# Libsx

A Simple X library

---

**Reference**

# Contents:

# Author's Preamble

Welcome to libsx, the simple X library. Libsx is an attempt to simplify the vagaries of programming under X windows, making it simple to create user interfaces for programs. With libsx, 10 lines of code may be all you need to create a user-interface complete with many different types of widgets.

Libsx is layered on top of the Athena widget set and basically acts as a front end to all the Athena and Xlib garbage so that programming reasonable interfaces isn't so painful. For example, libsx has a simple to use one-line string entry widget that you can create with a single function call (it's based on the Athena text widget but hides all the gory details). Libsx encapsulates the common operations people usually want to perform in a window system and makes them easy to accomplish (at the loss of some flexibility).

If you've ever wanted to just open a window with a few buttons and draw some graphics, but were turned away by the complexity of trying to do that, then libsx may be your ticket. Libsx is capable of easily creating many types of user-interface components each with a single function call of a few arguments. The library supports the following Athena "widgets":

- Labels
- Buttons
- Toggle buttons and Radio buttons
- String Entry areas
- Scrolling Lists
- Menus
- Scrollbars
- Drawing Areas
- Text Edit boxes

The goal of libsx was to make the creation and manipulation of each of these items as simple as possible. The standard simplicity litmus test is a "Hello World" program, which in libsx is:

```
#include "libsx.h"
main() {
MakeLabel("Hello World");
MainLoop();
}
```

More complicated interfaces use a similar style of creation and complete applications usually require less than 30 lines of code to create an entire user interface complete with menus, buttons, string entry widgets, etc. For example, to create an application that opens a window with a drawing area and a "Quit" button, all one must do is:

```
#include "libsx.h"
main() {
Widget quit_button, draw_area; quit_button = MakeButton("Quit", quit, NULL);
draw_area = MakeDrawArea(500, 500, draw_stuff, NULL);
SetWidgetPos(draw_area, PLACE_UNDER, quit_button, NO_CARE, NULL);
MainLoop();
}
```

So in only a handful of lines of code, we created a simple X application that would have required inordinate amounts of code using traditional methods. All that is required now is for the user to write the routines "quit" and "draw_stuff".

# Introduction

## How to use Libsx

Using libsx is pretty simple. At the minimum, you #include "libsx.h" and link with libsx.a. To actually have X windows pop open and such, you need to do the following:

(1) To get everything started, you should call OpenDisplay(). If OpenDisplay() returns a non-zero value, it's ok to go on. OpenDisplay() creates what will eventually be your first window.

(2) After calling OpenDisplay(), you can go on to create all sorts of widgets with the MakeXXX() calls. You can lay them out with calls to SetWidgetPos().

(3) When you are done creating the user interface, call ShowDisplay(). This causes the window and components you've created to be displayed on the workstation screen. Until you call ShowDisplay(), the user can NOT see your window, and drawing into drawing areas has NO effect. If you need to, you can call any of the color allocation functions such as GetStandardColors(), etc.

(4) Finally, once the window is displayed and you've done all the initializations you wish, you must then call MainLoop(). After you call MainLoop(), events get processed as they come in and your callback functions are called as necessary. After calling MainLoop(), the correct way for your program to exit is to have one of your callback routines call exit() when appropriate (like after the user clicks on a "Quit" button).

That's all you need to do. Even though that may look like a lot to do, it's really pretty simple in practice. For example, here is a hello world program with libsx:

```
#include "libsx.h"

main()
{
MakeLabel("Hello World!");
MainLoop();
}
```

Granted it's one more line than a standard printf() type of hello world program, but it's not all that bad.

Hello world programs are nice, but you don't tend to write very many of them. Real applications need to be able to do much more. Even these "real" programs aren't all that bad in libsx.

Here is a simple program that opens a window with a quit button and a drawing area that you could use to draw whatever graphics you wanted:

```
#include ¡stdio.h¿
#include "libsx.h"

void quit(Widget w, void *data)
{
exit(0);
}
```

```
void draw_stuff(Widget w, int width, int height, void *data)
{
ClearDrawArea();
DrawLine(0,0, width, height); /* just draw a diagonal line */
}

int main(int argc, char **argv)
{
Widget w[2];

argc = OpenDisplay(argc, argv);
/* woops, couldn't get started */
if (argc == 0)
exit(5);


w[0] = MakeButton("Quit", quit, NULL);
w[1] = MakeDrawArea(300,300, draw_stuff, NULL);
SetWidgetPos(w[1], PLACE_UNDER, w[0], NO_CARE, NULL);
ShowDisplay();
GetStandardColors();
/* off we go! */
MainLoop();
}
```

The code above is the basic skeleton for a libsx program, even complicated ones. First you open the display with OpenDisplay(). Then you build your interface by creating a bunch of widgets with the MakeXXX() calls. Next you layout the display by specifying the relative positions of the widgets to each other. Then you would get any fonts or colors you may need, and finally you just enter the main loop.

In libsx, your callback functions are where all the real work happens. The program above has two callback functions, quit() and draw_stuff(). They are tied to events that happen in the interface. When the user clicks on the "Quit" button, your quit() function is called. When the drawing area gets resized or needs to be redrawn, your draw_stuff() function gets called.

Usually the process of creating the interface would get separated into a separate function that is easy to modify (instead of cluttering up main). However, the basic outline is the same as above. The only real difference with more complicated interfaces is that they usually have a lot more calls to the MakeXXX() functions and they tend to make use of the extra void pointer argument in the callback routines.

If you'd like more examples, take a look at the provided source code. There are several reasonable examples of varying complexity that you can take and modify as you like. Each of the demos tries to demonstrate a certain group of features, so take a look at each to find the one that most closely matches what you want to do and start hacking from there!

Have fun.

Dominic Giampaolo
dbg@sgi.com

# Chapter 1

## Buttons

A button widget is a button that a user can click on with the left mouse button to indicate an action. When a button is pressed, it is drawn in inverse video and some action takes place.

A button is connected to your code by a callback function which is called when the user clicks on the button widget with the left mouse button.

## MakeButton()

Widget MakeButton(char *label, ButtonCB func, void *data);

This function creates a small rectangular button which the user can click on. The character string pointed at by "label" will be printed inside the button. If the string has newline characters in it, they will be interpreted properly (i.e. you will get a multiline label). The next two arguments are a callback function, func, and an arbitrary data pointer, data, that will be passed to the function func.

If you plan to attach a bitmap to this widget, you can specify NULL for the label text (see the docs for SetWidgetBitmap().

When the button is pressed, the function, "func" will be called. The function, func, will have two arguments. The first argument is the widget that user clicked on (which you can ignore if you do not need it). The second argument is the void pointer, 'data', specified in the call to MakeButton(). The function "func" should be declared as follows:

```
void func(Widget w, void *data)
```

The last argument (called "data") is the same as the last argument to the MakeButton() function call. It is declared as a void pointer and you can cast to whatever type you need. Generally you'll have something like:

```
MyProgram *me = (MyProgram *)data;
```

You use "buttons" to allow the user to indicate various actions (things like load, save, cut, copy, and paste operations are good examples). The mental model of how a button works is that when the user clicks on the button with the mouse, the function you specify is called.

If something goes wrong in creating the button, a NULL value is returned. This is a rare occurrence, but good code will still check for it.

SEE ALSO : SetWidgetPos(), SetWidgetBitmap(), SetLabel(), SetWidgetFont(), SetFgColor(), SetBgColor(), SetBorderColor()

# Chapter 2

## Colors

This chapter describes the routines for managing colors in your window. For example if you want to change what the foreground color is, or need to get specific colors. To get specific colors you use the functions discussed here. It is important to remember that you can not call any of these functions until you have called ShowDisplay().

Colors are represented by integers. When you get a color, you are returned an integer that you can use in calls to SetFgColor(), SetBgColor(), and SetColor(). You should attach no meaning to the numbers, and just because green is 17 does not mean that 18 is a lighter or darker shade of green.

There are three ways to manipulate colors with libsx. The first way handles most of the common cases, and is done with GetNamedColor() or GetRGBColor().

The next method, GetPrivateColor(), allows your application to modify the actual display color represented by a color number (something you cannot do with the the previous methods).

The final method gives you complete control in specifying the entire colormap. That is, you can determine exactly what integers map to what colors so you can obtain smooth gradients (so for example black is color 0, and white is 255). These routines work best on 8 bit displays but will work on 24 bit displays.

NOTE: You can NOT call any color function until you have called ShowDisplay().

The way colors work for drawing is like this. There are usually 256 available colors on a workstation. This is called an 8-bit display because 2 to the 8'th power == 256. These colors are stored in a table (array) of 256 entries. If you allocate a color, and it is in entry 37, then to draw with the color that is stored there, you must use 37 as an argument to the SetColor() function. When you ask for a color, it may be taken from anywhere in the array of 256 entries, and there is NO guarantee that if you allocate a green color that the next color in the table will be a lighter or darker green. Even if you allocate many colors using GetNamedColor() or GetRGBColor(), you have NO assurances about where those colors are in the array (chances are they won't be contiguous). If you need to have a contiguous set of numbers, you must use GetAllColors() and then SetColorMap() or SetMyColorMap() to set up a custom colormap with a known set of values. When you get a private color, your application can specify what values that color index should have. This is useful when you want to interactively modify a color.

It is important to remember that 'getting a color' really means getting an index into the color table where the actual color is stored.

## GetStandardColors()

void GetStandardColors(void);

This function gets 6 standard colors, RED, GREEN, BLUE, YELLOW, BLACK, and WHITE. These 6 variables contain values which can be used in calls to SetColor(), SetFgColor(), SetBgColor(), etc.

Do not use the values in RED, GREEN, BLUE, YELLOW, BLACK or WHITE before calling GetStandardColors(). The results are undefined if you do this.

NOTE: You can only call GetStandardColors() after calling the ShowDisplay() function.

SEE ALSO : GetNamedColor(), GetRGBColor(), GetAllColors() SetColor(), SetFgColor(), SetBg-Color()

## GetNamedColor()

int GetNamedColor(char *name);

This function allocates an entry in the color table for the color given by the ascii string "name". You can view the list of available color names with the showrgb command in a shell (some nice ones are "peachpuff", "burlywood3", "aquamarine", and "paleturquoise3"). Color names can have spaces in them. The return value of the function is an integer that you can use in calls to SetColor() (or any of the other SetXXColor() calls). If an error occurred trying to allocate the color (very possible if you allocate a lot of colors), a -1 is returned.

NOTE: the return value of zero is valid, a -1 indicates an error NOT zero.

NOTE: You can only call GetNamedColor() after calling the ShowDisplay() function.

SEE ALSO : GetStandardColor(), GetRGBColor(), GetAllColors() SetColor(), SetFgColor(), SetBgColor()

## GetRGBColor()

int GetRGBColor(int r, int g, int b);

This function tries to allocate the color given by the red, green, blue triple r,g,b. The arguments r,g, and b should be between 0 and 255. Overflow is not checked for. The return value is an integer value usable in the SetColor() calls or a -1 if an error occurred.

NOTE: the return value of zero is valid, a -1 indicates an error NOT zero.

NOTE: You can only call GetRGBColor() after calling the ShowDisplay() function.

SEE ALSO : GetStandardColor(), GetNamedColor(), GetAllColors() SetColor(), SetFgColor(), SetBgColor()

## GetPrivateColor()

int GetPrivateColor(void);

This function allocates a private color cell for use by the application. A private color cell is one which you can change what color it represents. For example, if you would like to let the user interactively manipulate some color, you would need to allocate a private color cell.

The integer returned by this function is a reference to a color cell whose values you can set with SetPrivateColor(). The intial contents of the private color cell are undefined and you should probably call SetPrivateColor() immediately to set it to some known value.

If an error occurs, a -1 is returned.

When you are done with a private color cell, you should free it with FreePrivateColor().

SEE ALSO: SetPrivateColor(), FreePrivateColor(), GetRGBColor()

## SetPrivateColor()

void SetPrivateColor(int which, int r, int g, int b);

This function sets the color cell referred to by 'which' to have the r,g,b values specified. The r,g,b values are given in the range 0-255 (inclusive). Once this function is called, any thing drawn in the display with the color 'which' will now have the new color determined by the r,g,b arguments.

SEE ALSO: GetPrivateColor(), FreePrivateColor(), SetFgColor(), SetBgColor(),

## FreePrivateColor()

void FreePrivateColor(int which);

This function returns the color associated with the private color cell 'which' to the system. You should have allocated the color referred to by 'which' with GetPrivateColor().

SEE ALSO GetPrivatecolor(), SetPrivatecolor().

## GetAllColors()

int GetAllColors(void);

This function is rather drastic and should be used with caution. It immediately grabs an entire 256 entry colormap for private use. This has the unfortunate effect of (temporarily) wiping out the colors in all the other windows on the display. However this is necessary if you wish to get a smooth colormap to use in displaying a smooth-shaded or continuous tone picture. Once GetAllColors() is called, the entire colormap is free for manipulation by your program. The colormap remains allocated until you call FreeAllColors(), at which time everything goes back to normal.

If an error occurred (quite possible), this routine returns FALSE. If everything went ok and the colormap was successfully allocated, TRUE is returned.

If you can avoid using this function, try to. It is disconcerting for the user to have the colormap get wacked out and have most of their windows disappear (they don't really disappear of course, you just can see them usually). However it is sometimes necessary to do this as there is no other way to get a smoothly continuous color map.

Usually, you will want to call SetColorMap() or SetMyColorMap() right after this function.

NOTE: On a 24 bit machine (like the SGI Indigo Elan I tested this with), only current drawing area gets the colormap, other widgets and windows are not affected.

NOTE: You can only call GetAllColors() after calling the ShowDisplay() function.

SEE ALSO : SetColorMap(), SetMyColorMap(), FreeAllColors(), GetStandardColors(), GetNamed-Color(), GetRGBColor()

## FreeAllColors()

void FreeAllColors(void);

This function frees a private colormap that was allocated with GetAllColors(). It has the beneficial effect of immediately restoring the rest of the colors on the screen and in other windows to those that existed prior to the call to GetAllColors(). This function is useful if wish to let the user restore their original colors temporarily (although this will happen automatically when the mouse moves outside the window).

## SetColorMap()

void SetColorMap(int num);

This function creates several predefined color maps that are very smoothly continuous. It saves you the hassle of writing them yourself (even though they are mostly easy). The "num" argument you pass in should be one of the following #define's :

```
#define GREY_SCALE_1 0
#define GREY_SCALE_2 1
#define RAINBOW_1 2
```

#define RAINBOW_2 3

The colormap GREY_SCALE_2 is a complete smooth color ramp from pure black (color 0) to pure white (color 255). The other grey scale, GREY_SCALE_1 is a nearly pure ramp from black (color 0) to white (color 252), but also has a few additional colors thrown in near the end of the colormap. The two RAINBOW_? colormaps have different types of smooth changing rainbows of color. This are really only useful for drawing pretty patterns or doing false coloring.

NOTE: You should call GetAllColors() before you call this routine. It is not necessary, but if you don't, and GetAllColors() fails, you will never know about it, and your application may not work very well.

SEE ALSO : SetMyColorMap(), GetAllColors(), GetNamedColor(), GetStandardColors(), GetRGBColor()

## SetMyColorMap()

void SetMyColorMap(int n, unsigned char *r, unsigned char *g,unsigned char *b);

Occasionally it is necessary to have absolute control over your colormap, and this function lets you do that. This function lets you completely specify each and every color that will be in the colormap. The three arrays r,g, and b are simply the red, green, and blue components of each color. The values in the array range from 0 to 255, hence they are unsigned char's. You need not specify a full array of 256 colors, you can in fact only specify a few. The integer argument "n" indicates how many entries there are in the r,g, and b arrays. The argument "n" should be greater than 0 and less than 255.

NOTE: You should call GetAllColors() before you call this routine. It is not necessary, but if you don't and GetAllColors() fails, you will never know about it, and your application may not work very well.

SEE ALSO : SetMyColorMap(), GetAllColors(), GetNamedColor(), GetStandardColors(), GetRGBColor()

# Chapter 3

## Drawing areas

A drawing area is a rectangular area that supports drawing into, receiving input from (mouse clicks, mouse motion and keypresses) and redisplay requests from X. You can draw any sort of graphics into a drawing area as well as perform various types of interaction with mouse and keyboard input.

It is also useful to read the next chapter of this document for more information on the actual drawing routines available.

## MakeDrawArea()

Widget MakeDrawArea(int width, int height, RedisplayCB func, void *data);

The function MakeDrawArea() creates a drawing area which you can later use to draw graphics into and receive mouse and keyboard input from. The drawing are will have a width and height as you specify. The callback function, func, is called whenever the drawing area should be redisplayed (because it was obscured or resized). The argument "data" is a pointer to any arbitrary data you want, and it is passed directly to the resize callback (and the other callbacks as well).

The redisplay callback is where you should put all of your drawing code. It is called for you when the application opens the window for the first time (by calling MainLoop()). The redisplay callback function should be declared as follows:

void redisplay(Widget w, int width, int height, void *data)

The first argument, w, is the drawing area widget that needs to be redrawn. The second and third arguments are the new width and height of the drawing area (it may have been resized). The final argument is the void pointer passed to MakeDrawArea().

If you are interested in receiving other types of input, see the functions, SetButtonDownCB(), SetButtonUpCB(), SetKeypressCB() and SetMouseMotionCB(). These functions will let you set callbacks for the other types of input.

Each drawing area you create has its own state (foreground and background colors, drawing mode, and line width). Only one drawing area can be active at any given time. When an event happens for a drawing area, that drawing area becomes the active drawing area. You can make other drawing areas active with SetDrawArea().

If something goes wrong in creating the DrawArea, a NULL value is returned.

SEE ALSO : drawing.doc, SetButtonDownCB(), SetButtonUpCB(), SetKeypressCB(), SetMouse-MotionCB(), SetWidgetPos(), SetWidgetFont(), SetFgColor(), SetBgColor(), SetBorderColor()

## SetButtonDownCB()

void SetButtonDownCB(Widget w, MouseButtonCB func);

This function sets up a callback that will be called everytime the user presses a mouse button in the specified drawing area widget 'w'.

The callback function should be declared as follows:

void func(Widget w, int which_button, int x, int y, void *data)

Then, whenever the user presses a mouse button in the drawing area, your callback is called. The first argument is the drawing area widget where the event happened. The next argument is an integer specifying which button was pressed. It is a small positive integer. A value of one is the left mouse button, two is the middle mouse button and three is the right mouse button. Technically, values of four and five are also possible though I've never seen a mouse with five buttons. The x and y arguments are the position of the mouse where the user pressed the mouse button. The final argument is the void pointer argument given to MakeDrawArea().

You can specify a NULL for the function to turn off receiving button down events.

## SetButtonUpCB()

void SetButtonUpCB(Widget w, MouseButtonCB button_up);

This function sets up a callback that will be called everytime the user releases a mouse button in the specified drawing area widget 'w'.

The callback function should be declared as follows:

    void func(Widget w, int which_button, int x, int y, void *data)

Then, whenever the user releases a mouse button in the drawing area, your callback is called. The first argument is the drawing area widget where the event happened. The next argument is an integer specifying which button was released. It is a small positive integer. A value of one is the left mouse button, two is the middle mouse button and three is the right mouse button. Technically, values of four and five are also possible though I've never seen a mouse with five buttons. The x and y arguments are the position of the mouse where the user released the mouse button. The final argument is the void pointer argument given to MakeDrawArea().

You can specify a NULL for the function to turn off receiving button up events.

## SetKeypressCB()

void SetKeypressCB(Widget w, KeyCB func);

This function lets you set a callback so that you will receive keyboard input in the drawing area.

The callback function should be declared as follows:

    void func(Widget w, char *input, int up_or_down, void *data)

Then, whenever the user presses keys in the drawing area, your callback function is called. The first argument is the drawing area widget where the event happened. The next argument is a character pointer to a null-terminated string that contains what was typed by the user. The up_or_down argument indicates whether the key was pressed released (a zero indicates a press, a 1 indicates a key release). The final argument is the void ponter argument given to MakeDrawArea().

It is useful to know that the string returned to your program is not necessarily a single ASCII character. You will get the usual ASCII characters, including control characters (such as Ĉ or Ĥ). But, the workstation's function keys will also be returned in a string such as "F11" or "F23". You will also get other longer strings such as "Control_L", "Alt_R", or "Shift_L". It is important to understand that even if you just press the shift key to get a capital letter, you will first receive the string "Shift_L" or "Shift_R", then you will receive a capital letter (say, "H"). You should probably ignore the "Shift_L" or "Shift_R" messages (but who knows, you may find some use for them).

The argument, up_or_down, tells you whether the given key was pressed or released. If the key was pressed down, up_or_down has a zero (0), if the key was released, up_or_down contains a 1. This is useful for doing things like shift-clicking with the mouse or handling control-key combinations in an editor or other program.

The arrow keys return strings such as "Left", "Up", "Right", or "Down". Other keys on the keyboard may return strings such as "Home", "Prior", "Next", "Undo", "Help", etc. Of course not all keyboards generate all of the strings (because they aren't set up to).

NOTE WELL: The string that is returned to you can NOT (I'll repeat that, can NOT) be modified by your program. Got it? Do NOT modify the string. If you want to munge with it, make a copy using strdup() or strcpy() into your own buffer space.

You can specify a NULL for the function to turn off receiving keypress events.

## SetMouseMotionCB()

void SetMouseMotionCB(Widget w, MotionCB func);

This function sets a callback so that whenever the mouse moves in your drawing area, the specified function will be called. It is important to keep in mind that the function you specify is called every time the mouse moves in the drawing area, even if it is just passing through.

The callback function should be declared as follows:

```
void func(Widget w, int x, int y, void *data);
```

The first argument is (as usual) the Widget where the mouse was moved in. The next two arguments are the current X and Y values of the mouse. The final argument is the void pointer passed into MakeDrawArea().

You should be very frugal with what happens in this function so as not to cause the application to lag behind the user too much. Calling functions like sleep() are definitely out of the question.

You can specify a NULL for the function to turn off receiving mouse motion events.

# Chapter 4

## Drawing

This chapter contains documentation about the routines that let you draw in a drawing area.

The documentation for these functions is quite brief because they are not all that complicated (how much can one really say about DrawLine()).

Keep in mind that for all the drawing functions, the top-left corner of a drawing area is considered to be 0,0.

Also, all primitives are drawn in the current foreground color (set either by SetColor() or SetFgColor(). Text is drawn with the current foreground color and the background color. Line, arc, and box primitives are drawn with the current line width (as set by SetLineWidth()), and all primitives are drawn in the current draw mode (set by SetDrawMode()).

### SetDrawArea()

void SetDrawArea(Widget w);

This sets the current drawing area to be that named by the Widget w. If 'w' is not a drawing area widget, nothing happens.

You need to call this function when you want to switch between multiple drawing areas.

If you only have one drawing area you do not need to worry about this function at all.

Any callbacks for a drawing area already have the current drawing area set to be the one where the event happened (so it is not necessary to call this function in a callback for a drawing area).

### SetColor()

void SetColor(int color);

This sets the foreground color to draw with in the current drawing area (each drawing area has its own foreground color). The argument "color" should be a valid color obtained with one of the color functions (such as GetNamedColor() or GetRGBColor(), etc).

To some extent this function duplicates the SetFgColor() function, but exists because it is faster than SetFgColor().

SEE ALSO : SetBgColor(), GetFgColor(), SetFgColor(), GetStadardColors()

### SetLineWidth()

void SetLineWidth(int width);

This functions sets the width of lines drawn in the current drawing area. Each drawing area has its own line width.

A width of zero is valid and tells the X server to draw lines as fast as it possibly can, possibly being a little inaccurate. Larger numbers of course draw wider lines.

SEE ALSO : SetDrawMode(), SetWidgetFont(), SetFgColor(), SetBgColor()

## SetDrawMode()

void SetDrawMode(int mode);

This function sets the drawing mode for the current drawing area. A drawing mode is one of:

GXcopy, GXxor, GXinvert, GXor, GXclear, GXand, GXandReverse, GXnoop, GXnor, GXequiv, GXinvert, GXorReverse, GXcopyInverted, GXorInverted, GXnand, and GXset

Most of these are stupid/useless modes defined by X (so ignore them).

The primary mode is GXcopy (the default mode). This causes all primitives to draw in the foreground color, overwriting any pixels already drawn.

Libsx also defines a special mode: SANE_XOR. The SANE_XOR mode will actually draw primitives in a true XOR mode so that you can draw things like rubber-band boxes that the user stretches with the mouse. You must use SANE_XOR if you want true XOR'ed primitives, GXxor will definitely NOT work as you expect.

When you are done using SANE_XOR, you would normally call SetDrawMode() with an argument of GXcopy to restore normal drawing.

SEE ALSO : SetLineWidth(), SetWidgetFont(), SetFgColor(), SetBgColor()

## GetDrawAreaSize()

void GetDrawAreaSize(int *w, int *h);

This is a convience function that returns the size of the current drawing area. The window dimension are returned in the two variables. It is important to note that "w" and "h" are POINTERS to integers, not just regular integers.

## ClearDisplay()

void ClearDisplay(void);

This function completely clears the current drawing area and sets it to the current background color (which may not be white).

Generally, when your redisplay callback is called, this is the first thing want to do.

SEE ALSO : SetDrawArea(), GetBgColor(), SetBgColor()

## DrawPixel()

void DrawPixel(int x1, int y1);

This function draws a point in the current foreground color at the location x1, y1 in your current drawing area. The top left corner of the drawing area is considered 0,0.

SEE ALSO : GetPixel(), SetColor(), SetDrawArea(), SetBgColor(), SetFgColor()

## GetPixel()

int GetPixel(int x1, int y1);

This function retrieves the pixel value at the location x1, y1 in the current drawing area. The top left corner of the drawing area is considered 0,0.

The pixel value returned to you will be between 0 and 255 (inclusive). The value you get back should

be treated as an index to a colormap. To find out what actual color is displayed at that location, you need to look up the color in the colormap (which you should be maintaining as there is no way to get it after you've set it).

NOTE: This function is NOT very high performance. It has to call GetImage() to do the bulk of the work. This is unfortunate, but unavoidable because X does not provide an easy way to read individual pixels.

SEE ALSO : GetImage(), SetColor(), SetDrawArea(), SetBgColor(), SetFgColor()

## DrawLine()

void DrawLine(int x1, int y1, int x2, int y2);

This function draws a line from x1,y1 to x2,y2 in the current foreground color in the current drawing area. The top left corner of the drawing area is considered 0,0.

SEE ALSO : DrawPolyline(), DrawPixel(), SetColor(), SetBgColor(), SetFgColor(), SetDrawArea(),

## DrawPolyline()

void DrawPolyline(XPoint *points, int n);

This function accepts an array of points and draws them as a connected polyline on the display. The line is drawn in the current foreground color in the current drawing area. The top left corner of the drawing area is considered 0,0.

The 'points' argument is an array of XPoint structures which are as follows:

typedef struct XPoint  short x,y; XPoint;

You do not need to define this structure yourself. It is defined for you already, it is just reprinted here so you can see what it is.

You should have an array of these structures with each entry holding a vertex of the polyline.

SEE ALSO : DrawLine(), DrawFilledPolygon(), SetColor(), SetDrawArea(), SetBgColor(), SetFgColor()

## DrawFilledPolygon ()

void DrawFilledPolygon (XPoint *points, int n);

This function takes an array of points and draws them as a filled polygon on the display. The polygon is filled with the current foreground color and is drawn in the current drawing area. The top left corner of the drawing area is considered 0,0.

The 'points' argument is an array of XPoint structures which are as follows:

typedef struct XPoint  short x,y; XPoint;

You do not need to define this structure yourself. It is defined for you already, it is just reprinted here so you can see what it is.

You should have an array of these structures with each entry holding a vertex of the polygon to be filled.

SEE ALSO : DrawPolyline(), DrawBox(), SetColor(), SetDrawArea(), SetBgColor(), SetFgColor()

## DrawBox()

void DrawBox(int x, int y, int width, int height);

This function draws a rectangular box starting at x,y with a width and height as specified. If you make the call: DrawBox(50,50, 75,75), you will get a box that starts at position 50,50 and goes for 75 pixels in the X and Y directions (i.e the other extreme of the box would be at 125,125). The box is drawn in the current foreground color in the current drawing area. The top left corner of the drawing area is considered 0,0.

If the width and height are negative, the box is still drawn properly.

SEE ALSO : DrawFilledBox(), DrawPolygon(), SetColor(), SetDrawArea(), SetBgColor(), SetFgColor()

## DrawFilledBox()

void DrawFilledBox(int x, int y, int width, int height);

This function draws a filled rectangular box starting at x,y with a width and height as specified. If you make the call: DrawFilledBox(50,50, 75,75), you will get a filled box that starts at position 50,50 and goes for 75 pixels in the X and Y directions (i.e the other extreme of the box would be at 125,125). The box is filled with the current foreground color in the current drawing area. The top left corner of the drawing area is considered 0,0.

If the width and height are negative, the box is still drawn properly.

SEE ALSO : DrawBox(), DrawFilledPolygon(), SetColor(), SetDrawArea(), SetBgColor(), SetFgColor()

## DrawText()

void DrawText(char *string, int x, int y);

This function prints the text string "string" starting at x,y. The text is drawn in the current foreground color. The background of the text is filled with current background color of the drawing area widget. The top left of the drawing area is 0,0. The X,Y position you specify is the bottom left corner of where the text is drawn.

SEE ALSO : GetFont(), FontHeight(), TextWidth(), SetColor(), SetDrawArea(), SetWidgetFont(), GetBgColor(), SetFgColor(), SetBgColor()

## DrawArc()

void DrawArc(int x, int y, int width, int height, int angle1, int angle2);

This function draws an arc/ellipse from the location x,y that is bounded by the box defined by the x,y, width and height. That is, the arc/ellipse will always be contained in the box defined by the x,y position and the width and height arguments. The X,Y arguments are not the center of the arc/circle.

The arc begines at angle1 degrees and continues for angle2 degrees around the circle. The arc is drawn in the current foreground color in the current drawing area. The top left corner of the drawing area is considered 0,0.

If you want a circle, you would specify angle1 as 0, and angle2 as 360.

If the width and height are negative, the arc is still drawn properly.

SEE ALSO : DrawPolyline(), SetColor(), SetDrawArea(), SetBgColor(), SetFgColor()

## DrawFilledArc()

void DrawFilledArc(int x, int y, int width, int height, int angle1, int angle2);

This function draws a filled arc/ellipse from the location x,y that is bounded by the box defined by the x,y, width and height. That is, the arc/ellipse will always be contained in the box defined by the x,y position and the width and height arguments. The X,Y arguments are not the center of the arc/circle.

The arc begines at angle1 degrees and continues for angle2 degrees around the circle. The arc is filled with the current foreground color in the current drawing area. The top left corner of the drawing area is considered 0,0.

If you want a circle, you would specify angle1 as 0, and angle2 as 360.

If the width and height are negative, the arc is still drawn properly.

SEE ALSO : DrawArc(), SetColor(), SetDrawArea(), SetBgColor(), SetFgColor()

## DrawImage()

void DrawImage(char *data, int x, int y, int width, int height);

This function draws a bitmap image that has a width and height as specified by the arguments. The image is drawn at location x,y in the current drawing area. The "data" argument should point to at least width*height bytes of data.

Each byte of the data is interpreted as a color value to draw the corresponding pixel with.

Normally you would use this function when you have taken over the colormap with GetAllColors() (so that you can be guaranteed certain colors are in a given range). If you have not taken over the colormap, you need to make sure that the bytes in the image data contain valid values that you've allocated with the color allocation functions (GetNamedColor(), GetRGBColor() or GetPrivateColor()).

The top left corner of the drawing area is considered 0,0.

SEE ALSO : GetImage(), SetColor(), SetDrawArea(), GetAllColors()

## GetImage()

void GetImage(char *data, int x, int y, int width, int height);

This function retrieves a bitmap image from your drawing area that has a width and height as specified by the arguments. The image is taken from the starting location x,y in the current drawing area. The "data" argument should point to at least width*height bytes of data.

The area of memory pointed to by data will be filled with the 8-bit pixel values of the current drawing area. Note that the pixel values are not the actual color values. If you want the actual color values, you'll need to know what the current colormap is (which you would know if you've set the colormap) and then use the pixel values to index the colormap.

The memory pointed to by data is packed with width*height bytes, with no extra padding or filling. That is, the first width bytes correspond to line 0, the next width bytes correspond to line 1 of the image, etc.

It is important to keep in mind that if you plan to save the pixel data in an image file, you need to also keep track of the colormap so that you can save that as well. By themselves, the pixel values don't correspond to any particular color.

A serious drawback of this function arises from the way X operates. If the drawing area from which you are "getting" the image is obscured by another window, that part of the bitmap will be empty. The only way around this is to make sure that your window is in front of all the others before you call GetImage(). This is a serious limitation, but it's the way X operates.

The top left corner of the drawing area is considered 0,0. If you specify a starting x,y and width,height dimensions that are larger than your drawing area, you will get a BadMatch error and X will terminate your program (so be careful).

SEE ALSO : DrawImage(), SetColor(), SetDrawArea(), GetAllColors()

## ScrollDrawArea()

void ScrollDrawArea(int dx, int dy, int x1,int y1, int x2, int y2);

This function scrolls the box defined by (x1,y1) (x2,y2) by the amounts dx and dy in the X and Y directions respectively. This means that the box whose upper left corner is (x1,y1) and whose lower right corner is (x2,y2) are scrolled by dx and dy pixels in X and Y.

A positive value for dx causes the drawing area to scroll its contents to the left. That is, whatever is at the left edge gets pushed off and the dx columns of pixels on the right hand side are cleared to the background color. A negative value has the opposite effect.

A positive value for dy corresponds to scrolling upwards. That is, whatever is at the top of the drawing area is pushed up by dy pixels and the bottom dy rows of pixels are cleared to the background color. A negative value has the opposite effect.

This function is useful for scrolling the drawing area to draw new information (such as a text editor might do to scroll text up or down).

The new area exposed by the scroll is filled with the current background color of the drawing area.

SEE ALSO : SetWidgetFont(), SetColor()

# Chapter 5

## Fonts

Fonts are different type styles that can be used in various widgets. You load a font by name and get a handle in return. The handle you get allows you to set the font in the various widgets and font information calls.

### GetFont()

XFont GetFont(char *fontname);

This function loads the font named by fontname and returns a handle to the font or NULL if there is an error. The handle returned by this function is an XFontStruct pointer, but should be treated as an opaque data type.

After you've loaded a font, you can then set that font in any widget that displays text. You can also use the handle in calls to TextWidget() and FontHeight().

When you are done with a font, you should free it with FreeFont().

SEE ALSO: SetWidgetFont(), FreeFont(), GetWidgetFont(), FontHeight(), TextWidth()

### SetWidgetFont()

void SetWidgetFont(Widget w, XFont f);

This functions sets the font used by the widget 'w', to be the font referred to by the argument 'f'. The argument f should have been obtained with GetFont().

SEE ALSO: GetFont(), FreeFont(), FontHeight(), TextWidth()

### GetWidgetFont()

XFont GetWidgetFont(Widget w);

This function returns a handle to the font currently used by the Widget w. If an error occurs or there is no default font for the widget, a NULL is returned.

You should NOT call FreeFont() on any value returned by this function unless you are sure that you allocated the font with GetFont().

SEE ALSO: GetFont(), SetWidgetFont(), FontHeight(), TextWidth()

### FreeFont()

void FreeFont(XFont f);

This function frees the resources associated with the font, f. You should call this function when your application is done using a particular font.

Of course you should make sure that no widget still uses the identified font.

SEE ALSO: SetWidgetFont(), GetFont()

## FontHeight()

int FontHeight(XFont f);

This function returns an integer value that is the height in pixels of the specified font. The height is defined to be the ascent of the characters (from the baseline to the top of a capital letter) plus the descent of the characters (the distance from the baseline to bottom of a descender character like 'g' or 'p').

SEE ALSO: TextWidth(), GetFont(), SetWidgetFont()

## TextWidth()

int TextWidth(XFont f, char *txt);

This functions returns the width in pixels used by the string pointed to by txt in the font f. The string should be null-terminated and the entire string is used to determine the width.

SEE ALSO: FontHeight(), GetFont(), GetWidgetFont()

# Chapter 6

## Forms

A form widget is a container that holds other widgets.

## MakeForm()

Widget MakeForm(Widget parent, int where1, Widget from1 int where2, Widget from2)

This function lets you create a new "form" widget in which to put child widgets. A form widget is a container that holds other widgets. Normally there is no need to call this function, but if you want to have separate "groups" of widgets in your display and you can't lay them out that way with SetWidgetPos(), then using multiple form widgets may be the right thing. In addition, a nifty little box gets drawn around the form widget (and all the children) and this can be a nice visual cue in your interface indicating what groups of widgets belong together. A form widget creates a box that surrounds all the widgets contained inside of it (but the form widget itself is inactive and can't be clicked on by the user).

If you use multiple form widgets in your display, the basic logic of how you create the display is a little different. You can think of form widgets as miniature windows inside a larger window.

Once you create a form widget, any other widgets you create with calls like MakeButton() and MakeLabel() become children of this form widget. Before you create another form widget, you must lay out all the children of the current form widget with calls to SetWidgetPos(). After you lay out all the children of the current widget, then you can create another form widget, and repeat the process (or call SetForm()).

Form widgets are layed out in a manner similar to regular widgets, except that usually their placement is relative to other form widgets. When you create a new form widget (after the first one), you specify where it should be placed relative to other form widgets that you created. The first form widget is always placed in the top left corner of the window.

The 'parent' argument to MakeForm() specifies at what level the new form should be created. If you specify TOP_LEVEL_FORM (which is the usual thing to do) the new form is created at the top level of the window. If you pass another form widget for 'parent', then this new form widget will be a child of the other form widget. This lets you create hierarchical "boxes" in your display.

The arguments where1, from1, where2, from2 are the same as in SetWidgetPos(). That is, you specify either NO_CARE, PLACE_UNDER, or PLACE_RIGHT for where1 and where2 and the from1/from2 arguments are the widgets you would like to place something to the right of or under (or they are NULL if you specified NO_CARE). See SetWidgetPos() for more documentation.

Now for an example:

Let's say we want to make a drawing area and a 'palette', i.e. a child window with - say - two buttons on it, to the left of the drawing area.

We have two rectangles (forms) which contain other widgets. Inside the leftmost form are two buttons. The form on the right has a single drawing area. Skipping some of the unnecessary details, we could accomplish the above display with the following code:

```
form1 = MakeForm(TOP_LEVEL_FORM, NO_CARE, NULL, NO_CARE, NULL);
w[0] = MakeButton("Btn1", NULL, NULL);
w[1] = MakeButton("Btn2", NULL, NULL);
```

```
SetWidgetPos(w[1], PLACE_UNDER, w[0], NO_CARE, NULL);


form2 = MakeForm(TOP_LEVEL_FORM, PLACE_RIGHT, form1, NO_CARE, NULL);
w[2] = MakeDrawArea(200, 200, NULL, NULL);
```

As you can see, we create the first form and specify that we don't care where it goes (the first form widget is always placed in the top left corner of the window). Then we create some widgets to place inside of our new form. Next, and this is important, we layout all the widgets inside of the first form. In this case we only need to make one call to SetWidgetPos(). Then we create the next form, and specify that we want to place it to the right of form1. Finally we create a drawing area widget, which is placed inside of form2.

If you want to create hiearchies of form widgets, you would specify the form widget that should be the parent for the first argument to MakeForm(). This can get quite complicted, so you should make sure you know what you're doing if you want to create big hierarchies.

NOTE: It is important that you layout all your widgets before you create a new form (unless you're creating a child form).

SEE ALSO: SetForm(), MakeWindow()

## SetForm()

void SetForm(Widget w)

The SetForm() function allows you to change what is considered the current form. Normally you only use this function to set the current form to be TOP_LEVEL_FORM. You can cause your program to crash if you are not careful about what you set as the current form.

The main purpose of this function is to let you create displays that have both form widgets and other "normal" widgets at the same level. Mainly you would want to do this if you wanted a large drawing area (or some other type of widget) but didn't want to bother creating an form widget just to hold that one widget.

After calling this function, you can position any new widgets relative to other widgets (usually form widgets) created at the top level of the window.

The normal calling sequence is: SetForm(TOP_LEVEL_FORM), although you can specify any other form widget you like. Be careful, as it is possible to confuse the X layout routines and cause your program to crash.

NOTE: Before you call SetForm() and start creating new widgets and positioning them, any previous form widgets should be completely layed out (i.e. you called SetWidgetPos() for all child widgets of any previously created form widgets).

SEE ALSO: MakeForm()

# Chapter 7

## Labels

A label widget is a widget that displays some text or bitmap but can not be clicked on or interacted with by the user. Generally a label is for informational purposes, such as saying what the current file name is.

## MakeLabel()

Widget MakeLabel(char *txt);

This function creates a label that contains the text in the character string pointed to by "txt". The text will simply be displayed, with no fancy borders or special highlighting. If the text contains new line characters, they will be interpreted properly.

If the argument txt is NULL, then no label will be set for the widget. This is convienent if you plan to put a bitmap on this widget with the SetWidgetBitmap() function.

This widget is useful for displaying a piece of textual information like a filename or user name.

If this routine fails, a NULL is returned.

SEE ALSO : SetWidgetPos(), SetWidgetBitmap(), SetLabel(), SetWidgetFont(), SetFgColor(), SetBgColor(), SetBorderColor()

# Chapter 8

## Lists

A scroll list is a scrollable list of items organized in a veritical fashion. The user can scroll through the list of items using the mouse and select individual items in the list of available choices.

## MakeScrollList()

Widget MakeScrollList(char **item_list, int width, int height, ListCB func, void *data);

This function makes a scrollable list of items from which a user can select individual items. The list contains strings of text, pointed to by the table, item_list. The list, item_list, MUST contain a NULL entry as its last element (this is not optional). The area given to display the list is width and height pixels large. If the entire list will not fit, scrollbars appear that will let the user easily scroll through the list.

The callback function, func, should expect to be called with a Widget argument, the string of text the user clicked on, the string's index in your table, and whatever user data pointer you gave at widget creation time. The callback should be declared as follows:

void func(Widget w, char *str, int index, void *data)

The list of strings passed in to MakeScrollList() must not be free()'d or otherwise deallocated for the entire lifetime of the widget (or until the list of strings is changed with ChangeScrollList()).

SEE ALSO: GetCurrentListItem(), ChangeScrollList(), SetCurrentListItem(), SetFgColor(), SetBgColor(), SetWidgetFont()

## GetCurrentListItem()

int GetCurrentListItem(Widget w);

This function returns the index of the currently selected item in the list widget 'w'. The index value returned is an index into the table displayed by the list (specified when the widget was created or with ChangeScrollList()).

If no item is selected in the list widget, this routine will return a -1.

SEE ALSO: ChangeScrollList(), SetCurrentListItem(), MakeScrollList()

## ChangeScrollList()

void ChangeScrollList(Widget w, char **new_list);

This function changes the list of strings displayed by the list widget 'w'. The new list of items is taken from the argument 'new_list'. After this function is called, the old list can be free()'d. Of course you can not free the new_list until the application is done or you change the list again.

You must remember to make the last entry of the new_list be NULL. This is very important.

SEE ALSO: GetCurrentListItem(), SetCurrentListItem(), MakeScrollList()

## SetCurrentListItem()

void SetCurrentListItem(Widget w, int list_index);

This function causes the item with index, 'list_index', to be highlighted in the list widget 'w'. You must make sure that list_index is a valid index into the currently displayed list, results are undefined otherwise.

After calling this function, the item with the index 'list_index' is highlighted in the list widget.

SEE ALSO: GetCurrentListItem(), ChangeScrollList(), MakeScrollList()

# Chapter 9

## Menus

Menus provide standard drop-down style menus that let the user select from a variety of choices. The Athena widgets do not support cascaded menus, so a menu is only a single list of items. A menu contains menu items that are tied to callback functions in the application. Menu items must be text and can not be bitmaps.

## MakeMenu()

Widget MakeMenu(char *name);

This function creates a menu button that contains the text pointed to by the character string name. When the button is clicked, a menu pops up. The menu contains items created with MakeMenuItem().

You need to save the return value of this function to be able to pass it to MakeMenuItem() so that menu items can attached to a menu.

If this function fails, it returns NULL.

SEE ALSO: MakeMenuItem(), MakeButton(), SetWidgetPos()

## MakeMenuItem()

Widget MakeMenuItem(Widget menu, char *name, ButtonCB func, void *arg);

This function adds a menu item to a menu. The menu item contains the text pointed to by the string name. Whenever the user selects this menu item, the callback function, func, is called. The final argument is an arbitrary void pointer that is passed to the callback function.

The first argument must be a widget returned by MakeMenu() (results are undefined if it is not).

If MakeMenuItem() fails for any reason, a NULL is returned.

The callback function for the menu item should be declared as follows:

    void func(Widget w, void *data)

Whenever the user selects this menu item, the callback will be called.

Setting of widget attributes with SetFgColor(), SetBgColor(), etc work normally except that only one background color may be specified and it takes effect for the entire menu. You can set different fonts for each menu item.

NOTE: You do not need to call SetWidgetPos() for menu items. Successive menu items are placed below previous menu items.

SEE ALSO: SetMenuItemChecked(), GetMenuItemChecked(), MakeMenu().

## SetMenuItemChecked()

void SetMenuItemChecked(Widget w, int state);

This function sets the "checked" state of a menu item. If a menu item is in the checked state, a

bitmap of a check mark appears to the left of the menu item text.

The first argument, w, is a menu item widget created with MakeMenuItem(). The second argument, state, is a boolean value of either TRUE (1) or FALSE (0) indicating whether or not the check mark should be drawn to the left of the menu item. If the state argument is TRUE, the check mark is drawn. If the state argument is FALSE, the check mark is removed.

SEE ALSO: GetMenuItemChecked(), MakeMenuItem()

## GetMenuItemChecked()

int GetMenuItemChecked(Widget w);

This function returns a boolean result indicating whether the menu item referred to by the Widget w, is checked or not.

If the menu item referred to by 'w' is checked, a value of TRUE (1) is returned. If the menu item does not currently have a check mark next to it, a value of FALSE (0) is returned.

SEE ALSO: SetMenuItemChecked(), MakeMenuItem()

# Chapter 10

## Popups

Popup windows are simple dialog boxes that get a simple yes/no or string answer from the user. When these windows popup, they block input to the previously active window.

### GetYesNo()

int GetYesNo(char *question);

This function allows you to prompt the user for the answer to a simple yes or no type of question. It simply pops up a dialog box with the text contained in the string question, and two okay/cancel buttons.

If the user clicks Okay, this function returns TRUE. If the user clicks Cancel, this function returns FALSE. The text in the question string can have embedded newlines to break things up or to add spacing.

SEE ALSO : GetString()

### GetString()

char *GetString(char *msg, char *default);

This function allows you to prompt the user for a string of input. The first argument, msg, is a pointer to a string which will be displayed in the dialog box. The next argument, default, is the default string to place in the text string box (it can be NULL or "").

When you call this function, it pops up a small dialog box on the screen, and the user can enter the line of text. When the user clicks ok or cancel, the function returns a pointer to the string of text the user entered.

If the user clicks cancel, you get a NULL return value.

SEE ALSO : GetYesNo()

# Chapter 11

## Scrollbars

A scrollbar is a widget that allows a user to control some numeric quantity interactively by using the mouse to scroll a slider, similar to the volume slider on some radios. slider is called the "thumb" or "thumb area" in X terminoligy.

## MakeHorizScrollbar()
## MakeVertScrollbar()

Widget MakeHorizScrollbar(int length, ScrollCB scroll_func, void *data);
Widget MakeVertScrollbar(int height, ScrollCB scroll_func, void *data);

These two routines create scrollbars. Scrollbars allow the user to interactively control some numeric quantity with the mouse.

When the user presses the left mouse button, the value represented by the scrollbar increases. When the press the middle mouse button, they can interactively adjust the value. Clicking the right mouse button decreases the value represented by the scrollbar.

The arguments to create a scrollbar are its length or height in pixels, a callback function to call when the scrollbar changes value and an extra void pointer argument that is passed to the callback function.

If these routines fail, they return NULL.

To set what values a scrollbar represents, you must use SetScrollbar(). These two routines only make a scrollbar and do not set it to return useful values. You always have to call SetScrollbar() to set what a scrollbar represents (see the documentation for SetScrollbar() for more information).

Your callback routine is called everytime the scrollbar changes. Since the calculations are done in floating point, the value may not have changed enough to be interesting, but your routine is still called. You should take care to see that the value changed enough to be interesting to your applications (i.e. it is wasteful for a text editor to repaint the window when the new value is 0.003 different than the old value).

A scrollbar callback routine should be declared as follows:

```
void scroll_func(Widget w, float new_val, void *data)
```

The first argument, w, is the scrollbar widget that the user is manipulating. The second argument is a floating point value that represents the new value of the scrollbar. The third argument is the void pointer argument that was passed to MakeHoriz,VertScrollbar().

SEE ALSO : SetScrollbar(), SetWidgetPos(), SetFgColor(), SetBgColor(), SetBorderColor()

## SetScrollbar()

void SetScrollbar(Widget w, float where, float max, float size_shown);

This function lets you set the values that a scrollbar represents. The first argument is a scrollbar widget. The next three arguments are floating point numbers that specify the parameters of the scrollbar.

Before discussing the details of the three float arguments, let us get some terms straight. When we refer to the 'container' of a scrollbar, we mean the entire box that makes up the scrollbar. The 'thumb' of a scrollbar is the gray area that the user can grab to manipulate. We refer to the size or length of the thumb area (the amount of grey) as the 'size shown'. The total amount represented by the scrollbar is called 'max'.

The arguments mean the following:

where:
this floating point number specifies where in the container the top of the thumb area should start. If you have a maximum value of 100 and where is 50, the beginning of the thumb will be in the middle of the container.

max:
The maximum value that the scroll bar can have. You will receive numbers in the range 0 to max (inclusive). Obviously max should be a positive, and is a float.

size:
This float value controls how big the grey area that the user can grab is. This represents how much of hatever it is you are representing is visible. For example, a text editor which shows 24 lines of text would set size_shown to be 24 (out of whatever max is). If you want the scrollbar to represent a percentage of something, where when it is 100grey area is also 100set the size_shown to be equal to max. If this is confusing, there are examples below.

Now, some examples to clarify things (in the following, assume that the argument, w, represents a scrollbar widget created with the MakeHorizScrollbar or MakeVertScrollbar() routines).

For the first example, let's assume you want a scrollbar to let you set a color value which can range between 0 and 255 with the initial value at 67. You sould set the scrollbar as follows:

```
SetScrollbar(w, 67.0, 255.0, 1.0);
```

The first value, 67.0, is where we would like the beginning of the thumb area to appear. The next value, 255, is the maximum value the scrollbar can attain. The third value, 1, is the size of the thumb area (the amount represented by the thumb relative to the maximum size). This scrollbar will now return values in the range of 0 to 255 (inclusive). The thumb area will be small, and represents one value of the 255 possible divisions. The position of the thumb area in the container represents its value.

For the next example, suppose we wish to make a scrollbar represent some percentage of a value. That is, the size of the thumb area should be proportional to the value of the scrollbar relative to its maximum (so when the value is at it maximum, the thumb area is 100the scrollbar).

In this case we would like the size of the thumb area to represent the amount of the variable (note the difference from the above example). Let us suppose we want a scrollbar which can represent a percentage 0 to 100, with the initial value being 50.

```
SetScrollbar(w, 50.0, 100.0, 100.0);
```

The first value, 50, is where the thumb area begins (in the middle of the container). The next number is 100, and represents the maximum value of the scrollbar. The next number, again 100, is the size shown. Making this value the same as the max value (in this case 100) causes the thumb area to vary according to the value the scrollbar represents.

As a final example, let us take a text editor which is displaying a file. In this case, let us assume the text file is 259 lines long, the window can display 47 lines, and the top line currently displayed is 72. To create the correct scrollbar, we would do the following:

```
SetScrollbar(w, 72.0, 259.0, 47.0);
```

This creates a scrollbar which has a thumb area whose size is 47/259 of the entire container, and is positioned 72/259'ths of the way down the container.

SEE ALSO: MakeHorizScrollbar(), MakeVertScrollbar(), SetWidgetPos()

# Chapter 12

## String entry widgets

A string entry widget is a widget that lets a user enter a single line of ASCII text. When the user presses return in the widget, a callback is made to your application with a pointer to the new text. Support routines also exist to Set and Get the text in the widget.

If you want multiple lines of text, see the text edit widget documentation.

## MakeStringEntry()

Widget MakeStringEntry(char *txt, int size, StringCB func, void *data);

This function makes a string input widget. A string input widget is a widget that lets a user enter/edit a single line string value such as a filename.

The first argument is any default text you would like in the string entry widget. You can specify NULL or "" if you don't want anything to appear.

The next argument is the width in pixels of the string entry area. Be careful in specifying the width since the default font used by the widget may not be wide enough to contain the text you want. It is best if you call GetWidgetFont() and then call TextWidth() on a string of reasonable length and use the value returned by TextWidth() to be the width of the widget. If you're lazy, a value of 150-200 is usually pretty good.

The next argument is a callback function that is called whenever the user presses return in the string entry widget. The callback function should be declared as follows:

```
void func(Widget w, char *string, void *data)
```

The first argument to the callback is the widget where the user pressed return. For the most part you can ignore this (unless you want to change the text). The second argument is a pointer to the string of text the user entered. The final argument is the user data pointer you passed in to MakeStringEntry().

The string of text passed to your callback function should be copied elsewhere (using strdup() if necessary) because it is internal to the widget. The string passed to your callback function should never be modified directly.

SEE ALSO : SetStringEntry(), GetStringEntry(), SetWidgetPos(), GetWidgetFont(), TextWidth(), SetWidgetFont(), SetFgColor(), SetBgColor(), SetBorderColor()

## SetStringEntry()

void SetStringEntry(Widget w, char *new_text);

This function allows you to change the string of text displayed in a string entry widget.

The first argument is the widget in which you would like to change the string (this widget should be a string entry widget). The second argument is a pointer to the new text you would like displayed in the string entry area.

After calling this function, the new text is displayed in the string entry area and any old strings are gone.

SEE ALSO : GetStringEntry(), MakeStringEntry(),

## GetStringEntry()

char *GetStringEntry(Widget w)

This function lets you retrieve the text a user entered in a string widget. The widget argument, w, should be a string entry widget.

The return value of this function is a char pointer to a null-terminated string that is the contents of the string entry widget.

If there is a problem, the function returns NULL.

NOTE: You should not free the string returned to you by this function. If you need to modify the string or otherwise use, you should make a copy with strdup() or some other method.

SEE ALSO : SetStringEntry(), MakeStringEntry(),

# Chapter 14

## Text edit widgets

A text edit widget is an area used to display and optionally edit multiple lines of text. You can specify the text to be displayed as either an in memory string or as a file name. The text edit area manages its scrollbars and internal affairs, you need not do anything (in fact there are no callbacks for the text edit widget).

### MakeTextWidget()

Widget MakeTextWidget(char *txt, int is_file, int editable, int w, int h);

This functions lets you create a text edit widget that will display some text and optionally let the user edit and manipulate it. The first argument, txt, is a pointer to a string (NULL is ok). The second argument, is_file, is a boolean value indicating if the first argument should be interpreted as a file name or not. The next argument, editable, is a boolean value indicating whether or not the user is allowed to edit the text. The final two arguments specify the width and height of the drawing area box. If the area is too small to display the text, scrollbars will appear.

The txt argument can either contain the entire string (null terminated) that you would like the user to edit, or it can contain the name of a file to be loaded into the text edit widget. If the second argument is_file is TRUE (1), then the first argument gets interpreted as a file name. If is_file is FALSE (0), then the first argument contains the actual text to be displayed.

If the txt argument contains the actual text to be displayed, after calling MakeTextWidget() you can free the memory if necessary (the text edit widget makes an internal copy of the string).

The argument 'editable' is a boolean value indicating whether or not to allow editing of the text in the widget. If you just wish to display some text (such as a help file), set the editable argument to FALSE (0) and the user will not be allowed to modify the text.

SEE ALSO: Set/GetTextWidgetText(), SetWidgetPos(), SetFgColor(), SetBgColor(), SetWidget-Font()

### SetTextWidgetText()

void SetTextWidgetText(Widget w, char *text, int is_file);

This argument lets you modify the text displayed in a text edit widget. The first argument identifies the text edit widget to change. The second argument is a null-terminated string that either contains the actual text to display or the name of a file to read in. If the is_file argument is TRUE (1), then the string pointed to by the argument, text, is interpreted as a file name. If is_file is FALSE, the string pointed to by text is directly displayed in the text edit widget.

After calling this function, you can free the string pointed to by text if necessary. The text edit widget makes an internal copy of the text. If you wish to update the displayed text again, you should call SetTextWidgetText() again.

BUGS: The function name is way too long.

SEE ALSO: GetTextWidgetText(), MakeTextWidget()

## GetTextWidgetText()

char *GetTextWidgetText(Widget w);

This function lets you retrieve the text contained in a text edit widget. The only argument, w, should be a text edit widget created with MakeTextWidget().

The return from this function is a character pointer to a null-terminated string that contains the current text in the widget. If there is an error, a NULL is returned.

NOTE: You should not free or otherwise modify the text returned by this function. If you need to make modifications, make a copy of the buffer. Again, DO NOT MODIFY THE TEXT RETURNED BY THIS FUNCTION. Make a copy if you need to modify it.

BUGS: The function name is way too long.

SEE ALSO: SetTextWidgetText(), MakeTextWidget()

# Chapter 15

## Toggle widgets

A toggle widget is similar to a button widget except that it maintains state. That is, when a user clicks a toggle widget, it remains highlighted until it is clicked again. This is similar to an on/off switch.

Toggle widgets can also be used to create a group of "radio buttons". A radio button group is a set of toggle widgets in which at most one of them can be selected at any one time (it is possible for none of them to be selected).

## MakeToggle()

Widget MakeToggle(char *txt, int state, Widget w, ToggleCB func, void *data);

This function makes a widget that will toggle between a highlighted 'on' state and an unhighlighted 'off' state.

The first argument is the text that will be displayed inside the widget. The 'state' argument is a boolean value of the initial state of the toggle button (TRUE == on/highlighted, FALSE == off). The next argument, a Widget, is NULL if this widget is a simple toggle button by itself and not part of a radio group (described below).

If you plan to display a bitmap for the toggle button, you may specify a NULL for the txt argument (and then call SetWidgetBitmap()).

The func argument is a standard callback function, that should be declared as follows:

```
void func(Widget w, void *data)
```

The last argument, data, is an arbitrary pointer you would like passed to your callback function (it becomes the second argument to the callback function.

Each time the widget changes state, your callback function is called. That is, each time the user clicks the toggle, your function is called.

Radio Groups:

It is possible to connect toggle widgets together to form a group of widgets that are mutually exclusive. That is to say, with a radio group, you can have a set of widgets in which at most one of them will be highlighted at any given time. Therefore, if you had 3 widgets, A, B, and C, only one could be highlighted at any one time, and clicking on another unhighlights the current one and highlights the toggle clicked on. This is useful for selecting one choice of several (such as a size, which is either small, medium or large, but not two at the same time). Keep in mind that it is possible for none of them to be selected.

To build a radio group, you use the Widget argument of the MakeToggle() function. If you specify another valid toggle widget in the call to MakeToggle(), the new widget becomes connected to the widget you specified. All the widgets you connect together form a radio group. Any single widget can only be in one radio group.

EXAMPLE:

```
Widget widg1, widg2, widg3;

widg1 = MakeToggleWidget("Thing 1", TRUE, NULL, func1, NULL);
widg2 = MakeToggleWidget("Thing 2", FALSE, widg1, func2, NULL);
widg3 = MakeToggleWidget("Thing 3", FALSE, widg1, func3, NULL);
```

Notice how widg2 and widg3 specify widg1 as their Widget argument. This connects all three into a radio group in which only one can be set at a time. We initialize widg1 to be initially set and the others off. If you specify more than one widget as 'on', the results are undefined.

The callback functions are called whenever a widget is highlighted or unhighlighted. The callbacks to the widget being unhighlighted happen before the callbacks to widgets being highlighted.

SEE ALSO: SetToggleState(), GetToggleState(), SetWidgetBitmap(), SetFgColor(), SetBgColor(), SetBorderColor(), SetWidgetFont()

## SetToggleState()

void SetToggleState(Widget w, int state);

SetToggleState() explicitly sets the state of a widget.

The 'state' argument is either TRUE (set the toggle to its highlighted state), or FALSE (unhighlight the widget). The callback routine for the widget is only called if there is a change in state.

SEE ALSO: GetToggleState(), MakeToggle()

## GetToggleState()

int GetToggleState(Widget w);

This function returns the current state of the toggle widget w. The return values are either TRUE (the widget is selected) or FALSE (the widget is not highlighted).

SEE ALSO: MakeToggle(), SetToggleState()

# Chapter 16

## Windows

This chapter contains descriptions of the main high level startup and window creation functions.

## OpenDisplay()

int OpenDisplay(int argc, char **argv);

This function initiates everything with libsx. You should call this function before you call any of the other functions. A correctly written application will call OpenDisplay(), passing its command line arguments and count. The return value from this function is the new number of arguments (or zero if an error occurred). The X libraries accept various standard command line options such as -display or -font, and if your application passes the command line arguments to OpenDisplay(), these will be handled properly. Any X options are removed from the argv array, therefore it is best if you do your command line processing after calling OpenDisplay().

You only need to call this function once to initialize the first window your program uses. Any other windows you need should be created with MakeWindow().

Technically, calling OpenDisplay() is optional (the MakeXXX() routines will call it for you if you didn't), but it's usually a good idea to call it (since it is only one line of code and it's pretty straightforward.

This function returns FALSE (0) if something went wrong (like being unable to open the display, etc). If everything went ok, it returns the new number of arguments in argv.

SEE ALSO: MakeWindow()

## ShowDisplay()

void ShowDisplay(void);

This function displays the currently active window (user interface) you've created with the MakeXXX() calls. After this call completes, the interface will be visible on the display.

Until you call this function, your interface will not be visible and drawing into a draw area will have no effect.

Usually one calls ShowDisplay(), allocates some colors and then immediately calls MainLoop(). If you do not call ShowDisplay(), but just directly call MainLoop(), then MainLoop() implicitly calls ShowDisplay().

SEE ALSO: MainLoop(), MakeWindow()

## MainLoop()

void MainLoop(void);

After calling this function, your program yields control to the user interface, and it is entirely driven by what the user does and the callbacks associated with the various widgets. For a single window application, the general flow of events is:

```
/* initialize the first window */
OpenDisplay(argc, argv);
```

```
/* create widgets */
MakeButton(....);
/* put the window on the screen */
ShowDisplay();
/* optionally allocate colors */
/* start the main loop going */
MainLoop();
```

When you call this after calling ShowDisplay() for your first window (created by OpenDisplay()), the MainLoop() function never returns and your application should have some callback function that will exit() the program (such as a quit button or menu option).

If you did not call ShowDisplay(), MainLoop() will call it for you and then launch into the main loop.

You should not call MainLoop() for NONEXCLUSIVE mode windows created with MakeWindow(). Those type of windows have their callbacks handled by the MainLoop() function that is already executing (i.e. the one you called for your original window).

If the window is an EXCLUSIVE mode window, then MainLoop() keeps executing until CloseWindow() is called on the EXCLUSIVE mode window. That is, MainLoop() blocks until the EXCLUSIVE mode window is closed, and then it returns.

If you create a non-exclusive mode window, the general order of events is:

```
MakeWindow(NONEXCLUSIVE_WINDOW, ....);
MakeButton(...);
ShowDisplay();
```

This creates a window, puts interface objects into it, and then puts that window on the screen. No other actions need to be taken, and when the callback that created this new window returns, all processing takes place normally, including the processing of the new window and its callbacks.

For a window of EXCLUSIVE_WINDOW mode (like a popup), the general order execution is:

```
MakeWindow(NONEXCLUSIVE_WINDOW, ....);
MakeButton(...);
ShowDisplay();

/* blocks until CloseWindow() is called */
MainLoop();


/* do something with whatever values the popup got for us */
```

When MainLoop() returns for an EXCLUSIVE_WINDOW, the window has been closed.

SEE ALSO: MakeWindow(), ShowDisplay().

## SyncDisplay()

void SyncDisplay(void);

This function synchronizes the display with all drawing requests you have made. Normally it is not necessary to call this function, but if you make many repeated function calls to draw graphics, they will be updated in a chunky fashion because X buffers drawing requests and sends a bunch of them at once.

After this function completes, all drawing requests you have made are visible on the screen.

NOTE: Normally you do not need to call this function because X ensures that everything you request gets drawn, but sometimes it is necessary to insure the synchronization of the display.

## MakeWindow()

Widget MakeWindow(char *window_name, char *display_name, int exclusive);

NOTE: Do not call this function to open your first window. Your application's first window is opened for you by OpenDisplay(). If your application only needs one window, do NOT call this function.

This function opens a new window, possibly on a different display (workstation). The new window has the name specified by the argument window_name and is opened on the display named by display_name (a string usually in the form of, "machine_name:0.0"). The final argument indicates whether the window should be an exclusive window or not (described below).

After this functions returns, the current window is the one you just created and you can begin adding widgets to it with the MakeXXX() calls. After have created and added any widgets you want, you should call ShowDisplay(), and if the window is an EXCLUSIVE_MODE window, then you should call MainLoop() (which blocks until the window is closed). If you opened the window with the NONEXCLUSIVE_WINDOW option, you should NOT call MainLoop().

If you pass a NULL for the window_name, it receives a default title of "Untitled".

If you pass the #define, SAME_DISPLAY, for the display name, the window opens on the same display as the original window opened by OpenDisplay().

The argument, exclusive, indicates what type of window to open. A normal window is a NONEXCLUSIVE_WINDOW. That is, it will not block the user from interacting with your existing window. An EXCLUSIVE_WINDOW is a popup window that blocks input to your main window until the popup is closed with CloseWindow().

The EXCLUSIVE_WINDOW mode is useful for requestors that need an answer and the user should not be able to do other things in the application. Of course some user-interface folks don't think modal windows like this are good, but tough cookies for them because some times it's necessary.

SEE ALSO: SetCurrentWindow()

## SetCurrentWindow()

void SetCurrentWindow(Widget w);

This function sets the currently active window for other function calls such as CloseWindow(). If you have multiple windows open on several displays, you must call this function switch the currently active one when you wish to manipulate the various windows.

The argument, w, must be a valid value returned by MakeWindow(). If you would like to set the current window to be the original window opened by OpenDisplay(), you can pass the #define, ORIGINAL_WINDOW.

When you change windows, the current drawing area is also changed to be the last current drawing area in the new window (if there is a drawing area in the new window).

SEE ALSO: MakeWindow(), CloseWindow()

## CloseWindow()

void CloseWindow(void);

This function closes and removes from the display, the currently active window.

After calling this function, you should not refer to any of the widgets contained in the window as they are invalid (as is the window handle).

SEE ALSO: SetCurrentWindow(), MakeWindow()

# Chapter 17 - Appendix

## Miscellaneous functions

The following function allows you to specify how the display should be layed out. It lets you logically position the components you created with the MakeXXX() functions. You will use this function to layout the arrangement of your buttons, labels and drawing area(s).

## SetWidgetPos()

void SetWidgetPos(Widget w, int where1, Widget from1, int where2,Widget from2);

This function lets you position a Widget in your window. The idea is that you specify logical placement of the Widget (i.e. place it to the right of this widget, and under that widget). Many layouts are possible, and you can even specify that you don't care where a specific widget is placed.

There are three types of placement. You can place a widget to the right of another widget with PLACE_RIGHT. If the argument "where1" is PLACE_RIGHT, then the Widget "w" will be placed to the right of the Widget "from1". If "where1" is equal to PLACE_UNDER, "w" will be placed under the widget "from1". The same holds true for the argument "where2" and Widget "from2". Having two arguments is necessary to be able to unambiguously specify where you want components placed in the display. If you don't care about where a widget is placed, you can use NO_CARE for the 'where' argument and a NULL value for the 'from' argument.

Generally, the first widget created need not be specified, it will always be in the top left corner. Other widgets can the be placed relative to that widget. For example, if you created 4 widgets (w[0] through w[3]) and wanted to arrange them in a column, you would do the following :

```
SetWidgetPos(w[1], PLACE_UNDER, w[0], NO_CARE, NULL);
SetWidgetPos(w[2], PLACE_UNDER, w[1], NO_CARE, NULL);
SetWidgetPos(w[3], PLACE_UNDER, w[2], NO_CARE, NULL);
```

Notice how the third argument changes; we are placing the next widget underneath the previous widget. The zero'th widget (w[0]) doesn't have to be placed because it is always in the top left corner (this can not be changed).

If you wanted to arrange things in a row, you would use PLACE_RIGHT instead of PLACE_UNDER.

As a more complicated example, supposed you want to create two rows of widgets, and a drawing area. You would do the following :

```
/* first three across the top */
SetWidgetPos(w[1], PLACE_RIGHT, w[0], NO_CARE, NULL);
SetWidgetPos(w[2], PLACE_RIGHT, w[1], NO_CARE, NULL);
SetWidgetPos(w[3], PLACE_RIGHT, w[2], NO_CARE, NULL);


/* next three underneath the top row */
SetWidgetPos(w[4], PLACE_UNDER, w[0], NO_CARE, NULL);
SetWidgetPos(w[5], PLACE_UNDER, w[0], PLACE_RIGHT, w[4]);
SetWidgetPos(w[6], PLACE_UNDER, w[0], PLACE_RIGHT, w[5]);


/* put the drawing area under the second row */
SetWidgetPos(w[7], PLACE_UNDER, w[4], NO_CARE, NULL);
```

It is useful to think of the window as a kind of grid in which you can put various pieces. Just draw a picture of what you want and then use SetWidgetPos() to indicate to the system what is next to/underneath of what.

Also, all imaginable layouts are not possible with SetWidgetPos(). For example, you cannot specify specific pixel offsets for a widget, or that it be centered in the display, or right justified. This limitaton is for the sake of simplicity. Generally this should not be a problem (if it is, you are probably getting beyond the scope of what libsx was intended to provide, i.e. you're becoming an X hacker :).

You can simulate more complicated layouts by cheating and creating label widgets whose label is just spaces and then placing other widget the left or underneath the label. This works but is kind of hackish.

## SetFgColor()

void SetFgColor(Widget w, int color);

This function sets the foreground color of a widget. If the widget is a drawing area, all future primitives are drawn with the specified color. If the widget is some other type of widget, it sets the foreground color of the widget (such as its text) to be the specified color.

The argument "color" should be an integer that was returned from the colormap functions (GetNamedColor(), GetRGBColor(), GetPrivateColor() or GetStandardColors()).

SEE ALSO : SetColor(), SetBgColor(), SetBorderColor(), GetStandardColors(), GetNamedColor(), GetRGBColor()

## SetBgColor()

void SetBgColor(Widget w, int color);

This function sets the background color of a widget. If the specified widget is a drawing area, the next call to ClearDisplay() will clear the drawing area to the specified background color.

The argument "color" should be an integer that was returned from the colormap functions (GetNamedColor(), GetRGBColor(), GetPrivateColor() or GetStandardColors()).

SEE ALSO : SetBgColor(), SetBorderColor(), GetStandardColors(), GetNamedColor(), GetRGBColor()

## SetBorderColor()

void SetBorderColor(Widget w, int color);

This argument will set the border color that is drawn around a widget. The same effect happens for all of the different widgets – the border is redrawn with the new color. This can be very useful for giving a nice visual offset to an important or dangerous button. Of course you should avoid garish combinations of colors that are hard to look at.

SEE ALSO : SetBgColor(), SetBorderColor(), GetStandardColors(), GetNamedColor(), GetRGBColor()

## GetFgColor()

int GetFgColor(Widget w);

This routine is a convience function that will return the current foreground color of any kind of widget. This is mainly useful for drawing widgets to make sure that you draw things in the proper

foreground color. This can arise as a problem if you assume that black is going to be the default foreground color (which it normally is). However, the user can change this default by using the -fg "color" option on the command line. This is an X command line option, and can not be overriden by your program. A real application would use this function to check the value and use it to draw in the user's preferred foreground color. Other programs can just ignore the problem and still work ok as long as the user doesn't change the program's colors.

This function returns the integer value of the foreground color that you can use in later calls to SetFgColor() or SetColor(). It returns -1 if you passed an invalid Widget to it.

SEE ALSO : GetBgColor(), SetColor(), SetFgColor()

## GetBgColor()

int GetBgColor(Widget w);

This routine is a convience function that will return the current background color of any kind of widget. This is mainly useful for drawing widgets to make sure that you draw things in the proper background color. This can be a problem if you assume that white is going to be the default background color (which it normally is). However, the user can change this default by using the -bg "color" option on the command line. This is an X command line option, and can not be overriden by your program. A real application would use this function to check the value and use it to draw in the user's preferred background color. Other programs can just ignore the problem and still work ok as long as the user doesn't change the program's colors.

The other problem that crops up if you ignore the background color is that if you go to erase something by just drawing in white and white doesn't happen to be the actual background color, your program will look funny.

This function returns the integer value of the background color that you can use in later calls to SetBgColor() or SetColor(). It returns -1 if you passed an invalid Widget to it.

SEE ALSO : GetFgColor(), SetColor(), SetFgColor()

## AddTimeOut()

void AddTimeOut(unsigned long interval, void (*func)(), void *data);

If you would like to animate a display or do some periodic processing (such as an auto-save feature for an editor), you can use time-outs.

A time-out is a callback function that gets called when the specified amount of time has expired (or I should say more precisely, when at least that much time has passed, Unix a'int no real time system).

The argument 'interval' is an unsigned long and is specified in milliseconds. That is, a time out of 1 second would be an argument of 1000.

The function, func, declared as follows:

```
void func(void *data, XtIntervalId *id)
```

The second argument should be ignored by function's code, but it should appear in the function prototype.

The function is only called once, if you would like the function to be called repeatedly (to update an animation for example), the last thing the function should do is to call AddTimeOut() again.

## SetWidgetState()

void SetWidgetState(Widget w, int state);

This function lets you enable or disable particular widgets in an interface. If, for example, choosing one item from a menu should disable various other widgets, you can call this function.

The Widget argument is the widget in question. The state argument is a boolean, which indicates whether the widget should be active or not. A value of TRUE indicates that the widget should accept input, and a value of FALSE indicates that the widget should not accept input (it becomes greyed out).

When you disable a widget, the user can no longer interact with that widget in any way (it becomes grey'ed out and just ignores all input).

## GetWidgetState()

int GetWidgetState(Widget w);

This function returns a boolean value indicating whether or not the specified widget is currently active.

If the widget is active and accepting input, the return is TRUE, if the widget is inactive, the return value is FALSE.

## Beep()

void Beep(void);

This function is real complicated. It beeps the workstation speaker.

## SetWidgetBitmap()

void SetWidgetBitmap(Widget w, char *data, int width, int height);

This function lets you attach a bitmap to a widget instead of its default text. This function only works correctly on Button, Toggle and Label widgets. Using it on another type of widget yields undefined results.

The Widget, w, will display the bitmap data given by the argument, data, whose width and height are given as the last two arguments.

The bitmap data is only one bitplane deep, and is usually produced by a somewhat brain-dead X program called 'bitmap'. The output of the bitmap program is a file you can directly #include in your source code. The contents of the file are a static array of characters and two #defines that give the width and height of the bitmap.

Thus, making a widget with a bitmap is a two step process. First you would edit a bitmap using the 'bitmap' program, then you would do the following:

```
#include "file_bmap.h"
Widget w;

w = MakeButton(NULL, func, some_data_ptr);
SetWidgetBitmap(w, file_bmap_bits,file_bmap_width,file_bmap_height);
```

Bits which are a one in the bitmap are drawn in the widget's current foreground color and zero bits are drawn in the current background color.