreatePixmapPushButtonGadget(Widget Parent, String Name,
        char **Normal, char **Armed, char **Insensitive,
        ArgList Args, Cardinal ArgCount)
```

Same as `XmCreatePixmapPushButton` but this time a gadget is created instead of a widget.

```
Widget XmCreateStandardPushButton(Widget Parent, String Name,
        int FaceType, ArgList Args, Cardinal ArgCount)
```

This convenience function creates one of the predefined pictural push buttons. Possible values for **FaceType** can be found in table 1.1.

```
Widget XmCreateStandardPushButtonGadget(Widget Parent, String Name,
        int FaceType, ArgList Args, Cardinal ArgCount)
```

Same as **XmCreateStandardPushButton** but this time a gadget is created instead of a widget.

```
Widget XmCreatePixmapLabel(Widget Parent, String Name,
        char **Normal, char **Insensitive, ArgList Args, Cardinal ArgCount)
```

Creates a pictural label widget from user supplied pixmaps. The parameters are the same ones as with `XmCreatePixmapPushButton` with the exception of **Armed**. This parameter isn't applicable to label widgets (for obvious reasons).

```
Widget XmCreatePixmapLabelGadget(Widget Parent, String Name,
        char **Normal, char **Insensitive, ArgList Args, Cardinal ArgCount)
```

Same as **XmCreatePixmapLabel** but it creates a gadget instead of a widget.

```
Widget XmCreateStandardLabel(Widget Parent, String Name,
        int FaceType, ArgList Args, Cardinal ArgCount)
```

This convenience function creates one of the predefined pictural label widgets. Possible values for **FaceType** can be found in table 1.2.

```
Widget XmCreateStandardLabelGadget(Widget Parent, String Name,
        int FaceType, ArgList Args, Cardinal ArgCount)
```

The same procedure as every... sorry – instead of creating a widget this convenience function creates a gadget.

2.3 Convenience functions in `msgbox.h`

```
int XmMessageBox(Widget Parent, char *Caption, char *Text, int Flags,
        XmEventLoopProc *ExternalEventLoop)
```

This is the simplest and easiest way to create a message dialog and wait for the user's answer. In **Parent** specify the parent widget for the shell of the message dialog. In **Caption** and **Text** you can specify the title as well as the message text itself. Both parameters are ordinary `C` strings. In **Flags** you specify which push buttons and what icon the message dialog should display. Please refer to tables 1.4 and 1.5 for a list of available constants. In the last parameter you can specify the address of an external event dispatching loop – or more precisely: the address of a one-event dispatch function. If you supply `NULL` as it's value `XmMessageBox` uses it's own internal event dispatch loop. If you supply a value this must be a pointer to a function of the type `XmEventLoopProc` as described in the next paragraph.

```
typedef Boolean (XmEventLoopProc)(XtAppContext)
```

This is the function prototype for a event dispatch function. Such a function is called repeatly by `XmMessageBox` in order to get *one* event and dispatch it. The function then should return `True` if it wants to continue dispatching or `False` if the message dialog should be closed immediately.

`Widget XmCreateMsgBox(Widget Parent, char *Name, char *Caption, char *Text, int Flags, ArgList Args, Cardinal ArgCount)`

If you want to create message dialogs the usual way, you have to use this function instead of the "microwave cooked" `XmMessageBox`. You must supply the parent of the message dialog as well as the dialog's widget name (in **Name**). The remaining parameters are the same as in a call to `XmMessageBox` with the exception of **Args** and **ArgCount**. With these two parameters you can specify additional resource settings when creating the message dialog.

`Widget XmMsgBoxGetChild(Widget w, int WhichChild)`

Query the widget ID of one of a message dialog's children. **w** points to the message dialog. Possible values for **WhichChild** can be looked up in table 1.1. If such a child does not exists `XmMsgBoxGetChild` returns `NULL`.

2.4 Convenience functions in `resource.h`

`char *XmLoadStringResource(Widget w, char *ResourceName, char *ResourceClass, char *Default)`

This function helps in reading resource entries out of the application's resource file. The first parameter **w** specifies which widget you want to query. **ResourceName** and **ResourceClass** specify the resources entry's name as well as it's class. You can set **ResourceClass** to `NULL`. In this case `XmLoadStringResource` will use the **ResourceName** as the class name with the first letter capitalized. You can specify a default value for the return value in **Default** in case no entry can be found.

Chapter 3

The XmCenter Widget

The *"ButtonFace Library"* includes a new widget called "XmCenter". It is mainly used by the message boxes for the purpose of centering the push buttons in a row within the dialog window. For this reason the center widget is a comparably easy to handle widget class. (In terms of modern advertising one would call it a "light" widget.) It is intended to be plugged into the existing widget tree and therefore shares the common XmNxxxx name space (although the OSF didn't like doing so – but who cares...). The common use of the center widget is the centering of one or more push buttons in a row or a column within a dialog. Below is a description of all the widget's resources.

This documentation follows the guide lines introduced by the standard literature on the X window system (attention: singular!) and especially Motif. Below you'll find a table, which contains information about all the widget resources of the center widget: their names and classes as well as the default values. In addition, the table shows whether a resource can be set at the widget's creation time (**C**) or get and set during the program's run time (**G** and **S**). The description of each resource then follows.

A widget inherits all of its resources from its ancestors, starting with its parent and continuing until the root of the family (class) tree, the core widget class. Since the center widget contains other child widgets, one of its ancestors must be the widget class Composite. To provide the accustomed keyboard interface with which the Motif user is already familiar, the center widget must be an ancestor of the Motif class XmManager. Only then the keys <Key>osfCancel (known as ESC) and <Key>osfCancel (labeled Return or Enter) will work as expected and can be used to confirm or cancel Motif dialogs. Other features of Constraint or XmManager, like the geometry constraints on its children, aren't needed anymore.

All those resources which the center widget inherits from its ancestors are not shown in this document. Instead, please refer to the Motif documentation in the case of XmManager or to the X toolkit documentation for the classes Core, Composite and Constraint. This is to keep this paper short.Please note that all resources representing a width or height share Motif's common way of resolution independence.

XmCenter Resource Set		
Name	**Default**	**Access**
Class	**Type**	
XmNhorizontalBorder	0	CSG
XmCHorizontalBorder	Dimension	
XmNhorizontalSpacing	8	CSG
XmCSpacing	Dimension	
XmNorientation	XmHORIZONTAL	CG
XmCOrientation	unsigned char	
XmNstrictSpacing	True	CSG
XmCStrictSpacing	Boolean	
XmNverticalBorder	0	CSG
XmCVerticalBorder	Dimension	
XmNverticalSpacing	8	CSG
XmCSpacing	Dimension	

Core Resource Set		
Name	**Default**	**Access**
Class	**Type**	
XmNborderWidth	0	CSG
XmCBorderWidth	Dimension	

XmNhorizontalBorder

When **XmNorientation** is `XmHORIZONTAL` this resource specifies the width of the border between the center widget's border and the leftmost child as well as between the right side of the center widget and the rightmost child. If **XmNorientation** is `XmVERTICAL` then the resource specifies the border to the right and to the left side of the row of children.

XmNhorizontalSpacing

Specifies the distance between children of the center widget when **XmNorientation** is `XmHORIZONTAL` and **XmNstrictSpacing** is set to `True`.

XmNorientation

Controls how the childen are positioned within the center widget. Possible values are either `XmHORIZONTAL` or `XmVERTICAL`.

XmNstrictSpacing

Specifies how the children are centered within the center widget. If this resource is set to `True` all managed children are placed tightly together as one entity. This block of children is then centered either horizontally or vertically within the center widget (depending on **XmNorientation**). When setting **XmNstrictSpacing** to `False`, all children are equally spaced among the center widget instead.

XmNverticalBorder

Much like **XmNhorizontalBorder** but this time this resource specifies the top border and bottom border width.

XmNverticalSpacing

Specifies the distance between children of the center widget when **XmNorientation** is `XmVERTICAL` and **XmNstrictSpacing** is set to `True`.

The following pictures should give you a knowledge how the resources affect the alignment of children. You should note that the center widget imposes the same height as well as the same width on everyone of its children. The child with the tallest width determines the width of *all* children within that center widget. The same also applies to the width.

The center widget can be configured to align its childen in either one row (when **XmNorientation** is set to `XmHORIZONTAL`) or one column (by setting **XmNorientation** to `XmVERTICAL`). Figure 3.1 shows the influences of some of the widget's resources on the layout in the case of **XmNorientation** is set to `XmHORIZONTAL`.

The top part of the figure shows the desired geometry a center widget will ask it's parent for during initialization. Below you'll see how the children are layed out within a XmCenter whenever this widget gets resized such that it is taller than needed.

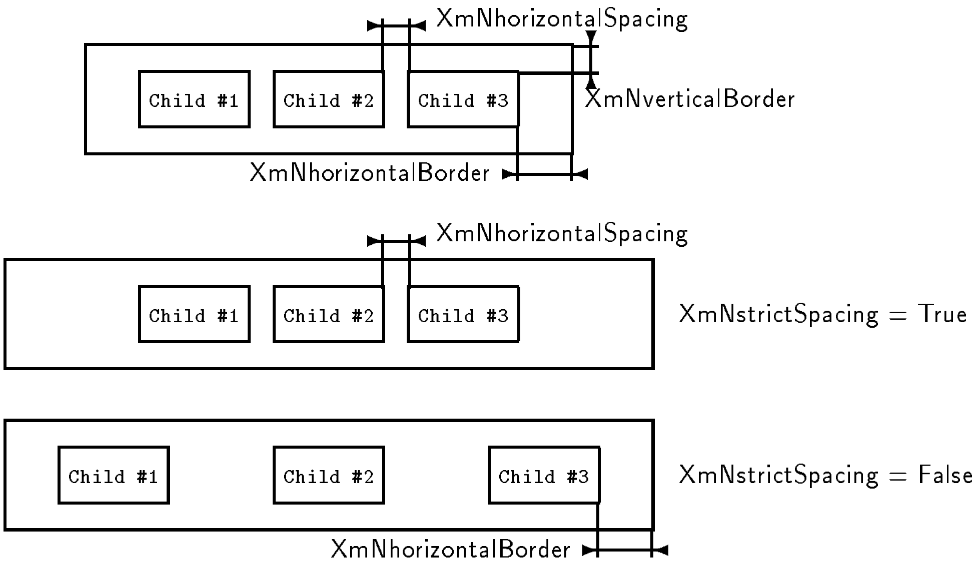

Figure 3.1: Horizontal centering of children of a center widget