TWS Graphical User Interface Toolkit

Reference Manual & Tutorial

for use with Metagraphics MetaWINDOW™

Release 4.1.1

Distributed by:

METAGRAPHICS SOFTWARE CORPORATION
PO Box 225
Woodinville, WA 98072

Phone:	425-844-1110
Fax:	425-844-1112
Web:	http://www.metagraphics.com
Email:	sales@metagraphics.com
	support@metagraphics.com

CONTENTS

1. Introduction to the TWS Window System

The TWS™ is a Graphical User Interface (GUI) programming toolkit designed for use with applications running on PC-compatible platforms in either real- or protected-mode, or with embedded real-time operating systems supported by Metagraphics MetaWINDOW™. TWS uses an object-oriented event-driven programming model that is easy to program, and that provides a look and feel similar to Windows and Motif. TWS, however, is much smaller, faster and easier to use.

At its core, TWS provides support for multiple independent overlapping windows. In addition, a full compliment of user interface tools allow your application to integrate pull-down menus, dialog boxes, list boxes, edit boxes, check boxes, combo boxes, push buttons, radio buttons, sliders, scroll bars, and more. The TWS library provides a high-level layer between an application program and the underlying MetaWINDOW graphics kernel that contains the foundation graphic drivers and drawing functions.

You are encouraged to run the demos and begin using the TWS system. Be sure to register your TWS Development Toolkit to receive the latest source code updates, documentation, sample programs, tutorials and email-based help.

1.1. Legal Stuff

1.1.1. Copyrights

1.1.2. Warranty

The diskette on which this document and accompanying materials are distributed is warranted to be free from material defects for a period of thirty (30) days following the date of purchase. If notified within the warranty period, any defective diskette(s) will be replaced. Remedy for breach of warranty shall be limited to replacement of the defective diskette(s) and shall not include any other damages, including but not limited to loss of profit or other special or incidental damages. This constitutes the entire warranty. All other warranties expressed or implied are specifically denied, including but not limited to implied warranties of merchantability and fitness for a particular purpose. The software in the TWS distribution is not warranted to be free from defect. In no event shall TWS be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages in excess of the registration fee.

1.1.3. Software License Agreement

Registration implies acceptance of the following terms and conditions. If you do not agree with the following then promptly discontinue use of this product.

This software is protected by United States copyright law and international treaty provisions. Distribution outside the United States is subject to export restrictions.

Single-User License
The TWS Development Toolkit is licensed for use by a single developer. Similar to your compiler and other development tools, additional developers are required to each purchase an individual license of the TWS Development Toolkit.

Redistributable Components
Executable application programs written using TWS may be used, distributed or sold without additional licensing under the following conditions:

1) The application developer(s) are registered TWS Development Toolkit licensees. Distribution of software containing elements of TWS prior to registering the TWS software is a violation of federal law;
2) The programs developed using TWS must not compete with TWS - that is, you can't use the TWS library to develop another Graphics User Interface library or similar toolkit;
3) The TWS source code, example programs source code, or any part of this documentation may not be redistributed.

Permission to copy the TWS Development Toolkit is granted for personal use (such as backups).

This license applies to the TWS Graphical User Interface Library System only. License agreements for Metagraphics MetaWINDOW, your programming language compiler, or any other required software must be obtained separately.

1.1.4. Trademarks

Borland C, Borland C++, Turbo C, Turbo C++ are trademarks of Borland International, Inc. *MetaWINDOW, FontBUILDER, FontWINDOW* are trademarks of Metagraphics Software Corporation, PO Box 225, Woodinville, WA 98072 (425) 844-1110. All other trademarks used herein are the property of their respective owners.

1.2. Files in the Distribution

The TWS distribution consists of the following files:

TWS.LIB	Borland/Turbo C library for the TWS window system. The library was compiled using Borland C/C++ version 3.
TWSDOC.ASC	7-bit ASCII documentation, 80-column lines, no page breaks.
README	ASCII file of last minute notes, registration information, etc.
EXAMPLES.EXE	Self-extracting archive of TWS demos and example programs to accompany the tutorial in the documentation.
*.FNT	Various font files. The TWS system will use any font file supported by the underlying graphics kernel. A collection of fonts the TWS system will default to is supplied with the distribution, if appropriate.
*.ICN	Various pixmap/icon files. This is an ever-changing collection. Pixmap files are device-independent. A program for viewing/editing icon pixmap files is included in the distribution.

The EXAMPLES archive contains programs to both demonstrate the TWS library and to help validate code changes. This archive contains the following programs:

Program	*Purpose*	*Description*
WINTEST2	Basic event loop	Initializes the graphics system and workspace, and enters the event loop. See the Tutorial section later in this document.

WINTEST3	Application menu	Adds an application menu to the workspace window. See the Tutorial section later in this document.
WINTEST4	DOCUMENT window create and display	Opens an empty DOCUMENT application window inside the workspace. See the Tutorial section later in this document.
WINTEST5	Pull-down menus, window menus.	Adds pull-down menu items to the workspace window and a window menu to the application window.
WINTEST6	Button gadget	Puts a button and default button in a window, attaches a user function to the buttons
WINTEST7	Editstring gadget	Puts an editstring in the workspace window with text.
WINTEST8	Color	Tests the basic TWS color system. ***Requires 256-color graphics.***
WINTEST9	Backing store	Same as WINTEST8 but with window backing store enabled.
WINTESTA	Color sharing	Same as WINTEST9 but with two windows using the same colors but in different palette positions. Test of the sharing logic in the color system.
WINTESTB	Checkbox gadget	Displays a checkbox in a window. Attaches a user function to it.
WINTESTC	Label gadget	Displays bordered and unbordered labels in a window.
WINTESTD	Stringlist gadget	Displays a stringlist in a window. Attaches a user function to the stringlist.
WINTESTE	Slider gadget	Displays horizontal and vertical sliders in a window, attaches a user function to them.
WINTESTF	Picklist composite gadget	Displays a picklist.
WINTESTG	Graphics label	Creates a label with graphics instead of text. Demonstrates casting a gadget to a window for drawing.
WINTESTH	Pixmap-in-a-button	Demonstrates using pixmaps as button facings.
WINTESTJ	Pixmap gadgets	Demonstrates pixmaps as standalone gadgets.
WINTESTK	GetFilename standard dialog	Demonstrates the standard dialog, SM_GetFilename.
WINTESTL	YesNoDialog	Demonstrates the two-state standard dialog.
WINTESTM	Rotatelist test	Demonstrates the new Rotatelist gadget for selecting from a group of strings in a limited space.
WINTESTN	Focus window change event procedures	Demonstrates setting up application handlers for the window GetFocus and LoseFocus events.
WINTESTO	Event return values	Provides a real-time display of the values returned by the event handler as the mouse is moved around the screen, keys pressed, etc.

WINTESTP	Graphics image routines	Demonstrates the new ImageType window graphics routines. Copies a block of graphics from one window to another.
WINTESTQ	Borders	*NEW!* Demonstrates the new decorative border gadget.
WINTESTR	Hotregion	*NEW!* Demonstrates the new Hotregion gadget.
WINTESTS	Window icons	*NEW!* Demonstrates iconizing windows sharing the same Pixmap.
CALC	A simple 4-function calculator.	Demonstrates virtual coordinates for buttons and labels; gadget functions; attaching application data to a gadget. See the Tutorial section later in this document for a detailed discussion of this program.
CLOCK	Analog clock display	Demonstrates virtual coordinates for the graphics canvas; background processing; the CLOCKTICK event.
TRISERAC	Abstract graphic display	Demonstrates background processing; Textbox gadget; window resize and close procedures. See the Tutorial section later in this document for a detailed discussion of this program.
ICONEDIT	Pixmap/icon editor	A program for viewing and modifying TWS pixmaps suitable for window icons. Uses a number of TWS features, including pixmap gadgets, pixmaps as button facing, drawing, modal dialog functions, button graphics functions, label graphics functions, and standard dialogs.
WINIMAGE	PCX Image file display	Reads and displays PCX files in a window. If the image window is resized, the image in it is resized to match. Demonstrates color management, among other things. *256-color graphics recommended.*
WINDRAW	Simple interactive graphics	Demonstrates interaction techniques; trapping the mouse; window procedures. See the Tutorial section later in this document for a detailed discussion of this program.

1.3. Linking the TWS Library with Application Code

The TWS library is linked with your application just like any other third party code library. Here's an example using the MetaWINDOW graphics kernel library and Borland's linker:

```
tlink c0l myprog, myprog,, tws.lib metawin\met_xd1d.lib emu.lib mathl.lib cl.lib
```

This example assumes your program object file is called `myprog.obj` and you want the executable to be called `myprog.exe`, and the appropriate directories are part of your system path.
A few important points about building and using the TWS library:

- For real-mode DOS, all library code is built using the LARGE memory model. Your program must also be compiled using this model;
- The TWS library uses some floating point math so your application must link in a floating point library;

- The TWS code requires the Metagraphics MetaWINDOW library. The linker may have restrictions concerning which library should be first;

Borland C/C++:

- Application code must be *byte-aligned* for proper operation.

Make files are included for all the TWS demo programs. Use these as templates for building your own applications. For more about writing TWS programs see the **Tutorial** chapter later in this manual.

Part One

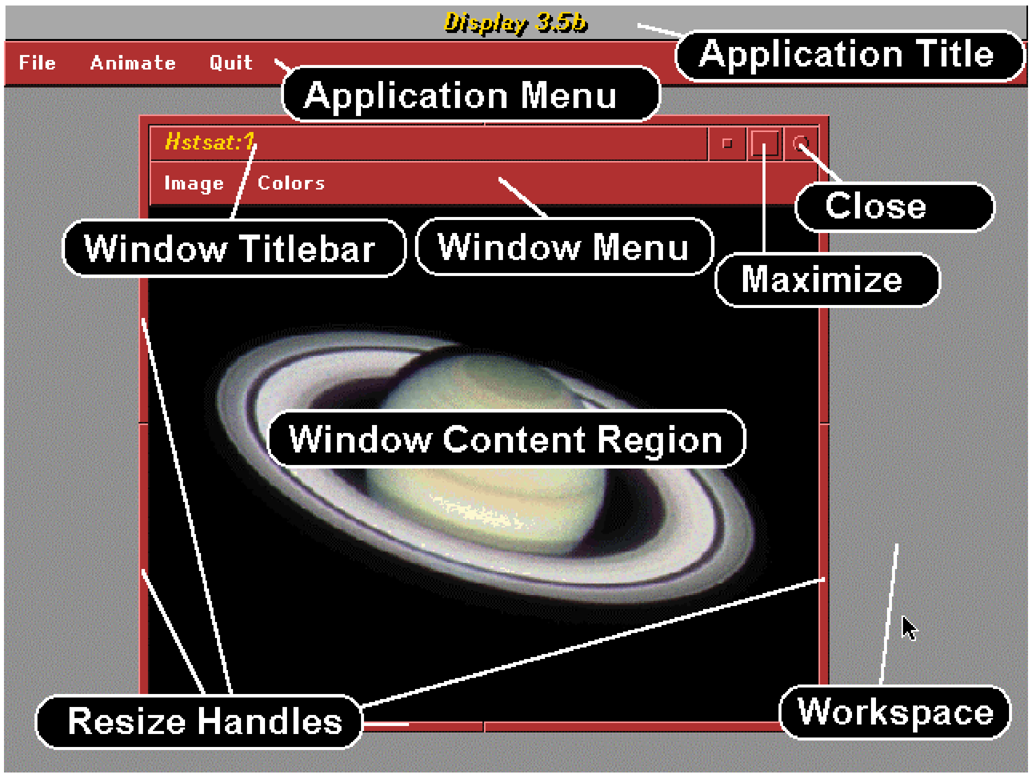
The Window System

2. System Overview

TWS is a library of functions for creating robust event-driven, object-centric DOS applications that employ a graphical user interface. The look and feel of TWS is derived from the X-Windows OSF/Motif interface. However, TWS is much simpler to program.

2.1. Screen Display Organization

The TWS screen is organized into distinct functional areas. Those areas are:



....

Workspace All application activity takes place within the *workspace* which occupies the entire screen. This window is divided into the following regions:

Application Title	The name of the application is displayed here; otherwise this region has no special significance. The title may be omitted.
Application Menu	The main menu for the application. If present, this menu is always enabled even when application windows have the focus. An application doesn't have to have an application menu.
Workspace Content	This is where application activity can occur, normally within application windows. As far as most TWS functions are concerned, the workspace is just another window. An application can attach gadgets, draw graphics, or just about anything else in the workspace.

Application Windows An application can create any number of individual windows for drawing, displaying text, user input, program output, etc. Application windows can only exist within the Workspace Region. Three kinds of windows are supported:

Dialog Window	A basic fixed-size window for data input and messages. Its components are:
Title Bar	Displays the window's label. The window is moved by grabbing the title bar and dragging the window around within the Base Window Desktop Region.
Close Button	Closes the window. Control will go to the next highest window in the stack, if any.
Window Menu	(Optional). If present the window's menu is a horizontal bar underneath the window title bar. The menu will work in a "pull-down" fashion.
Content Region	All application output to any particular window will appear in the content region of the window. This is the only part of the window the application will be able to directly modify.
Document Window	A general window. It contains the following components:
Title Bar	Same as under the Dialog Window.
Resize Handles	Each quadrant of the window border has a resize handle region. The document window can be resized in the direction of the quadrant by grabbing and dragging a resize handle.
Maximize Button	Will cause the window to be redrawn at it's maximum possible size.
Minimize Button	Causes the window to become "iconized" if it has an icon attached to it, otherwise does nothing.
Close Button	Closes the window. Control will go to the next highest window in the stack, if any.
Window Menu	(Optional). If present the window's menu is a horizontal bar underneath the window title bar. The menu will work in a "pull-down" fashion.
Content Region	All application output to any particular window will appear in the content region of the window. This is the only part of the window the application will be able to directly modify.
Simple Window	This window has only a border. It's useful for simple message dialogs.
Panel Window	(Optional, not illustrated). This is a special kind of "tiled" window at the workspace level.

2.2. System Requirements

Developing applications with the TWS window system requires a computer based on an 80*x*86 processor running DOS 3.0 or above, 640k bytes RAM, keyboard, Microsoft-compatible mouse, and hard disk drive. A supported graphics subsystem is also required. The graphics kernel system vendor can supply a list of supported devices.

Applications developed using TWS will have varying requirements depending on the application but all will require a supported graphics subsystem including the mouse. A hard disk drive is strongly recommended for all TWS-based applications.

2.3. System Design

This section describes the major concepts and subsystems of the TWS GUI library. Each part of the TWS library works in concert with the rest. Rather than being a disjoint collection of independent graphical add-on functions, TWS is a comprehensive user interface and application design environment.

2.3.1. Windows

The *window* is one of the fundamental structures of a TWS application. It provides the basis for all other TWS operations -- user interface gadgets, drawing, etc., all happen relative to a parent window. Windows also provide an encapsulation layer for application data and functionality — data, by means of generic 'hooks' in the window data structure, and functionality through gadget and window callback functions.

2.3.2. The Window Stack

Windows in TWS are organized as a stack. The active window is at the top of the stack. Most output can be sent to any window, but only the active window will respond to interactive events. For example, you can draw in an inactive window but clicking on a button in that window will not activate the button.

When a window is created and opened it is placed at the top of the stack. When a window is closed the next-highest window in stacking order is automatically activated.

2.3.3. Save-unders and Backing Store

TWS uses save-unders and backing store. Save-unders are mostly used for pull-down menus. Backing store is used for window graphics canvas regions. If either save-under or backing store fails it aborts the application. The application determines whether or not a window will use backing store when the window is created, but menu save-unders are non-negotiable.

2.3.4. Gadgets

Most user interaction in a TWS application is via interface devices called *gadgets*. Gadgets include buttons, labels, editable strings, radio buttons and checkboxes, sliders, etc. By manipulating the gadgets in an application, the user causes things to happen.

TWS gadgets are objects in the sense that they encapsulate their own data and behavior, as well as application data and behavior. Data is passed to the application function by attaching it to the button itself, so gadgets also serve as a primary data passing mechanism.

2.3.5. Event Processing

Actions in an event-driven application are generally initiated as the result of external events like mouse buttons and keyboard keys being pressed, and internal events like the system clock. There is no way in general to predict what event will occur next based on preceding events, and there is no good mechanism for ensuring that any specific sequence of events will occur. This is somewhat contrary to "traditional" programming, but it's a lot like real life. Your stereo doesn't know which buttons or channel you're going to select next, but manages to work fine nonetheless.

2.3.6. Window Coordinate System

The window coordinate system is relative to the upper-left corner of the window content region, which is at coordinate (0,0). Coordinates increase to the right and downward. A position outside the content region is invisible.

The sizes of all objects in a window are in screen pixels. An exception is certain gadgets which can be specified in *virtual coordinates*, or fractions of the size of the window content. The actual display size of such a gadget depends on the actual size of the window content when the gadget is drawn.

2.3.7. Graphics

TWS graphics are part of a subsystem called the *graphics state*. The graphics state contains all information necessary to do freeform drawing — a local color palette table, current foreground and background colors, line thickness, graphics position, etc. The graphics state also contains the rectangular region where all graphics are drawn. This region is relative to the content region of the window, like gadgets. All drawing is clipped to the graphics canvas, which is clipped within the window content.

2.3.8. Background Processing

A simple “cooperative multi-threading” system allows portions of an application to execute in the background with minimum impact on the main program and user interface. The application decides how much processing is done during each time slice. The background task manager works with the event manager so that background functions can be sensitive to events, such as clock timer events, and the system state, such as only being called if a particular window is active.

2.3.9. Callback Functions

The main link between applications and the TWS user interface is via callback functions. A callback function is an application function that’s designed to be called whenever a particular action occurs. Mostly, callback functions are executed when a gadget is activated. Each gadget has its own set of conditions that cause it to execute its callback function, usually when the state of the gadget is changed. For example, clicking a button causes its callback function to be executed.
Windows may also have callback functions, called *window procedures*. The window procedure for the active window, if it has one, is called on each pass through the event loop.

3. The TWS Window System

The *window* is the fundamental data structure in the TWS system. Nothing happens in a TWS program except as part of or relative to some window. Even the workspace region is implemented as a window (albeit a special one) and has just about the same interface as other windows (for example, gadgets can be attached to the workspace window, and you can create a workspace graphics canvas and draw in it).

3.1. Window Types

Windows are defined by their type and their attributes. The window *type* determines how the window responds to user input, and the *attributes* determine other types of behavior and appearance. The different window types are described below.

3.1.1. Document-Type

A **DOCUMENT**-type window is the most flexible type. It can be moved, resized, maximized, minimized, and closed, all by user-input or through application functions. Visually a DOCUMENT window is distinguished by four chiseled sections on its border. DOCUMENT windows respond to the following input events:

Event	***Result***
Click-and-drag title bar	Moves the window on the screen.
Click-and-drag within a border section	Resize the window in the direction of the border. The opposite border is anchored. Resizing is “diagonal” — the window can be resized in both width and height.
Click the Minimize button	If the window has an icon attached to it, the window is minimized. Otherwise nothing happens.
Click the Maximize button	Enlarges the window to fit the workspace content region. The maximized window obscures the application menu, if any, but not any panels.
Click the Close button	Closes the window.

3.1.2. Dialog-Type

DIALOG-type windows are similar to DOCUMENT-type, except they can’t be resized by the user. They can be moved, however, in the manner described above. Their borders are solid and are not sensitive to user input actions.

3.1.3. Simple-Type

A SIMPLE-type window has no border or title and can’t be moved or resized by user input. They may not have menus. Since they don’t have title bars and therefore don’t have a close button, the application must supply a specific means for the user to close a SIMPLE-type window, such as a button or a time-out. SIMPLE windows are used for basic message boxes (for example, see the chapter on Standard Dialogs).

3.1.4. Window Attributes

Attributes determine other internal behaviors and appearance features of the window. The window attributes are set when the window is created. Attributes can be combined by OR-ing them.

3.1.4.1. SCROLLATTRIB

The SCROLLATTRIB attribute determines whether the window will be scrollable. If this attribute is set then the window will be drawn with horizontal and vertical scrollbars.
Window scrolling is described in a following chapter.

3.1.4.2. BACKING

When a window's BACKING attribute is set, the contents of the window's graphics canvas region is written to disk each time the window is drawn. When a window graphics canvas must be redrawn (i.e., restored with no changes), the contents can be restored from disk.
The BACKING attribute has some limitations. This is discussed in detail in the Graphics chapter.

3.1.4.3. ICONONLY

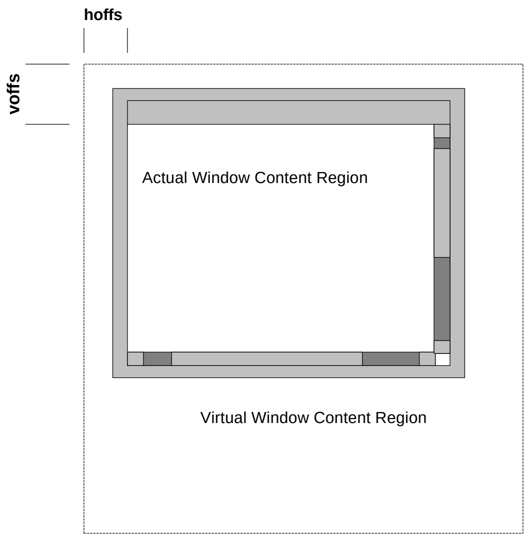
The window can only exist as an icon. Implies the ICONINIT attribute.

3.1.4.4. ICONINIT

When the window is opened, start it as an icon. The window must have an icon attached before **SM_OpenWindow** is called.

3.2. Window Scrollbars

A window with its SCROLLATTRIB attribute set when it's created is drawn with scrollbars around its border. When a window is scrollable, the area displayed in the window's content region is a portion of a larger 'virtual' region whose boundaries are defined by the application. As the scrollbars are moved, a different area is displayed.



In order for a window to have scrollbars, it must be created with the SCROLLATTRIB attribute:

```
w = SM_NewWindow(&winrect, "ScrollWindow", DIALOG | SCROLLATTRIB, NULL, NULL);
```

By default, the size of the virtual region is the same as the window content. The virtual region size can be changed by calling **SM_SetWindowScrollbarMinMax**:

```
winrect.Xmin = 150;
winrect.Xmax = 300;
winrect.Ymin = 100;
winrect.Ymax = 350;
w = SM_NewWindow(&winrect, "ScrollWindow", DIALOG | SCROLLATTRIB, NULL, NULL);
SM_SetWindowScrollbarMinMax(w, HVERTICAL, -50, 350);
```

The effect of the code segment above is to create a window whose content region is 150 pixels in the X (horizontal) direction, and 250 pixels in the vertical direction. Then, the range of pixels for the vertical (Y) scrollbar is changed from the default (0..250) to (-50, 350). Since the range displayable in the window is now less than the total virtual range, the scrollbar drag bar is resized accordingly.

3.3. Using Scrollbars

In the simplest case the application doesn't have to do anything to use window scrollbars except include the SCROLLATTRIB attribute when the window is created and set the minimum and maximum values as shown above. TWS sets a default procedure for the scrollbar that changes the offset for the virtual content region and redraws the window in real time as the scrollbar is moved.
There will be many cases when it's impractical for the system to update a window as the scrollbar is dragged, particularly when there are complicated graphics involved. The application will instead

want to defer updating the screen until the user finishes dragging the scrollbar. In these cases the application will want to replace the default scrollbar procedure with a different one.
In the end a window scrollbar is just a scrollbar (so most everything in the Scrollbar chapter applies). To modify a scrollbar's attribute requires the scrollbar pointer. For window scrollbars use the **SM_GetWindowScrollbar** function to get this pointer:

```
ScrollbarType  *hwscroll;
hwscroll = SM_GetWindowScrollbar(w, HSCROLLBAR)
```

To get the vertical scrollbar, use VSCROLLBAR instead.
The scrollbar callback function can now be changed using the **SM_SetScrollbarProc** function:

```
SM_SetScrollbarProc(hwscroll, MyNewFunc);
```

The application window scrollbar callback must do one thing that a gadget scrollbar doesn't have to do — it must explicitly change the content region's position relative to the virtual content region. Otherwise the window contents won't change position.[1] Set the window's horizontal and vertical offsets using the **SM_SetWindowOffset** function:

```
int MyScrollbarFunc(SliderType *slider)
{
        int     position;
        /* other processing ... */
        position = SM_GetSliderPosition(slider);
        SM_SetWindowOffset(SM_GetGadgetWindow(slider), HSCROLLBAR, position);
}
```

3.3.1. Window Scrollbars and Resizing

When a user resizes a window, the area inside the content region changes, but the area of the *virtual* content region doesn't. Some applications may want to make the virtual area always relative to the size of the actual window content. This can be done by attaching an application callback function to the resize window event. This is discussed in a later chapter.
As the window content region size changes relative to the virtual content, the window scrollbar thumbbars (or drag bars) are resized accordingly. This does not trigger the scrollbar callback function.
The virtual content will not become smaller than the actual window content region no matter how small the window gets. All objects whose coordinates are within the actual window content boundaries will be visible.

3.3.2. Window Scrollbar Callback Procedures

The default action when a window scrollbar is manipulated by the user is to change the origin within the content region relative to the virtual region, then redraw the window. The effect of this is to move all the window contents in real time as the user drags the thumbbar. This includes the graphics state region, which means any graphics will also be redrawn.
The application can attach a local callback function to the window scrollbar which is called whenever the scrollbar's position changes (see the chapter on the ScrollbarType gadget for details). The application callback function can augment or change the behavior of the scrollbar.
In particular, the application may want to override the window redrawing because if there are many gadgets or complex drawings, the redraw won't be able to keep up. The application will want to change this so that the window is redrawn only after the user has finished moving the scrollbar. The chapter on the Scrollbar gadget describes how to do this.

[1] Unless you don't *want* the contents to change. It's up to you, really.

3.4. Window Management Events

Internally every window responds to a number of *window management events*. There are six window management events: Drawing, Redrawing, Resizing, Closing, Getting Focus, and Losing Focus. In each case the window manager performs some default action (often it does nothing). The default behavior can be modified by application code. The application functions which modify window management events are called *window management callback procedures.*
All windows that draw graphics must supply callback functions for the Draw and Resize events if the drawing is going to survive moving, iconizing, and resizing of the window it's in. It is *not* necessary to provide any extra handling just to manage gadgets, because the system itself takes care of drawing them as necessary. This includes gadgets that can contain graphics, like pixmaps, buttons and labels.

3.4.1. Window Management Callback Procedures

For each window management event there is a hook for the user to supply functionality to replace the default TWS behavior. A window manager procedure is an int function whose single argument is a pointer to a window. When called, the pointer will point to the window where the window management event occurred. Currently, the return value from the function is ignored.

Event	*Default Behavior*	*Description*
Draw	Clear the content region to the window's content color.	Function called when the window contents must be generated from scratch, such as when the window is first created.
Redraw	Do the same thing that is done for the Draw event.	Function called when the window must be restored but the window hasn't changed; for example, when the window is moved. Usually this function is optimized to provide a very fast way of restoring the window's content or graphics canvas, usually by reading an image from disk. See also the BACKING window attribute.
Resize	Do the same thing that is done for the Draw event.	Called when the window is resized. Most applications that use resizeable windows will have window-size-dependent data. For example, a window that displays text may need to recalculate how many lines will fit in the window. An application that displays images may want to scale the image to fit the window.
Close	Nothing	When a window is closed TWS reclaims all memory and destroys all internal structures as well as window gadgets (but not menus). The window itself is erased and the next window on the window stack becomes active. Often an application window will have application-specific data attached to it's data field; TWS can't know how to deal with this data, so the user must supply a function for dealing with it. Also, TWS doesn't verify that a window should be closed; often an application will supply confirmation dialogs before closing certain windows, especially if data might be lost. The window management procedure can cancel the closing of the window by returning a non-zero value.
GetFocus	Nothing	

LoseFocus	Nothing	The GetFocus function is called *after* the window has been brought to the top of the stack and redrawn. The LoseFocus function is called *before* the active window actually changes. The return value from the function is ignored, and the application can't prevent the active window change through these functions.

3.4.2. Setting a Window Event Callback

A window management event callback function is an int function whose argument is the window affected by the event. Each event has a corresponding **Set...** and **Get...** function. To set an application function that will be called whenever the window becomes active, for example, use:

```
SM_SetGainFocusProc(w, AppFunc);
```

3.5. Configuration File

Most TWS system parameters have reasonable preset defaults so that a basic TWS program can be run with a minimum of fuss. However, some window parameters can be customized when a TWS application is started via a configuration file.
During initialization TWS looks for an environment variable called TWSCONFIG. This variable, if present, is assumed to contain the path and filename of a configuration file for all TWS applications. This file is opened and its contents read by the initialization procedure.
If the TWSCONFIG environment variable is not set, the initialization procedure then looks for a file called TWS.CFG in the current directory. If not found, it then looks for TWS.CFG in the root directory of the current disk. If either of these succeeds the file is opened and read. Otherwise the system proceeds with internal default values for all system parameters.

3.5.1. Configuration File Format

The TWS configuration file is a collection of attribute-value pairs. Defined attributes are:

Parameter	***Description***
BorderWidth	Width, in pixels, for the window borders/resize handles. If no window border width is specified in a configuration file then TWS calculates a reasonable size based on the display resolution.
GraphDevice	A code for the desired graphics display and resolution. Please see Appendix D in the Metagraphics *MetaWINDOW Reference Manual* for a list of supported display modes. The "metconst.h" header file included with MetaWINDOW provides an master list of graphics adaptor definitions for display modes supported by MetaWINDOW along with their associated values for GraphDevice.
MouseDevice	A code for the desired graphics pointer device. Please see Appendix D in the Metagraphics *MetaWINDOW Reference Manual* for a list of supported mouse input devices. The "metconst.h" header file included with MetaWINDOW provides a master list of input device definitions for mice supported by MetaWINDOW along with their associated values for MouseDevice.
FontPath	Directory path for system font files. If no font path is specified in a configuration file then TWS expects font files to be in the current directory.
IconPath	Directory path for icon pixmap files. If no path is specified, TWS expects icon pixmaps to be located in the current directory.

SystemFont	Name of the font file to use as the system font, which is used on menus, dialogs, labels, etc., but not window titles or text displays. If no system font file is specified in a configuration file then TWS uses the *sys256.fnt* if the display supports 256 colors, or *sys16.fnt* if the system supports 16 colors. These fonts are included in the TWS distribution. The files are assumed to be located in the FontPath directory, described above.
TitleFont	Name of the font file to use as the window title font. If no title font file is specified in a configuration file then TWS uses *title256.fnt* if the display supports 256 colors, or *title16.fnt* if the system supports 16 colors. These fonts are included in the TWS distribution. The files are assumed to be located in the FontPath directory, described above.
TextFont	Name of the font file to use as the window text font. If no text font file is specified in a configuration file then TWS uses the system font.
BevelDepth	The amount of "sculpting" for 3D window elements. Values is the number of pixels devoted to the sculpted appearance. For window borders the sculpting is drawn to the inside of the window border, so the border width should be at least 2*BevelDepth+1. For buttons and other gadgets the sculpting is drawn inside the gadget boundary. Default is 1 pixel of bevel for screen resolutions less than 1024x768, and 2 pixels for 1024x768 and higher resolutions.
FocusColor	Sets the active_border_color in the window defaults file, which is the active window border, title, and menu color. Specified as an RGB triple, with each primary color in the range 0..255 and separated by at least one space.
WorkspaceColor	Sets the color for the application workspace background. Specified as an RGB triple, with each primary color in the range 0..255 and separated by at least one space.
AppTitle	The string to display in the application title bar.
SysDevice	Device and path to use for window temporary storage. Normally TWS uses the default drive and directory for backing store, image saves and other temporary storage. The system device attribute lets the application use a fast RAM drive, if available, which will significantly speed some operations.
WindowLimits	Controls whether or not new windows will be constrained to the screen limits. If ON (the default), if the application tries to create a window that would extend off the edge of the physical screen, the system will reduce the window in that dimension to fit. When OFF the application can create windows of any size. In all cases, windows whose graphics canvases are not on the physical screen and which have BACKING enabled may not redraw properly due to hardware constraints.

Configuration data is not case-sensitive for either the attributes or values, although mixed case is shown in the table for readability. If an attribute is specified more than once then the last value specified has precedence.

Most information in the configuration file is device-independent. The exception is the GraphDevice and MouseDevice. These values depend on the hardware installed in the computer that's running the application. It's up to the application to insure these values are valid.

3.5.1.1. Example

The following is an example TWS configuration file for the MetaWINDOW 4 graphics kernel:

```
    GraphDevice        8500
    MouseDevice        49
    SystemFont sys16.fnt
    TitleFont  title16.fnt
    TextFont   sys16.fnt
    FocusColor 240 138 79
    WindowLimits       Off
```

In the example, the graphics device is an S3-based video system at 1024x768 resolution, 256 colors (8500), Microsoft Mouse driver (49). The system and text fonts are `sys16.fnt` in the local directory, and the title font `title16.fnt` is also in the local directory.

3.6. Windows

The window is the focus for all input and output in TWS. Windows are self-contained with their own coordinate system, interface gadgets, graphics attributes and so forth. You can also attach application data to windows.

3.7. Include file: smwindow.h

```
/*
** The window structure
*/
typedef struct _win {
    /*
    ** Window display regions
    */
    RectType window;                    /* Max extent of complete window    */
    RectType content;                   /* Drawing area                     */
    RectType orig_content;              /* Old content region for max       */
    RectType titlebar;
    RectType stretch_left,              /* Window 'grab' regions            */
             stretch_up,
             stretch_right,
             stretch_down,
             menubutton,
             iconize,
             fullscreen;
    PolygonType upleft,
            upright,
            downleft,
            downright;
    RectType windowmenu;                /* Window menu region               */

    /*
    ** Minimum width/height for the window
    */
    int     minx, miny;

    char    *title;                    /* Titlebar title                   */
    void    *icon;                     /* Icon version of the window       */
    int     is_icon;                   /* TRUE if window is iconized       */
    int     is_maximized;              /* TRUE if window is maximized      */
    int     type;                      /* Window type                      */
    void    *data;                     /* User-defined data                */
    void    *gadgets;                  /* Window gadget list               */
    /*
    ** Window procedures
    */
    int     (*windrawproc)(struct _win *);/* Procedure for drawing win borders*/
    int     (*drawproc)(struct _win *);   /* Procedure for drawing contents   */
    int     (*redrawproc)(struct _win *); /* Procedure for redrawing contents */
    int     (*resizeproc)(struct _win *); /* Procedure for redrawing contents */
    int     (*userproc)(struct _win *, EventType *e);
    int     (*closeproc)(struct _win *);  /* Procedure when window closed     */
    int     (*losefocusproc)(struct _win *);
    int     (*getfocusproc)(struct _win *);
    /*
    ** The window graphics state
    */
    GraphStateType *gs;
    /*
    ** Content region colors
    */
    int     ncolors;                    /* Size of the window color table   */
    MenuType *menu;                     /* Window menu                      */
    ColorType content_color;            /* Color of content region          */
    int     redraw;                     /* TRUE if window needs to be drawn */
    int     locked;                     /* TRUE if window is locked         */
    struct _win *next, *prev;
} WindowType;
```

3.8. Interface Functions *smwindow.h*

3.8.1. Initialization Functions

This group of functions create, open, and close the window system and individual windows. For any TWS application, the following steps are required:

- Initialize the graphics kernel system and window system by calling **SM_Init**;
- Initialize the application by calling **SM_OpenApplication**;
- For each window in the application, **SM_NewWindow** creates and initializes it, then...
- Call **SM_OpenWindow** to open each window. **SM_CloseWindow** closes them;
- Process events;
- Close the application and restore the system by calling **SM_Exit**.

3.8.1.1. int SM_CloseAllWindows()

Closes all open windows except the workspace window. This is more convenient than saving the window value for each open window, then closing each one individually.

3.8.1.2. int SM_CloseWindow(WindowType *w)

Destroys the specified window and frees all memory associated with it. The window must have been previously opened by the **SM_OpenWindow** function. **SM_CloseWindow** calls the argument window's close window management event procedure, if any, before closing the window itself. If the event procedure returns a non-zero value then the window is not closed. This is the function called by the event manager when a window's Close button is pressed.

3.8.1.3. int SM_Exit(WindowType *w)

Shuts down the window system and returns to DOS. This function does not return to the application. **SM_Exit** does not use the window argument, so the application often supplies a NULL argument. The *w* argument is part of the function so it can be attached to a menu item. All TWS applications *must* call **SM_Exit** to restore interrupt handlers and return to character-mode DOS; otherwise, the computer will surely hang.

3.8.1.4. int SM_Init(int graphdevice, int mousedevice)

Initializes the graphics kernel system, window manager, event system, fonts, system colors, and reads the configuration file. This function must be called before any other window manager functions.
The relevance of the command line argument string(s) in *argv* depends on the underlying graphics kernel system. In general they override the graphics device and mouse device that may be in the configuration file. If either device is 0 then the configuration option is used, otherwise the function argument is used.

3.8.1.5. int SM_LoadSystemFont(char *fontname)

Reads a font from the file *fontname* and makes it the system font. The font file is first searched for in the current directory, then in the directory specified by the FontPath configuration parameter, if any. The existing system font is discarded in favor of the new font. The font must be in the correct format for the graphics kernel system. Does not change any screen elements already drawn. *See also: SystemFont configuration parameter.*

3.8.1.6. WindowType *SM_NewWindow(RectType *s, char *t, int ty, MenuType *m, void *d)

Creates a new instance of a window and returns a pointer to it. Does not draw to the screen.

s ...... The size and position of the window's content region. The system builds the window borders, title, menus and so forth around the outside of the content. This is usually more convenient for the application. The system won't allow a window to be created that is larger than the physical screen. The window, including the size of the content region *r*, will be reduced if necessary to keep the entire window on the screen.
t....... Title for the window. The system makes a copy of this string.
ty..... Window type. Defines the window style and whether the window contents will use backing store. Use one of the pre-defined constants, SIMPLE, DIALOG or DOCUMENT, for the window style. The backing store constants are BACKING and NOBACKING -- the default is NOBACKING. The constants are 'or'-ed to build a complete type.
m..... The window's menu, drawn right under the title bar. Set to NULL for no menu.
d...... User-specific data to attach to the window. This is usually a pointer to a user structure. Send NULL for no user data.

Creating a new window doesn't cause the window to be drawn -- **SM_OpenWindow** does that.

3.8.1.7. int SM_OpenApplication(char *t, MenuType *m)

Initializes the workspace for a TWS application. The main result of this is to initialize and draw the workspace "window" including it's title and menu, if any. This function must be called only once in any application, usually at the beginning. **SM_Init** must be called before calling **SM_OpenApplication**.

t.... Application title, displayed in the application title bar. If NULL the application title bar is omitted and the application menu and workspace content region are enlarged accordingly. If an empty string is passed the application title bar is included but will be empty.
m.. Application menu. Send NULL for no menu, in which case the workspace content region fills the space where the menu would have been.

3.8.1.8. int SM_OpenWindow(WindowType *w)

Displays the window on the screen. **SM_OpenWindow** should be called only once for each window. The window must already be created using the **SM_NewWindow** function.

3.8.1.9. int SM_ReadConfig(void)

Re-read the TWS.CFG or the configuration file given by the TWSCONFIG environment variable. The configuration file is read when the TWS window system is initialized. Re-reading the file would change any window default value specified in the configuration file, while other parameters would be unchanged.

3.8.1.10. void SM_UnmapWindow(WindowType *w)

Removes a window from the window stack but doesn't otherwise affect the window. Redraws the screen with the window missing. The window can be "popped-up" by passing it to **SM_OpenWindow**. This function is useful for creating pop-up dialogs.

```
WindowType      *ErrorDialog;
LabelType       *ErrorLabel;
/*
** 'ErrorDialog' is a window that displays an error message to the user, then
** goes away when the user presses the button. The PopupErrorDialog function
** accepts a string to be displayed and puts it into ErrorLabel, which is a
** gadget that's part of the dialog window. In the application, the sequence of
** events would be something like:
**
**              InitErrorDialog();
**           /* ... */
**              if (error_condition) {
**                      PopupErrorDialog("An error has occurred!");
**              }
*/

LabelType       *ErrorLabel;
WindowType      *ErrorDialog;

/*
** Initialize the error dialog window but don't display it
*/
void InitErrorDialog(void)
{
        RectType        erect;

        erect.Xmin = erect.Xmax = 150;
        erect.Xmax = 350;
        erect.Ymax = erect.Ymin + 75;
        ErrorDialog = SM_NewWindow(&erect, "ERROR!", SIMPLE, NULL, NULL);
        erect.Xmin = erect.Ymin = 5;
        erect.Xmax = SM_GetContentWidth(ErrorDialog) - 5;
        erect.Ymax = erect.Ymin + 25;
        ErrorLabel = SM_CreateLabel(ErrorDialog,
                                &erect,
                                NULL,
                                SM_GetSystemFont(),
                                ALIGNCENTER,
                                False,
                                False,
                                True,
                                NULL);
    erect.Xmin = SM_GetContentWidth(ErrorDialog) / 2 - 50;
    erect.Xmax = erect.Xmin + 100;
        erect.Ymax = SM_GetContentDepth(ErrorDialog) - 5;
        erect.Ymin = erect.Ymax - 25;
        SM_CreateButton(ErrorDialog,
                    &erect,
                    "OK",
                    NULL,
                    CloseErrorDialog);
}

/*
** Set the error dialog string and display the dialog
*/
void PopupErrorDialog(char *str)
{
        SM_SetLabelString(ErrorLabel, str);
        SM_OpenWindow(ErrorDialog);
}

/*
** 'CloseErrorDialog' is the callback function attached to the
** ErrorDialog close button. When the user presses the button, the ErrorDialog
** window goes away, but it still exists.
*/
int CloseErrorDialog(ButtonType *b)
{
        SM_UnmapWindow(ErrorDialog);
}
```

3.8.2. Window Attribute Functions

3.8.2.1. int SM_GetColorScheme() *Obsolete*

Returns the color scheme constant associated with this instance of the windowing system. The color schemes are SMWARM and SMCOOL. The color scheme is established at system initialization.

3.8.2.2. ColorType *SM_GetContentColor(WindowType *w) *(Macro)*

Returns the current background color for a window's content region.

3.8.2.3. int SM_GetContentDepth(WindowType *w) *(Macro)*

Returns the depth in pixels of the content region for the argument window. The depth is Ymax - Ymin + 1.

3.8.2.4. RectType *SM_GetContentRect(WindowType *w) *(Macro)*

Returns a pointer to the rectangle describing the argument window's content region. The rectangle is in device coordinates. This should be treated as read-only data, as changes to the content rectangle will confuse the window manager.

3.8.2.5. int SM_GetContentWidth(WindowType *w) *(Macro)*

Returns the width in pixels of the content region for the argument window. The width is defined as Xmax - Xmin + 1.

3.8.2.6. int SM_GetDefaultBevelDepth()

Returns the width in pixels of the beveling or sculpting on button, window border, etc., rectangle edges. This value can be set in the TWS configuration file.

3.8.2.7. int SM_GetDefaultBorderWidth()

Returns the width of the window borders as stored in the window defaults data structure. All windows use this value for their borders.

3.8.2.8. int SM_GetDefaultMenubarDepth()

Returns the depth in pixels of a window menu bar. The depth of the menu bar is based on the size of the system font. This value can't be changed by the application.

3.8.2.9. ColorType *SM_GetDefaultTextColor()

Returns the color for window text as stored in the window defaults data structure.

3.8.2.10. int SM_GetDefaultTitlebarDepth()

Returns the depth in pixels of a window title bar. The value is calculated at initialization based on the size of the title font. It can't be changed by the application.

3.8.2.11. int SM_GetDisplayDepth()

Returns the device pixel index of the bottom-most screen pixel. For example, on a 640x480 display, *479* is returned.

3.8.2.12. int SM_GetDisplayWidth()

Returns the device pixel index of the right-most screen pixel. For example, on the 640x480 display, *639* is returned.

3.8.2.13. GadgetType *SM_GetGadgets(WindowType *w) *(Macro)*

Returns a pointer to the list of gadgets attached to window *w*. See the chapter on *Gadgets* for details on the GadgetType structure.

3.8.2.14. GraphStateType *SM_GetGraphState(WindowType *w) *(Macro)*

Returns a pointer to the graphics state structure for the argument window. This is a pointer to the actual window graphics state and in most cases should be treated as read-only. See the chapter on Graphics for information on the window Graphics State.

3.8.2.15. FontType *SM_GetSystemFont()

Returns a pointer to a font structure for the current system font. Most window text is drawn using the system font.

3.8.2.16. int SM_GetSystemFontDescent()

Returns the number of screen pixels below the system font's baseline. Not all graphics kernels' font formats support this field, in which case the return value is 0.

3.8.2.17. int SM_GetSystemFontHeight()

Returns the height in pixels for the current system font. The font height is based on the cell dimensions the font characters are drawn in, which may be larger than the actual character pixels.

3.8.2.18. ColorType *SM_GetTextColor()

Returns the current system text color. The text color is one of the 16 system colors.

3.8.2.19. FontType *SM_GetTextFont()

Returns the currently loaded text font. The Text font is presently used for the Textbox gadget.

3.8.2.20. FontType *SM_GetTextFontHeight()

Returns the height in pixels of the current text font. This font is used by default by the Textbox gadget. If a unique font isn't specified in the TWS configuration file *tws.cfg*, then the text font defaults to the system font.

3.8.2.21. int SM_GetTitleFontHeight()

Returns the height in pixels of the current title font. The font height is based on the cell in which the title font characters are drawn, which may be larger than the actual font pixels.

3.8.2.22. void *SM_GetUserData(WindowType *w) *(Macro)*

Returns the user data field from the argument window. This is typically a pointer to a data structure maintained by the application.
Example:

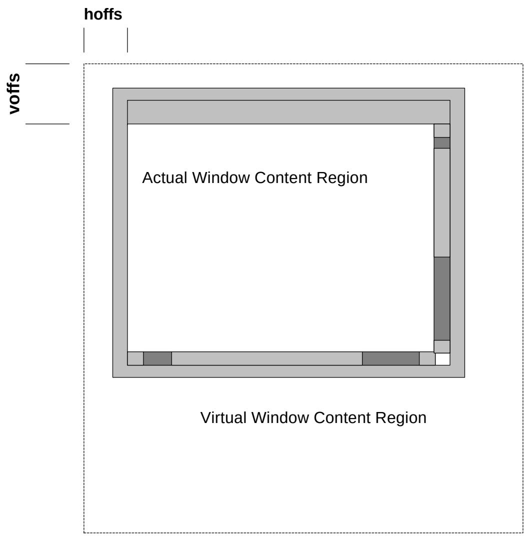
```
MyDataStruct   *mystruct;
mystruct = (MyDataStruct *)SM_GetUserData(w);
mystruct->intval = 1;
```

3.8.2.23. int SM_GetWindowDepth(WindowType *w) *(Macro)*

Returns the depth (Y_{max} - Y_{min} + 1) of the argument window. This is the total depth from border edge - to - border edge. *See also: SM_GetContentDepth; SM_GetWindowWidth; SM_GetCanvasDepth.*

3.8.2.24. void SM_GetWindowOffsets(WindowType *w, int *hoffs, int *voffs)

Returns the current horizontal and vertical translation of the actual content region from the virtual content region. If a window doesn't have the SCROLLATTRIB attribute then the offsets are always 0. For windows with scrollbars, all window elements are always set relative to the virtual content region.



3.8.2.25. int SM_GetWindowWidth(WindowType *w) *(Macro)*

Returns the width (X_{max} - X_{min} + 1) of the argument window. This is the total width from border edge - to - border edge. *See also: SM_GetWindowDepth; SM_GetContentWidth; SM_GetCanvasWidth.*

3.8.2.26. char *SM_GetWindowTitle(WindowType *)

Returns the window title string for the argument window. This is a pointer to the actual string and should be treated as read-only. *See also: SM_SetWindowTitle; SM_NewWindow.*

3.8.2.27. int SM_SetApplicationMenu(MenuType *m)

Sets the application menu to the menu pointer argument. The application menu bar is immediately redrawn to reflect the new menu. *See also: SM_NewWindow; SM_CreateMenu; SM_AddMenuItem.*

3.8.2.28. void SM_SetApplicationTitle(char *title)

Sets the string displayed in the application title bar to *title*. The display is immediately updated to reflect the change. The existing string, if any, is freed.

3.8.2.29. void SM_SetCloseProc(WindowType *w, int (*proc)()) *(Macro)*

Sets the window *w*'s close window management event application procedure to the specified function. This is an application function called when a window is closed and is called before the system frees the window. The function *proc* must be an int function with a single argument, a pointer to a window. When called, this pointer will be *w* and the system will be ready to close *w*. Typically this routine frees application specific data attached to the window.
If the function *proc* returns a non-zero value, the window is not closed and the value gets returned to the event loop as the return value from the **SM_ProcessEvent** function.

3.8.2.30. int SM_SetContentColor(WindowType *w, ColorType *c)

Sets the window argument's content region fill color to the color argument. The color *c* is assumed to be from the system color table. If the window is the active window then it's redrawn immediately. *See also: SM_GetContentColor; Color management.*

3.8.2.31. int SM_SetContentRect(WindowType *w, RectType *r)

Change the size/position of the window's content rectangle to *r*. The coordinates of the rectangle should be in absolute device coordinates. The system recalculates the window borders, title bar, and menu bar based on the new content rectangle size and position. If the window is the active window then the window is immediately redrawn. All window contents -- gadgets, graphics canvas, etc. -- keep their same positions relative to the content origin (upper-left corner).

3.8.2.32. void SM_SetDrawProc(WindowType *w, int (*proc)()) *(Macro)*

Set the window argument's draw window management event procedure to the procedure argument. This is the procedure that draws the window's content region.

3.8.2.33. void SM_SetGadgets(WindowType *w, GadgetType *g) *(Macro)*

Sets the window's gadget list to the GadgetType pointer *g*. Does not check if there is already a gadget list attached to the window -- if one is, it's replaced by *g*. *See also: SM_GetGadgets; SM_AddGadget*.

3.8.2.34. void SM_SetGainFocusProc(WindowType *w, int (*f)()) *(Macro)*

Sets the function called whenever the window *w* becomes the active window to the function *f*. *f* must be an int function with a single argument, a pointer to a WindowType.

3.8.2.35. void SM_SetLoseFocusProc(WindowType *w, int (*f)()) *(Macro)*

Sets the function called whenever the window *w* goes from being the active window to not being the active window. *f* must be an int function with a single argument, a pointer to a WindowType.

3.8.2.36. int SM_SetMenu(WindowType *w, MenuType *m)

Sets the window's menu to the menu argument. If the window is the active window then it is redrawn and the window content region resized to accommodate the menu. *See also: SM_CreateMenu, SM_AddMenuItem*.

3.8.2.37. void SM_SetRedrawFlag(WindowType *w, int flag) *(Macro)*

Sets the redraw flag of the argument window *w* to the value *flag*. Currently the only valid flags are True and False. Set a window's redraw flag to True whenever the window contents have been changed while the window was not the active window. The system resets the redraw flag after drawing the window.

3.8.2.38. int SM_SetRedrawProc(WindowType *w, int (*proc)()) *(Macro)*

Set the window argument's redraw window management event procedure to the function argument. This is the procedure that redraws the window's content region whenever the window is moved but not changed. By default, this procedure is the same as the draw procedure.

3.8.2.39. int SM_SetResizeProc(WindowType *w, int (*proc)()) *(Macro)*

Set the window argument's resize window management event procedure to the procedure argument. This is the procedure that draws the window's content region whenever the window is resized.

3.8.2.40. int SM_SetUserData(WindowType *w, void *d) *(Macro)*

Sets the window's user data field to the pointer argument. *See also: SM_GetUserData*.

3.8.2.41. int SM_SetWindow(WindowType *w)

Causes the window elements to be recalculated based on the size and position of the window's content region rect. The content region is specified in absolute *device* coordinates and can be set using **SM_SetContentRect**. If the window is the active window it is immediately redrawn using the window's resize procedure.

3.8.2.42. int SM_SetWindowProc(WindowType *w, int (*f)()) *(Macro)*

Sets the window procedure to the argument function *f.* The window function is called once on every pass through the event loop as long as the window is active. By default a window has no window function; the application must supply one if necessary.

3.8.2.43. int SM_SetWindowScrollProc(WindowType *w, int which, int (*f)())

Sets the application callback function for the window scrollbar *which* attached to the window *w*. *which* is a constant identifying which scrollbar to attach the function to: constants defined in smtypes.h are VSCROLLBAR and HSCROLLBAR. The function *f* is an int function whose argument is a pointer to a SliderType as described earlier in this section, and in the SliderType gadget section.

3.8.2.44. int SM_SetWindowTitle(WindowType *w, char *c)

Sets the window's title to the string argument. If the window is the active window the title is redrawn. The system makes a copy of the string.

3.8.3. Window Drawing Functions

Normally the window manager takes care of redrawing windows as necessary. For example, when the active window is moved, the window manager automatically redraws any windows that were exposed.
Sometimes an application needs to redraw a window (or part of one) explicitly for some reason. The following functions are provided. *See also: Graphics.*

3.8.3.1. int SM_DrawAppTitle()

Draws the application title at the top of the screen. It is unlikely that an application program would ever have to do this, since this title is established when the workspace is initialized by the **SM_OpenApplication** function , and the title is automatically redrawn whenever the title is changed by **SM_SetWindowTitle**.

3.8.3.2. int SM_DrawBorder(WindowType *w)

Regenerate the borders of the window *w*. Does not affect the window content and doesn't check if the window is the active window. If it isn't, the border for window *w* may be drawn over windows higher in the stacking order. Always returns 0.

3.8.3.3. int SM_DrawWindow(WindowType *w)

Generate the content of an application window by calling the application window's draw procedure. Normally the system handles all drawing and redrawing of a window's contents as necessary (for example, when the window becomes the active window, or some part of the window is exposed). The window must already be opened (see **SM_OpenWindow**) and must be the active window.

3.8.3.4. int SM_EraseContent(WindowType *w)

Clears the content region of the window argument to the content background color. The window must already be opened and must be the active window. *See also: SM_DrawContent.*

3.8.3.5. int SM_MoveWindow(WindowType *w, int dx, int dy)

Move the argument window *w* by *dx* pixels horizontally and *dy* pixels vertically. Positive values move the window to the right and down, negative values to the left and up. The window must be the active window. Returns 0 on success, any other value is an error.

3.8.3.6. int SM_RedrawWindow(WindowType *w)

Redraws the window content by calling the argument window's redraw procedure. The window must be the active window. Normally the window system takes care of redrawing the window contents when necessary, such as when the window is moved or exposed. An application should therefore have rare need to call **SM_RedrawWindow** directly. Returns 0 on success.

3.8.3.7. int SM_RefreshGadgets(WindowType *w)

Causes all gadgets attached to window *w* to be redrawn. The window must be the active window. Returns 0 on success, any other value indicates *w* was not the active window. *See also: Gadgets.*

3.8.3.8. int SM_ResizeWindow(WindowType *w, int dx, int dy)

Changes the window size and redraws the window at the new size. The window size is changed by *dx* pixels horizontally and *dy* pixels vertically in the same manner as **GR_InsetRect** -- positive values reduce the window size in that direction, while negative values expand the window. The window size adjusts symmetrically about its center.

If the window is the active window then it’s redrawn immediately. *See also: GR_InsetRect, SM_SetContentRect, SM_MoveWindow.*

3.8.4. Miscellaneous Functions

3.8.4.1. int SM_BringToFront(WindowType *w)

Makes the window argument the active window and brings it to the top of the window stack. If the window is already the active window then nothing changes. The window must be created and already opened by calls to **SM_NewWindow** and **SM_OpenWindow**.

3.8.4.2. WindowType *SM_FocusWindow()

Returns a pointer to the active window. There is always guaranteed to be a active window anytime TWS is running; if no application windows are open then the workspace window is returned.

3.8.4.3. int SM_FreeWindow(WindowType *w)

Releases all memory associated with the window *w.* Does not call the window's close procedure, if any. This function is mainly used to free temporary windows that have been cast from gadgets. *See also: SM_GadgetToWindow*.

3.8.4.4. int SM_IsFocus(WindowType *w)

Returns True if the window *w* is the active window, False otherwise. *See also: SM_CurrentWindow.*

3.8.4.5. int SM_IsMaximized(WindowType *w) *(Macro)*

Returns True if the argument window has been maximized, False otherwise.

3.8.4.6. int SM_IsMinimized(WindowType *w) *(Macro)*

Returns True if the argument window is minimized (reduced to an icon), False otherwise.

3.8.4.7. int SM_LockWindow(WindowType *w) *(Macro)*

Sets the argument window as 'locked', which means that if the window is the active window, no other window can become active until the window is either unlocked or closed. The window must be created but doesn't have to be open, and doesn't have to be the active window when **SM_LockWindow** is called. If a window is 'locked' while inactive then it doesn't affect program operation until the window becomes active. *See also: SM_UnlockWindow*.

3.8.4.8. int SM_nDisplayBits()

Returns the number of bits per pixel supported in the current display mode. Typically this will be 4 (for standard EGA/VGA), 8 (for extended VGA at 256 colors), 15 (for 32k truecolor systems), or 24 (for 16M truecolor systems).

3.8.4.9. int SM_nDisplayPlanes()

Returns the number of color planes supported by the hardware graphics device in the current display mode.

3.8.4.10. int SM_SaveContent(WindowType *w)

Writes the contents of the argument window's graphics canvas to backing store. The window must be the active window and must have a graphics state and canvas rectangle. Normally the window system takes care of saving a window's graphics canvas as necessary. If the window doesn't already have a backing file, one is created. If the window *does* have a backing file its contents are replaced by the new image. Returns 0 on success, any other value indicates failure.

3.8.4.11. int SM_StringWidth(char *string, FontType *font, int style)

Returns the width in pixels of *string* if it were drawn using the font *font* with facing style *style*. *style* is one of the text style constants TXTNORMAL, TXTBOLD, TXTITALIC, TXTUNDERLINE, TXTSTRIKEOUT, and TXTPROPORTIONAL, which may be ORed together as appropriate.

3.8.4.12. int SM_UnlockWindow(WindowType *w) *(Macro)*

Resets the lock field of the window. This allows other windows in the program to be selected. *See also: SM_LockWindow*.

3.8.4.13. int SM_WindowExists(WindowType *w)

Returns True if the window argument is anywhere on the stack of windows, False otherwise.

3.8.4.14. int SM_WindowRegion(WindowType *w, int x, int y)

Returns a value indicating which of the window decorations the point (*x*,*y*) is in, such as the titlebar, the content region, or one of the window controls. The region macros are in the file smtypes.h; they are:

Macro	*Value*	*Region*
MOVE_REGION	9	Window title bar
CLOSE_REGION	10	Close button
MENU_REGION	11	Window menu bar
BASE_MENU_REGION	12	Application menu bar
WINDOW_REGION	13	Anywhere within the window’s border
MAXIMIZE_REGION	14	Maximize button
CONTENT_REGION	15	Content rectangle
WORKSPACE_REGION	16	Not in the window but inside the workspace content rectangle
MINIMIZE_REGION	18	Minimize button
CANVAS_REGION	19	Graphics canvas rectangle
UPLEFT	5	Upper left drag border
UPRIGHT	6	Upper right drag border
DOWNLEFT	7	Lower left drag border
DOWNRIGHT	8	Lower right drag border

The point (*x*,*y*) is in device coordinates (upper-left corner of the screen is (0,0)). This is compatible with the coordinates returned by the **SM_GetNextEvent** function.
If a window doesn’t have a particular region then that value can’t possibly be returned. For example, a DIALOG window has no drag borders. If **SM_WindowRegion** returns WINDOW_REGION for a DIALOG style window, then the point must be on the border.

Example:

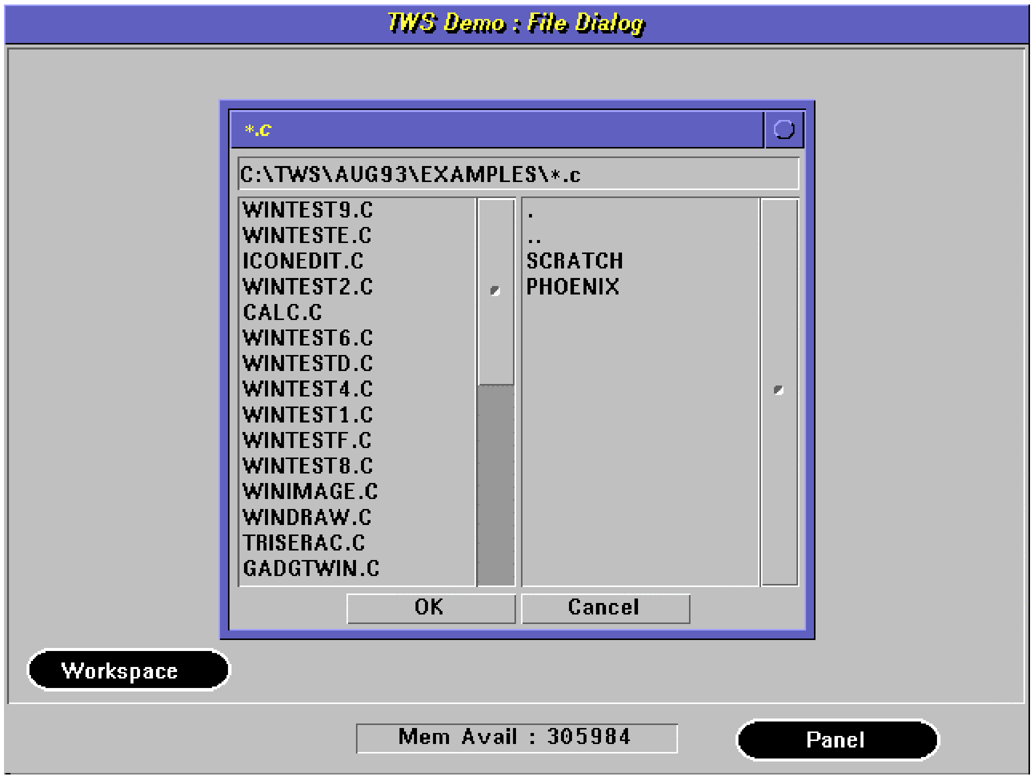
```
#include       <smtypes.h>
#include       <smwindow.h>
char    *regions[] = {“”,””,””,””,””,”UPLEFT”,”UPRIGHT”,”DOWNLEFT”,”DOWNRIGHT”,
                         ”MOVE”,”CLOSE”,”MENU”,”APPLICATION MENU”,”WINDOW”,
                         ”MAXIMIZE”,”CONTENT”,”WORKSPACE”,””,”MINIMIZE”,”CANVAS”};
int            loc;
EventType      event;
```

```
:
:
while (True) {
        SM_GetNextEvent(&event);
        loc = SM_WindowRegion(w, event.CursorX, event.CursorY);
        fprintf(stderr, “Graphics locator is in the %s region\n”,regions[loc]);
}
```

3.8.4.15. WindowType *SM_WorkspaceWindow()

Returns a pointer to the base, or workspace, window, which is always the lowest window in the window stack. The application must call **SM_Init** and **SM_OpenApplication** before using this function.

3.9. Workspace Panels



There is a special kind of application window called a *panel*. For most application purposes, a panel is just like a window. It has its own coordinate system, can contain user interface gadgets, can have its own graphics state, etc.
What makes a panel special is the following:

- Panels are *tiled* with the workspace window. They can't be moved and they can't obscure any other window, including other panels;

- Panels have only a depth and location. A panel's location must be one of UP, DOWN, LEFT or RIGHT. The panel's width is all the way across the location edge. For example, a DOWN panel with depth of 50 extends all the way across the bottom of the screen, and from the bottom of the current workspace to 50 pixels up from the bottom.

- Creating a panel shrinks the workspace content region correspondingly. Since application windows are drawn within the workspace content, application windows can never obscure a panel. This makes panels ideal for real-time, ongoing status displays, like available memory and disk space, time of day clock, etc.

- An application can create panels but must never close them.

Other than the above, a panel is just like any other window (in fact PanelType is defined internally as a WindowType). The following example creates a panel and puts a label into it:

```
#include       <smwindow.h>
```

```
int main(int argc, char *argv[])
{
       PanelType      *panel;
       RectType       r;
       char           memmsg[80];
       int            memleft;

       SM_Init(argc, argv);
       SM_OpenApplication("Panel Demo", NULL);

       panel = SM_CreatePanel(60, DOWN);
       memleft = coreleft();
       sprintf(memmsg, "Mem Avail : %ld", memleft); r.Xmin = 10;
       r.Ymin = 20;
       r.Xmax = r.Xmin
              + SM_StringWidth(memmsg, SM_GetSystemFont(), TXTNORMAL);
       r.Ymax = r.Ymin + SM_GetSystemFontHeight() + 4;
       SM_CreateLabel((WindowType *)panel,
                       &r,
                       memmsg,
                       NULL,
                       ALIGNCENTER,
                       False,
                       False,
                       True,
                       (void *)memleft);
```

It's important to remember what not to do with panels: Never attempt **SM_MoveWindow, SM_SetContentRect, SM_ResizeWindow, SM_CloseWindow,** or **SM_SetWindow** on a panel window. Some functions, like **SM_SetWindowMenu**, will have no noticeable effect because a panel window won't draw a menu even if one is attached to it. The function **SM_CloseAllWindows** ignores panels so can safely be used. All other functions will accept a panel window as an argument.

3.9.1. PanelType *SM_CreatePanel(int size, int location)

Create and initialize a new panel *size* pixels deep at the workspace location *location*. *location* is one of the directional constants defined in smtypes.h; LEFT, RIGHT, UP, or DOWN. *size* is the number of pixels the panel extends in the opposite direction. The pointer returned from **SM_CreatePanel** can be used as the window argument to most TWS functions.

4. The TWS Event System

The TWS graphical user interface is an event-driven system. Events can originate externally in the form of user keystrokes or mouse actions, or internally (clock events). The TWS event manager notices events as they occur and dispatches them to the appropriate objects.
A TWS application is never simply sitting around, waiting for the user to do a specific thing, like entering a file name[2]. Rather, an event driven application asks, “What should I do *if* the user enters a file name?” and provides the appropriate code to respond.
At the top level, the TWS event loop is quite simple:

```
#include        <smevent.h>
EventType       event;
int             msg;

while (True) {
        SM_GetNextEvent(&event);
        msg = SM_ProcessEvent(&event);
}
```

SM_GetNextEvent retrieves a pending event, if any, from the system event queue and copies it into *event*. **SM_ProcessEvent** takes an event and processes it through the window system, window gadgets and menus, and background functions.
SM_ProcessEvent returns a message value. The value depends on how the event was processed. It could be an application-specific value if a gadget or window callback was executed, a window manager event constant if a window management task was done, or simply an echo of the hardware event type.

4.1. Data Types

```
typedef struct _eventtype
{
    uint    Region;                  /* Window region where cursor is    */
    EventFlagsType  Type;            /* Event type (keypress, etc.)      */

    short   ASCII;                   /* ASCII code for keyboard press    */
    short   ScanCode;                /* Scan code for keyboard press     */
    short   ShiftState;              /* Keyboard shift state             */

    unsigned short Button;           /* Which mouse button for press/release*/
    unsigned short ButtonState;      /* Mouse button current state       */
    unsigned short ButtonEvent;      /* Convenience button event type    */
    int     CursorX, CursorY;        /* Hardware Cursor position         */
    int     WinX, WinY;              /* Window content cursor position   */
    int     GraphX, GraphY;          /* Window graph canvas cursor pos   */

    uint    Time;                    /* Sys clock at event               */
    long    localtime;               /* Time of day clock                */

    void   *EventWin;                /* Window the mouse is 'in'         */
    uint    EventMsg;                /* Event return message             */
} EventType;
```

[2]Well, this isn’t *strictly* true. Sometimes the system takes control of the input for specific reasons. For example, while a window is being dragged, all other system activity stops. Same for a pull-down menu. At the application level, you might count a modal dialog as “sitting around waiting.” Kinda depends on your point of view.

4.2. Getting Events

Each time the **SM_GetNextEvent** function is called the EventType argument is updated. Some fields are updated every time, other fields are updated only for certain event types. The fields that are always current are:
Type: This is the hardware-layer event description. The value in this field is always one of the following constants:

Event Constant	*Value*	*Description*
NONE	0	No “active” event occurred.
BUTTONMOTION	0x01	The cursor moved since last queried.
BUTTONPRESS	0x02	A mouse button was pressed. The *Button* field contains which button was pressed.
BUTTONRELEASE	0x04	A mouse button was released. The *Button* field contains which button was released
KEYPRESS	0x08	A keyboard key was pressed. See the *ASCII* and *ScanCode* fields to find out which one.
DRAGMOTION	0x80	The cursor moved while at least one mouse button was active. The *ButtonState* field shows which buttons are active.
CLOCKEVENT	0x20	A ‘tick’ from the clock timer occurred.

BUTTONPRESS and BUTTONRELEASE events have priority over BUTTONMOTION and BUTTONDRAG events. CLOCKEVENT events have the highest priority; however, if a button or key event occurs at the same time as a clock event, TWS buffers the button event and picks it up on the next pass. The system doesn’t allow the possibility of simultaneous key and mouse events, but will queue them in sequence.
ShiftState: Shows the current state of the keyboard *shift keys*, such as Shift, Alt, Num-Lock and so forth. Each bit in the field corresponds to a key state, described by the following constants. The constants ending in ...DOWN indicate a state transition occurred for that key. Those ending in ...ON indicate that the mode is active. Test for more than one condition by ORing the constants as needed. The following example sets a trigger if the right and left shift keys are both pressed:

```
SM_GetNextEvent(&event);
trigger = event.ShiftState & (RIGHTSHIFTDOWN | LEFTSHIFTDOWN);
if (trigger) then {
        ...
```

State constant	*Value*	*Definition*
RIGHTSHIFTDOWN	0x0001	The right shift key is pressed
LEFTSHIFTDOWN	0x0002	The left shift key is pressed
CTRLDOWN	0x0004	The CTRL key is pressed
ALTDOWN	0x0008	The ALT key is pressed
SCROLLOCKON	0x0010	SCROLL LOCK is active
NUMLOCKON	0x0020	NUMLOCK is active
CAPSLOCKON	0x0040	CAPS LOCK is active
INSERTON	0x0080	INSERT is active
LEFTCTRLDOWN	0x0100	The left CTRL key was pressed

LEFTALTDOWN	0x0200	The left ALT key was pressed
SYSREQDOWN	0x0400	The SYSREQ (Print Screen) key is pressed
CTRLNUMLOCK	0x0800	The CTRL-NUMLOCK sequence is active
SCROLLOCKDOWN	0x1000	The SCROLL LOCK key was pressed
NUMLOCKDOWN	0x2000	The NUMLOCK key was pressed
CAPSLOCKDOWN	0x4000	the CAPS LOCK key was pressed
INSERTDOWN	0x8000	The INSERT key was pressed

ButtonState: The current state (pressed or not) of the mouse buttons. The state of the mouse button isn't related to a mouse button event (BUTTONPRESS or BUTTONRELEASE). Any combination of the following constants are possible in this field. For example, to perform some action as long as the right mouse button is held by the user,

```
while (1) {
        SM_GetNextEvent(&event);
        if (event.ButtonState & RIGHTMOUSEBUTTON) {
                ...
        }
}
```

ButtonState Constant	*Value*	*Description*
LEFTMOUSEBUTTON	0x10	The left mouse button is pressed.
RIGHTMOUSEBUTTON	0x20	The right mouse button is pressed.
MIDDLEMOUSEBUTTON	0x40	The middle mouse button is pressed. This value can't be returned for a two-button mouse.

CursorX*, *CursorY: This is the screen position for the mouse cursor in absolute device coordinates.
Time: This is the value of the system timer. The system timer is a continuously-running counter that increments 18.2 times per second. The values repeat every 30 minutes or so. This has no relationship to the time of day. It's principle use is to determine if two events occur sufficiently close together (or far apart). The following example determines that a double-click has occurred if two button presses come within a half-second of each other* :

```
firsttime = 0;
while (1) {
        SM_GetNextEvent(&event);
        if (event.Type == BUTTONPRESS)  {
            if (firsttime) {
                if (abs(firsttime - event.Time) < 9) {
                    doubleclick = True;
                }
                firsttime = 0;
            }
            else {
                firsttime = event.Time;
            }
        }
}
```

* Note that this is a necessary but insufficient condition for detecting traditional double-clicks. In addition, we would have to see that the cursor position moved less than some delta amount, and that the same button was involved in both presses.

For event *Type*s of NONE or CLOCKEVENT, the fields above are the only ones updated. The remaining fields are not modified at all so retain their values from the last time they were set. This can be useful as well. For example, the *Region* field is updated when the cursor moves into a particular region of a window. If the cursor is never moved afterward, the *Region* field will not change, and it retains the correct value.

Any other events cause the remaining fields to be updated too. These fields are described below:

Region: The window area where the mouse cursor was when the event occurred. Only areas of the active window and the workspace are tested. If the cursor is over an inactive window, this field will contain the WORKSPACE_REGION constant. *Region* values are always the most specific possible. For example, the CANVAS_REGION is within the CONTENT_REGION, which is inside the WINDOW_REGION, so the value will only be CONTENT_REGION if the cursor is inside the window content region and *not* in the canvas region.

Region Constant	***Value***	***Description***
UPLEFT	5	Active window upper-left resize handle
UPRIGHT	6	Active window upper-right resize handle
DOWNLEFT	7	Active window lower-left resize handle
DOWNRIGHT	8	Active window lower-right resize handle
MOVE_REGION	9	Active window title bar (moves window when grabbed)
CLOSE_REGION	10	Active window close button
MENU_REGION	11	Active window menu
BASE_MENU_REGION	12	Application menu (workspace)
WINDOW_REGION	13	Inside the active window border
MAXIMIZE_REGION	14	Active window maximize button
CONTENT_REGION	15	Inside the active window content region
WORKSPACE_REGION	16	In the workspace content
MINIMIZE_REGION	17	Active window minimize button
CANVAS_REGION	18	Active window graphics canvas
APP_TITLE_REGION	19	Application (workspace) title bar

EventWin: The window the cursor was in when the event occurred. This can be any window, including the workspace and icons. Remember, though, that the window in the *EventWin* field is not necessarily the window that the *Region* values came from; *Region* values are always relative to the active window.

WinX*, *WinY: Mouse cursor coordinate relative to the content region of the active window. Since the cursor is not necessarily in the active window, these values could be negative, or greater than the content boundary.

GraphX*, *GraphY: Mouse cursor coordinate relative to the active window's graphics canvas. The cursor isn't necessarily inside the graphics canvas, so these values may be outside the canvas boundary. If the window doesn't have a graphics canvas then these two values are undefined.

ASCII*, *ScanCode: For KEYPRESS events, contain the character and keyboard scan code, respectively, for the key(s) pressed. For a normal alphanumeric key, *ASCII* contains the character and *ScanCode* is ignored. If *ASCII* is 0 then *ScanCode* contains the code for a special key or key combination, such as a function key, Home key, or Alt-key combination.

localtime: Only valid for the CLOCKEVENT event type, this is the number of system clock 'ticks' since midnight. The system ticks 18.2 times per second. The TWS function **SM_TimeOfDay** converts this value to the time in hours, minutes and seconds. This value is not updated for any other event types.

Button: For BUTTONPRESS and BUTTONRELEASE events, contains which button was used. Uses the same constants as the *ButtonState* field described earlier. Not updated for any other event types. Note that this is the button that made the transition; the button will be one of the active buttons in the *ButtonState* field only for a BUTTONPRESS event.
ButtonEvent: This is a convenience field which reports button press, release, and drag events in terms of which button was involved. It's created by ORing the *Type* field with the *Button* or *ButtonState* fields. The resulting values are captured in the constants shown below. For example, something like the following captures a middle-button press:

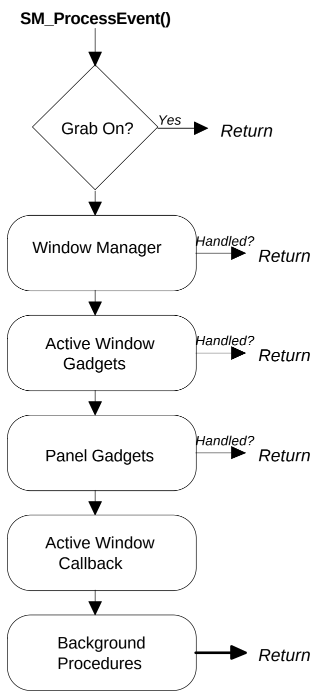
```
SM_GetNextEvent(&event);
if (event.ButtonEvent == MIDDLEBUTTONPRESS) {
        DoSomething();
}
```

ButtonEvent Constant	***Value***
LEFTBUTTONPRESS	0x12
RIGHTBUTTONPRESS	0x22
MIDDLEBUTTONPRESS	0x42
LEFTBUTTONRELEASE	0x14
RIGHTBUTTONRELEASE	0x24
MIDDLEBUTTONRELEASE	0x44
LEFTBUTTONDRAG	0x81
RIGHTBUTTONDRAG	0x82
MIDDLEBUTTONDRAG	0x84

This value is updated only for BUTTONPRESS, BUTTONRELEASE, and DRAGMOTION event types.

4.3. TWS Event Handler

After getting an event with **SM_GetNextEvent** the application typically passes it on to the **SM_ProcessEvent** function. This function is the workhorse of the event system, passing the event to window manager event handlers, window gadget event handlers, window procedures and background procedures as appropriate for a particular event.



The integer value returned by **SM_ProcessEvent** depends on how the event was handled. Generally there are four sources for event processing return values[3] :

Window Procedures: A window procedure (the Active Window Callback in the diagram) is an application function that can be attached to a window. If the active window has a window procedure, **SM_ProcessEvent** calls it. If a window procedure is called, **SM_ProcessEvent** returns its return value.

Gadget callback functions: For most gadgets the application supplies a function that is called whenever the user activates the gadget. This function is known as a callback. If an event triggers a gadget, **SM_ProcessEvent** returns the value returned by the gadget's callback function. Only the active window and panel window gadgets can be triggered.

Window manager events: When the active window is moved, resized, etc., **SM_ProcessEvent** returns a constant value identifying what kind of window management was performed. The constants are:

Event Constant	*Value*	*Description*
EV_WINMOVE	101	The active window was moved.
EV_CHANGEWIN	104	The active window was changed, probably by bringing a lower window to the top.
EV_NEWWIN	105	A new window was created.
EV_CLOSEWIN	106	A window was closed. This may result in a new active window.
EV_WINSIZE	107	The active window was resized.
EV_MINIMIZE	102	The active window was minimized (reduced to icon).
EV_MAXIMIZE	103	The active window was maximized.

[3] These values are often called *messages* by other user interface systems. Since *TWS* has a separate unrelated messaging system, we use the more generic term of *'value'*.

EV_RESTORESIZE	108	The active window was restored to its normal size from an icon or maximized state.
EV_CHANGEFOCUS	109	There was a change in the focus gadget in the current active window.

An application can get more information about the results of a window management event by querying the window manager or the active window. For example, an application can discover which window became the active window by watching for EV_CHANGEWIN events, then calling **SM_FocusWindow** to get a pointer to the new active window.
Menu selection: Selecting from a window or the application menu usually ends up calling an application function at some point. If the event is handled by the menu system, the final menu return value is passed back through the **SM_ProcessEvent** return value. In the diagram, menu selections are included in the window manager events block.
Hardware events: This is simply a copy of the *Type* field from the event struct itself. This means that something happened, like a button press or keyboard press (or nothing at all if the value is 0), but nothing in the application or window manager was interested in it.

As soon as the event has been handled, **SM_ProcessEvent** returns. A non-zero return value for a gadget or window manager event (like a menu) indicates that the event was handled. Event grabs are a little different; see the chapter on Event Grabs that follows.
By choosing callback function return values carefully, the application can accurately track local (gadget and window-level) activities at a global (application event loop) level while maintaining strict modularity. The application can even defer processing back to the event loop by designing callbacks to *only* return unique values, then testing those values in the main loop. Some other window manager systems use this method extensively as it opens the door for applications that can be modified at runtime using resource files. Specific details are beyond the scope of this document.
To prevent confusion between application and window manager message values, application values should always be greater than 256.
SM_ProcessEvent only returns one value for any one event. In cases where an event could be interpreted more than one way the following rules are used:

- Window manager handlers, menus, and gadgets cannot conflict since they're all in separate screen areas;

- If an application menu (attached to the workspace) and the active window's menu have accelerator keys in common, the active window has priority. See the Menus chapter for more details;

- If gadgets overlap, the gadget most recently created gets the first opportunity to process an event;

- If the active window has a window procedure, that function is called almost *every* pass through **SM_ProcessEvent**, and is called even if other handlers have already responded to the event. When the window procedure gets the event, the *EventMsg* field contains the code for whatever handling was already done with the event, or 0 for no prior handling. This includes codes for window manager events, gadget and menu callbacks. So the window procedure mechanism provides yet another way to provide extra global response to local event handling. However, realize that at this point the prior local processing is already complete (on a window resize, for example, the window has already been redrawn and the application resize procedure, if any, has been called).

4.4. Background Procedures

A background procedure is similar to a window function except that a background procedure may be called even when its parent window isn't active. Processes that can or should happen in the background, such was printing large files or displaying a time of day clock, are good candidates for background procedures.
The data structures for background procedures look like the following:

```
typedef int (*BackgroundProcType)(WindowType *, EventType *) ;
typedef unsigned char EventFlagsType;
typedef int BackgroundProcIDType;

/*
** Application event handler list node
*/
typedef struct _enode {
    EventFlagsType  events;             /* Types of events that trigger     */
    WindowType      *w;                 /* Parent window of this function   */
    int             (*evfunc)(WindowType *w, EventType *e);
} BackgroundProcNodeType;
```

An array of `BackgroundProcNodeType`-s is initialized by the application and contains all the background procedures for a program. The *events* field stores the kinds of system events the procedure should respond to. This is tested at the system level before the *evfunc* (the application-supplied background function) is called, so the function doesn't have to test this itself. The window *w* is the 'parent' window for the function. *evfunc* is the application function that is called when the background procedure is triggered. It takes two arguments, a window pointer and an event pointer. The window passed to the background procedure is *w*, the parent window of the procedure. The event *e* is the event that occurred which triggered the background procedure call.
A function is added to the background procedure list with the **SM_RegisterBackground-Procedure** function. This function is passed a "parent" window for the background procedure, the application procedure to be called, and the event type(s) that must occur for the background procedure to be called:

```
WindowType     *w;
int            MyHandlerFunction();

id = SM_RegisterBackgroundProc(w, MyHandlerFunction, CLOCKEVENT);
```

Once added, the background procedure is called once on every pass through the event processing loop in the **SM_ProcessEvent** function if the type of event matches the event type(s) the background procedure is interested in. The background procedure is passed a pointer to its parent window, and a pointer to the EventType struct that was passed to the **SM_ProcessEvent** function. Note that the background procedure receives control *after* window management (moving, resizing, etc.), window gadget, and window callback procedures have had a crack at the event.
Nothing else can happen while the background procedure is executing so processing should be done as quickly as possible.
The following additional event constants that are especially useful when registering a background procedure. These give control over when the background procedure can be called based on the state of the handler's parent window. They are:

Event Constant	***Description***
ACTIVEONLY	Call the background procedure only if its parent window is active
INACTIVEONLY	Call the background procedure only if its parent window is not active

NOTOBSCURED	Call the background procedure only if its parent window is completely visible. If the parent window is active then it can't be obscured and the handler function is called. If the parent window is not active then the handler function is called only if the parent window is not covered at all by any other window.
ALLEVENTS	The handler doesn't care what the event is, call it regardless.

Background procedures aren't limited to these event types -- a background procedure can be registered to respond to any event type. For example, you could write a background procedure that is called when a mouse button is pressed. The procedure would then be called every time the button is pressed, regardless of which window is active.
Event types can be ORed as necessary and as makes sense, for example:

```
int MyHandlerFunction(WindowType *w, EventType *evnt);
/*
** ....
*/
id = SM_RegisterBackgroundProc(w, MyHandlerFunction, ALLEVENTS | NOTOBSCURED);
```

This will cause the system to call **MyHandlerFunction** on every pass through the event loop as long as the window *w* is not covered by another window. This is useful when you want to draw into an inactive window, since the TWS system doesn't clip obscured windows properly.
It's important to remember to unregister background functions when they're no longer needed. A handler event can unregister itself.

4.4.1. Background Procedure Example

The following example shows how an application could set up a 'Print' button to print a large file in the background. We assume the existance of a user function called **PrintBuf** which outputs a block of text to the printer. We also assume that the application has attached a pointer to an Editstring gadget to the button's data field, and that this string contains the name of the file to be printed.
As is common in TWS applications, we define a structure to contain application-specific information:

```
typedef struct {
        FILE                  *printfile;
        BackgroundProcIDType   bgid;
        /* anything else that's necessary */
} WindowData;
```

When the application creates the window that contains the 'Print' button, it must attach a pointer to an instance of our WindowData struct to it:

```
WindowType            *printdialog;
WindowData            data;
ButtonType            *printbutton;
RectType              dialogrect, brect;
EditstringType *edstr;
/*
** Code to initialize the 'data' struct and dialog rectangle omitted ...
** Create the print dialog window and attach 'data' to it
*/
printdialog = SM_NewWindow(&dialogrect,
                           "Print",
                           DIALOG,
                           NULL,
                           (void *)&data);
/*
** Code to create the editstring and set up a rectangle for the button omitted ...
** Create the button to start printing and attach the editstring
** and print callback function to it
*/
printbutton = SM_CreateButton(printdialog,
                              &brect,
                              "Print",
                              (void *)edstr,
                              PrintButtonCallback);
```

The ‘Print’ button’s callback is set up:

```
int PrintButtonCallback(ButtonType *b)
{
       EdiststringType    *string;
       WindowData         *data;
       WindowType         *w;

       string = (EditstringType *)SM_GetButtonData(b);
       if (string) {
           /*
           ** Open the file and attach the file pointer to the ‘window data’
           ** structure. We assume that this is an application data structure
           ** that holds the file to be printed, among other things
           */
           w = SM_GetGadgetWindow(b);
           data = (WindowData *)SM_GetUserData(w);
           data->printfile = fopen(SM_GetEditstringString(string),"r");
           /*
           ** Register a background procedure to do the actual printing.
           ** Save the id of the background procedure so we can unregister it later
           */
           data->bgid = SM_RegisterBackgroundProc(w, PrintProc, ALLEVENTS);
           return PRINTING;
       }
       return PRINTERROR;
}
```

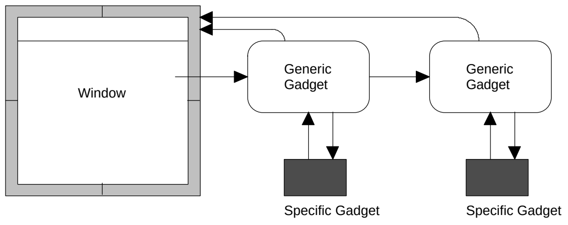
The background procedure itself is equally simple:

```
int PrintProc(WindowType *w, EventType *event)
{
       WindowData     *data;
       unsigned char  buffer[512];
       int            nbytes;

       data = (WindowData *)SM_GetUserData(w);
       if (nbytes = fread(buffer, 1, 512, data->printfile)) {
              PrintBuf(buffer, nbytes);
       }
       else {
              /*
              ** The read returned 0, either end of file or error.  Close the
              ** file and remove this background procedure
              */
              fclose(data->printfile);
              SM_RemoveBackgroundProcedure(data->bgid);
       }
       return 0;
}
```

A couple of points about the example and background procedures in general:

- Notice how the window is the common element for passing data around in the application. Windows know about the gadgets that are attached to them, but gadgets likewise know what window they’re attached to. This circular referencing means that the parent window of an object is always accessible from the object.



The illustration shows the relationship between windows and gadgets.

- The window argument to the background procedure has nothing to do with the active window on the screen. In fact the window doesn't even have to be displayed. In large part, the window argument is a mechanism for passing data to the background procedure (e.g., the 'WindowData' in the example);
- Background procedures can remove themselves. This is perfectly normal and even common behavior. Remember, though, that the background procedure itself doesn't know what its ID is, so the application must design in a way to tell it. This is done through the 'WindowData' variable in the example;
- The **PrintButtonCallback** function returned values indicating success or failure. This isn't success or failure of the print job itself, only whether or not there was a filename string available. In real life, an application would perform a lot more error checking than this. The value will eventually find its way back to the main event loop as the return value from the **SM_ProcessEvent** function;
- The TWS Messaging system is useful for communicating *from* background procedures *to* the rest of the application, or for any asynchronous state changes in the system.
- Background procedure return values are ignored. So how could a background procedure tell the main program that it has finished its work? The TWS messaging system is one way. By defining some message and id constants:

```
#define MAINPROC       257
#define PRINTBKGDPROC  258
#define PRINTSTART     1001
#define PRINTDONE      1002
#define PRINTERROR     1003
```

and adding message posting and retrieving in the proper places, parts of the program can broadcast what they are doing, and other parts of the program can tune in. For example, the background procedure could post a message when printing is done:

```
*/
fclose(data->printfile);
SM_RemoveBackgroundProcedure(data->bgid);
SM_PostMessage(PRINTDONE, PRINTBKGDPROC, MAINPROC, NULL);
```

The application can then watch for PRINTDONE messages. Let's assume the MAINPROC is where the main application event loop is:

```
MessagePacketType     msgpacket;

while (1) {
    SM_GetNextEvent(&event);
    SM_ProcessEvent(&event);
    if (SM_GetMessageFor(MAINPROC, &msgpacket)) {
        switch(SM_GetMessageType(&msgpacket)) {
            case PRINTDONE :
                SM_OKDialog("Print job is done");
                break;
            case PRINTERROR :
                ....
```

Alternative methods include setting global flags, or setting flags in global data structures and periodically polling those values. See the Messaging chapter for more information.

4.5. Event Grabs

Sometimes an application program needs to process events with minimal intervention from the standard event processor. A simple example would be to allow the user to click on any portion of the screen without changing the active window. One way to do this would be to call **SM_GetNextEvent** but never call **SM_ProcessEvent**. While this works, it also causes all background processing to stop until the next **SM_ProcessEvent** call.
Another way is to set an event grab. A *grab* is simply a message that tells the event manager not to process the event itself because the application will take care of it. However, background procedures *are* called. An application sets pointer and keyboard grabs using **SM_SetMouseGrab** and **SM_SetKeyboardGrab**.
The mouse or keyboard is either grabbed or not — nesting or hierarchies of grabs aren't supported. If a grab is already set then setting it again has no affect. Setting and releasing event grabs is global to an application.

4.6. Printscreen Event

TWS provides special handling for a Printscreen event. The user triggers this event by pressing the Control and Printscreen keys simultaneously (Ctrl-PrntScrn). The application can register a special function with the event system to handle this. The printscreen event handler function is a void function that takes three arguments: a pointer to a WindowType, a pointer to a RectType, and a character string. The application registers this function and the arguments to pass to the function using the **SM_SetPrintscreenProc** function:

```
#include       <smwindow.h>
#include       <smtypes.h>
void PrntScrn(WindowType *w, RectType *r, char *fname);
/* ... */
SM_SetPrintscreenProc(NULL, NULL, "winscreen.pcx", PrntScrn);
```

As shown above, if the printscreen function isn't interested in any particular window or rectangle region, those parameters can be NULL. In a typical robust application, the PrintScreen procedure would probably open a dialog prompting for various information -- a file name, printer or file type, etc., but in fact there are no restrictions on what the function may do.
The default handling for the Printscreen event is to do nothing. TWS provides several utility routines for handling screen printing in particular ways -- see the Utilities chapter for information.

4.7. Interface Functions *smevent.h*

4.7.1. int SM_CallBackgroundProcs(WindowType *w, EventType *evnt)

Causes the background procedure manager to scan through the pending background procedures and call the ones whose parent window and event mask fields are suitable to window *w* and event *evnt* (see the discussion above). It would be unusual for an application to call this function. Returns the function return value from the last background procedure executed (more than one background procedure could be executed).

4.7.2. int SM_GetEventButtons(EventType *event)

Returns the *buttons* field from the event structure argument.

4.7.3. void SM_GetEventCursor(EventType *event, int *x, int *y)

Returns the x and y cursor location in global device coordinates from the event structure argument.

4.7.4. void SM_GetEventKey(EventType *event, char *Ascii, char *scancode)

Returns the character and scan code for the event structure argument.

4.7.5. int SM_GetEventRegion(EventType *event)

Returns the *region* field from the event structure argument.

4.7.6. int SM_GetEventState(EventType *event)

Returns the *state* field from the event structure argument.

4.7.7. long SM_GetEventTime(EventType *event)

Returns the time of day in the format of **SM_GetSystemTime** from the event structure argument.

4.7.8. unsigned char SM_GetEventType(EventType *event)

Returns the *type* field from the event structure argument.

4.7.9. int SM_GetEventWinevent(EventType *event)

Returns the *manage_event* field from the event structure argument.

4.7.10. int SM_GetMouse(WindowType *w, int *state, int *x, int *y)

Returns the current mouse location, relative to the content region of window *w,* in *x* and *y*, and the mouse button state in *state*. The coordinates (*x,y*) are returned relative to the window *w*, so may be negative. Mouse states are:

ButtonState Constant	*Value*	*Description*
LEFTMOUSEBUTTON	0x10	The left mouse button is pressed.
RIGHTMOUSEBUTTON	0x20	The right mouse button is pressed.
MIDDLEMOUSEBUTTON	0x40	The middle mouse button is pressed. This value can't be returned for a two-button mouse.

The constants are defined in smevent.h.
SM_GetMouse returns True if the point (*x,y*) is within the window, False otherwise. *See also: GR_GetMouse*.

4.7.11. EventType *SM_GetNextEvent(EventType *evnt)

Retrieve an event from the system event queue. The event parameters are copied into the *evnt* struct, which is then processed by the **SM_ProcessEvent** function.

4.7.12. long SM_GetSystemTime()

Returns the current value of the system time of day clock in a packed format. The values for the hour, minute and second can be extracted using the **SM_TimeOfDay** function.

4.7.13. int SM_InitBackgroundProcedures()

Initializes the background procedure internal data structures. Background procedures cannot be registered before this function is called. Must be called at most *once* per program.

4.7.14. int SM_ProcessEvent(EventType *evnt)

Performs actions based on the event argument. **SM_ProcessEvent** determines the type of event and branches accordingly. All system events, including window moving and resizing, menus and so forth are handled through this function. The return value is either a user interface event constant, a window manager event constant, or the return value of an application function.

4.7.15. BackgroundProcIDType SM_RegisterBackgroundProcedure (WindowType *w, BackgroundProcType f, EventFlagsType fl)

Register an application function as a background procedure. *w* is the 'parent' window for the background procedure, *f* is the function to be added to the background procedure list, and *fl* is the event types the procedure is interested in.

4.7.16. int SM_RemoveBackgroundProcedure(BackgroundProcIDType id)

Takes the background procedure with id of *id* off the background procedure list. Usually the background procedure will call this function to remove itself when it is no longer needed. *id* is the identifier returned from the **SM_RegisterBackgroundProc** function. This effectively "halts" the background process.

4.7.17. int SM_TimeOfDay(long time, int *hr, int *min, int *sec)

Given a *time* value as returned by **SM_GetSystemTime**, returns the hour, minute and second. The function return value is the *hr* value. All values begin at 0 -- hour values are 0 - 23, minute and second values are from 0 - 59.

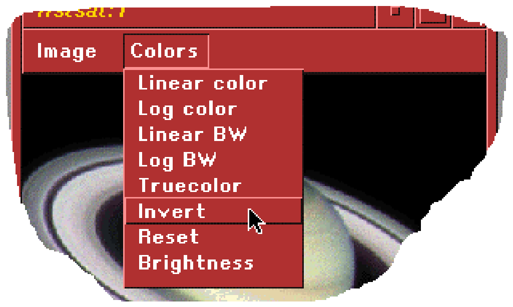
4.7.18. void SM_SetPrintscreenProc(WindowType *w, RectType *r, char *fname, void (*f)())

Sets the function, and the window, character string and screen region passed to the function, whenever the Ctrl-PrntScrn key combination is pressed. The rectangle *r* is in global device coordinates.
When the user presses Ctrl-PrntScrn, the function *f* is called with three parameters -- the window *w*, the rectangle *r*, and the string *fname*. Any or all of the parameters may be NULL, and in fact the function is not required to pay any attention to them.

```
#include       <smwindow.h>
#include       <smtypes.h>
void PrntScrn(WindowType *w, RectType *r, char *fname);
/* ... */
SM_SetPrintscreenProc(NULL, NULL, "winscreen.pcx", PrntScrn);
```

TWS provides some standard Printscreen event functions that can be called from the application function *f*. See the Utilities chapter for more details.

5. Menus



TWS supports one menu type, a pull-down menu bar immediately under the window title bar. The menu bar is always the full width of the window. Items are displayed in the menu bar from left to right in the same order they were added to the menu. Items are separated by the menu-spacing value in the window defaults data structure. If there are more items in a window or workspace menu than will fit, the items are clipped at the right edge of the menu bar.
Each menu item on a window menu can either trigger an application callback function or a pull-down menu. Each item of a pull-down menu can trigger either a user function or another pull-down menu, etc., etc.
A window can only have one menu. The menu can be activated only when the window has the focus. The workspace window can also have a menu, which can be selected any time. The workspace can only have a single menu as well. The width of a pull-down submenu's display box is determined by the longest item label at the time the menu is drawn.
A window menu item is selected by pressing the left mouse button over the menu label, or by pressing the optional Alt-key accelerator (described later). The selected menu item is indicated by a recessed rectangle around the label. If the item triggers a pull-down menu, the pull-down menu is drawn directly under the window menu item, aligned with the left edge of the recessed rectangle. As the mouse cursor is moved over a pull-down menu, the item under the cursor is indicated by a raised rectangle. When the left mouse button is pressed over an item the rectangle changes to recessed.
Unlike other window gadgets, menus are *not* destroyed when the window is closed. Once a menu has been allocated and built it is not released unless the application specifically releases it. This means one set of menus can be applied to any number of different windows. It also means that changing a menu item changes that item in all windows that use that menu.

5.1. Data Types

```
typedef struct _menu {
    int     type;            /* Type of menu (action or submenu) */
    char    *label;          /* Menu label                       */
    int     (*menuproc)();   /* Function if action item          */
    struct _menu *submenu;   /* Submenu if submenu item          */
    struct _menu *next;      /* Next item at this menu level     */
    RectType region;         /* Global boundary for this item    */
    int     active;          /* If False menu item unavailable   */
    short   altkey;          /* Key code that activates menu     */
    char    key;             /* Normal-key equivalent for altkey */
} MenuType;
```

The *type* field is a constant describing the type of the menu item, either an ACTION_ITEM, SUBMENU_ITEM or MENU_DIVIDER (these constants are defined in smmenu.h). *label* is the text displayed for the menu item. If the menu is an ACTION_ITEM then *menuproc* is the application function called when the menu is selected; otherwise it can be NULL. If the menu item is a

SUBMENU_ITEM then *submenu* is a pointer to the next menu level; otherwise it's NULL. *region* is the rectangle surrounding the menu item's sensitive region. This area is calculated when the menu is activated -- when the menu isn't active the value for *region* is unreliable. *active* is True if the menu item can be selected by the user (the normal case), or if False the item can't be selected. If an item isn't active it's displayed in a grayed manner, and an attempt to select the item fails.
A MENU_DIVIDER item is a sculpted line used to separate related groups of menu items. It cannot be selected. The values for *label*, *menuproc* and *submenu* are irrelevant for a MENU_DIVIDER menu item.

5.2. Menu Accelerator Keys

A window menu item can be selected via an Alt-key combination at the keyboard *if* the application sets it up to do so. The application sets which Alt-key combination will be used by prefixing a letter in the menu label string with an ampersand. The ampersand will be removed and the menu character will be drawn with an underline. Pressing the Alt-key combination for a window menu has the same effect as clicking on the item with the mouse.

```
MenuType       *mainmenu, *filemenu;
mainmenu = SM_CreateMenu();
filemenu = SM_AddMenuItem(mainmenu, SUBMENU_ITEM, “&File”, NULL, filemenu);
```

The above example creates a menu *mainmenu* and adds the item “File” to it. When drawn, the capital F will be underlined, and when a window containing the menu is active, pressing Alt-f on the keyboard will activate the filemenu's pull-down submenu.
Items on pull-down menus can also be selected by keyboard key, and the key is chosen the same way. For pull-down menus, either the Alt-key or the normal key can be used:

```
SM_AddMenuItem(filemenu, ACTION_ITEM, “Save &as”, SaveWithNewName, NULL);
```

The statement above adds an item to the *filemenu* menu from the previous example. When the *filemenu* pull-down menu is displayed, the ‘a’ in *as* will be underlined, and the menu function can be activated by pressing Alt-a, or just a, on the keyboard.
Since only one window can be active at a time, menu Alt-keys for different window menus don't necessarily have to be unique. However, the application menu is always active, and if an active window menu Alt-key is the same as an application menu Alt-key, the active window has precedence.
Also, note that if the application wants to use Alt-key combinations, it must *not use* menu accelerator keys. Alt-keys for menus are handled in the event processor function **SM_ProcessEvent**, and the menus set a keyboard (and mouse) event grab on all pull-down submenus. It would be very difficult to design an application that allowed menus and applications to share the same Alt-key events.
By default, menu items have no key assigned to them and cannot be selected via the keyboard. Pressing the Escape key closes all pull-down menus whether or not the Alt-keys are enabled.

5.3. Interface Functions *smmenu.h*

5.3.1. MenuType *SM_AddMenuItem(MenuType *m, int type, char *s, int (*f)(), MenuType *sub)

Adds a new item to the end of the menu *m*, which is the head of a list of menu items returned by **SM_CreateMenu**. The type of menu item is *type*, either ACTION_ITEM or SUBMENU_ITEM. The item's label is *s.* If the menu item is an ACTION_ITEM then *f* is the application function called when the menu item is selected; otherwise it can be NULL. If the menu item is a SUBMENU_ITEM then *sub* is the next menu level, otherwise it can be NULL.

Returns a pointer to the new menu item.

5.3.2. MenuType *SM_CreateMenu()

Creates a new blank menu and returns a pointer to it.

5.3.3. int SM_DestroyMenu(MenuType *m)

Frees a menu and all resources associated with it. If a menu has submenus then all the submenus are freed as well.

5.3.4. int SM_GetMenuActive(MenuType *m)

Returns True if the menu item *m* is active (selectable), False otherwise.

5.3.5. char *SM_GetMenuLabel(MenuType *m)

Returns the character string label for the menu item *m.* This is a pointer to the label string itself, not a copy.

5.3.6. void SM_SetMenuActive(MenuType *m)

Makes the menu item *m* active. If it's already active then nothing happens.

5.3.7. void SM_SetMenuInactive(MenuType *m)

Makes the menu item *m* inactive. If it's already inactive then nothing happens. Inactive menu items are displayed in a grayed manner and can't be selected.

5.3.8. int SM_SetMenuLabel(MenuType *m, char *label)

Sets the label of menu item *m* to *label*. The existing menu label is discarded. *label* is copied to the menu structure so doesn't have to be static.

5.3.9. int SM_SetMenuProc(MenuType *m, int (*f)())

Sets the application function called when the menu item *m* is selected to *f.* The menu type for *m* must be ACTION_ITEM for the menu function to be called. **SM_SetMenuProc** will not fail if the type is not ACTION_ITEM but selecting the menu won't call the function; it'll cause the pull-down menu to be activated. You can't have it both ways.

5.4. Example

Unlike gadgets, menus are created independently of any window. Menus can be attached to windows when they're created, or can be added later. The same menu can be attached to any number of windows. Since only one window can be active at any time, there is no ambiguity when windows share menus. Unlike gadgets, which the window manager automatically destroys when their parent window is closed, menus are never destroyed by TWS itself.
If a menu is attached to more than one window, those windows *share* the menu. Any change to the menu in one window will affect all other windows that use the same menu.
The first step is to create the top level menus using the **SM_CreateMenu** function. Then add items to the menus using the **SM_AddMenuItem** function:

```
MenuType       *windowmenu;
MenuType       *imagemenu, *colormenu;

windowmenu = SM_CreateMenu();
imagemenu = SM_CreateMenu();
colormenu = SM_CreateMenu();
```

windowmenu will be the menu attached to the window, while the others will be items drawn in the menu bar. The next statements establish this:

```
SM_SetWindowMenu(w, windowmenu);
SM_AddMenuItem(windowmenu, SUBMENU_ITEM, “Image”, NULL, imagemenu);
SM_AddMenuItem(windowmenu, SUBMENU_ITEM, “Colors”, NULL, colormenu);
SM_AddMenuItem(colormenu, ACTION_ITEM, “Linear Color”, SetLinearColor, NULL);
SM_AddMenuItem(colormenu, ACTION_ITEM, “Log Color”, SetLogColor, NULL);
SM_AddMenuItem(colormenu, ACTION_ITEM, “Linear BW”, SetLinearBW, NULL);
SM_AddMenuItem(colormenu, ACTION_ITEM, “Log BW”, SetLogBW, NULL);
SM_AddMenuItem(colormenu, ACTION_ITEM, “Truecolor”, SetTruecolor, NULL);
SM_AddMenuItem(colormenu, ACTION_ITEM, “Invert”, InvertColors, NULL);
SM_AddMenuItem(colormenu, ACTION_ITEM, “Reset”, RestoreColors, NULL);
SM_AddMenuItem(colormenu, ACTION_ITEM, “Brightness”, SetBrightness, NULL);
```

If an item will be a submenu of a menu, the submenu must be created first using **SM_CreateMenu**. Whether a menu will be a window menu or a pull-down menu depends on how the items are attached. In the above example, *windowmenu* has been attached to a window with *Image* and *Colors* as its contents. Therefore *windowmenu* is the bar menu of the window, and contains two submenu items. The *colormenu* menu contains eight ACTION_ITEM items, so when *colormenu* is selected those items will be displayed in a pull-down submenu. The above code reproduces the menu seen at the beginning of this chapter.

The ACTION_ITEM callback function is supplied by the application. It’s an int function with a single argument, a pointer to a WindowType. When called, the argument will point to the window the menu is attached to.

Note that since none of the labels in the example menus are flagged for Alt-key selection, the user can operate the menus only with the mouse.

6. Fonts

Through release 4.0, TWS had only rudimentary font handling abilities. The system maintained four global fonts: the *system font* for gadgets and menus; *title font* for window and group boundary titles; *text font* for the text display gadget; and *icon font* for icons.
Beginning with release 4.1, TWS adds an additional dimension to font control. Every gadget instance can now be assigned a unique font. A font is specified as a text string, which is the name of the disk file the font is stored in. The font name includes the file extension. For example,

```
SM_SetSystemFont("hel12.fnt");
```

sets the system font to the one in the file HEL12.FNT. Case doesn't matter.
To manage the potentially complex font interactions and to minimize memory requirements for multiple fonts, a *dynamic font loading* mechanism has been implemented for swapping font data in from disk as necessary (See **TWS Font Table**).
A TWS application can use any MetaWINDOW font. All font files are assumed to be in the directory specified by the *FontPath* configuration variable, or in the current default directory if *FontPath* isn't defined.

6.1. How TWS Uses Fonts

A font is an attribute of a gadget or window. When a gadget is about to be drawn, TWS takes care of whatever is necessary to get the gadget's font loaded into the system. All the application has to do is define which font to draw the gadget with using the **SM_SetGadgetFont** function. If not otherwise specified, gadgets use the system font.
An application can change the system resource fonts (system font, title font and icon font) whenever it wants to. Because of the event-driven nature of a TWS application, any system font changes would probably only be done when individual windows are created or become the focus window.

6.2. TWS Font Table

Fonts for gadgets are stored internally in a *font table*. There is room for up to five application fonts in the table, plus a system, title and icon font, all of which are stored in RAM.[4]
When a gadget requests to use a font, the font manager searches the table for the requested font. If the font is already in the table TWS simply makes it the active font and returns.
If the font isn't already in the table and there's an unused slot in the table, TWS loads the font from the directory specified by the *FontPath* configuration variable, or from the current directory if the *FontPath* variable isn't defined. The new font is then set as the active font.
If both of these attempts fail, TWS selects one of the slots in the font table and loads the requested font in that slot, discarding the one that was already there. The algorithm for selecting the font to be discarded is the same as the LRU (Least Recently Used) algorithm for discarding memory pages in a virtual memory system like Unix. The font manager maintains a "clock" or counter for each font. When a font is made active, its counter is reset to 0 and the counters of all other fonts are incremented. The larger the counter, the longer its been since the font was used. The font manager selects the font with the largest counter value as the one to discard.
The font table mechanism doesn't apply to the system or title fonts, which are used by menus, window titles and system dialogs. While these can be changed by the application they are not subject to swapping.
In addition to the font table and the system resource fonts, TWS maintains the concept of the *current font*. At any point in time, the current font is the text font that would be used if the

[4] The text font has been discontinued -- instead, the Text gadget can be assigned its own font just like any other gadget.

system were to draw text. Access to the current font is particularly efficient since it avoids searching the font table. Whenever a request is made to set a font, TWS first checks to see if the request is already the current font. If so the font manager can return immediately.

6.3. Managing Fonts

As a consequence of the use of dynamic swap tables for storing a limited number of fonts, any attempt to access a font could result in loading the font file from disk. This can be time-consuming if an application uses a great many different fonts. Here are some guidelines for maximizing performance.
Use only as many fonts as absolutely necessary. An application that limits itself to no more than five fonts, plus the system, title, and icon fonts, will never have to swap to disk.
The best possible case is to use the system font for everything.
For applications that use multiple fonts, whether swapping or not, performance is improved by grouping gadgets by common font. Remember that as each gadget is drawn the system sets the current font to the gadget's font, loading that font from disk if necessary. Once a font is set, the system does almost no work at all to set the same font again. Since gadgets are always drawn in the order they're created in the window, an effective strategy is to create all window gadgets that use the same font, then all gadgets that use the next font, etc.
Slight performance improvement comes from setting the most used fonts first, since the system scans the table for matching fonts. This can be difficult to ensure, however, and is probably impossible if fonts must be swapped in.
Avoid stroked fonts. They take no more time to load than bitmap fonts but take much longer to draw.

6.4. Interface functions *[smfont.h]*

6.4.1. char *SM_GetCurrentFont()

Returns the character string name of the current font. The string is the name of the file where the font is stored. *See also: SM_GetIconFont; SM_GetSystemFont; SM_GetTitleFont*.

6.4.2. int SM_GetCurrentFontDescent()

Returns the size in pixels of the descender portion of the current active font. *See also: SM_GetSystemFontDescent*.

6.4.3. int SM_GetCurrentFontHeight()

Returns the height in pixels of the current active font. *See also: SM_GetFontHeight, SM_GetCurrentFont***.**

6.4.4. void *SM_GetFontBuf(char *font)

Returns a pointer to the font's buffer area, loading the font from disk if necessary. The font buffer contains the graphics kernel representation of the font. The application can cast the buffer to the kernel representation, then extract information from it:

```
fontRcd *fbuf;
int           angle;
fbuf = (fontRcd *)SM_GetFontBuf("cour9.fnt");
if (fbuf) {
       angle = fbuf->chAngle;
}
```

The example shows how to obtain the italics slant angle from a MetaWINDOW™–based application. The **SM_GetFontBuf** function returns a pointer to the MetaWINDOW™ font record, loading it from disk (and thereby using a slot in the font table, potentially swapping out an existing font) if necessary.

6.4.5. int SM_GetFontHeight(char *font)

Returns the height in pixels of the requested font. If necessary, the font is temporarily read in from disk so its height can be determined. *See also: SM_GetSystemFontHeight, SM_GetTitleFontHeight, SM_GetIconFontHeight*.

6.4.6. char *SM_GetGadgetFont(void *gadget)

Returns the character string name of the font associated with the gadget *gadget*. All gadgets have a font, whether or not they output any text. The default font is the system font at the time the gadget was created.

6.4.7. char *SM_GetIconFont()

Returns the character string name of the icon font. The string is the name of the font file where the font is stored.

6.4.8. char *SM_GetSystemFont()

Returns the character string name of the system font. The string is the name of the font file where the system font is stored.

6.4.9. int SM_GetSystemFontDescent()

Returns the size in pixels of the descender portion of the system font. *See also: SM_GetCurrentFontDescent; SM_GetSystemFontHeight*.

6.4.10. int SM_GetSystemFontHeight()

Returns the height in pixels of the system font. The height is the full vertical size of the font's bounding box. *See also: SM_GetCurrentFontHeight; SM_GetSystemFontDescent; SM_GetTitleFontHeight*

6.4.11. char *SM_GetTitleFont()

Returns the character string name of the title font. The string is the name of the font file where the font is stored.

6.4.12. int SM_GetTitleFontHeight()

Returns the height in pixels of the current title font. The height is the full bounding box height of the character.

6.4.13. void SM_SetGadgetFont(void *gadget, char *fontname)

Sets the font that will be used when drawing the gadget *gadget* to that in the disk font file *fontname*. *Gadget* is the superclass gadget of a specific gadget (retrieved using the **GetGadgetSuperclass** function). The gadget must already be created. The font file must be in the directory specified by the *FontPath* configuration variable, or in the current directory if *FontPath* isn't defined.
If the gadget is in the focus window, it is redrawn to reflect the new font. Changing a gadget's font doesn't change the gadget's font size.

6.4.14. void SM_SetGadgetFontSize(void *gadget, int width, int height)

Sets the width and height for a gadget that uses stroked kernel fonts. If the gadget uses a bitmap font the function succeeds but has no affect -- the size of a bitmap font is fixed.

If the gadget is part of the focus window, the gadget is redrawn immediately with the new font size.

6.4.15. int SM_SetIconFont(char *fontname)

Sets the system font to that in the disk font file *fontname*. The new font will be used for all subsequent icon drawing, but doesn't affect any icons already drawn.

6.4.16. int SM_SetSystemFont(char *fontname)

Sets the system font to that in the disk font file *fontname*. The new font will be used for all subsequent system text, such as menus and certain dialogs, but doesn't affect any system text already drawn.

6.4.17. int SM_SetTitleFont(char *fontname)

Sets the system font to that in the disk font file *fontname*. The new font will be used for all subsequent titles, but doesn't affect any titles already displayed.

6.4.18. int SM_SetFont(char *font)

Sets the font *font* as the current font, loading it from disk if necessary. An application only has to do this explicitly if it draws text in the graphics canvas rather than in label or other text gadgets.

7. Cursors

Some individual regions in the TWS user interface have unique cursors associated with them. The *cursor* is the small pictograph on the screen that indicates where the graphics locator is, and shows where mouse events will be directed. TWS applications can change the cursor shape directly, and can also set the shape the cursor will be when it is in certain window regions.
The window regions where cursors are attached and their default cursor shapes are:

Region	Cursor Mnemonic	Default shape
Window title bar	MOVECURSOR	4-way arrows
Upper-left resize handle	ULRESIZECURSOR	2-way arrow
Upper-right resize handle	URRESIZECURSOR	2-way arrow
Lower-left resize handle	LLRESIZECURSOR	2-way arrow
Lower-right resize handle	LRRESIZECURSOR	2-way arrow
Graphics state canvas	GRAPHCURSOR	Crosshair
All others	MAINCURSOR	Standard upper-left pointer

TWS automatically switches the cursor to the appropriate shape when the cursor enters the appropriate region of the active window. Regions of inactive windows don't trigger a change. If the cursor isn't in any particular region then the MAINCURSOR is displayed.

7.1. Data Types

```
typedef struct {
        int             hotx, hoty;
        unsigned char   *foremask, *backmask;
} CursorType;
```

The CursorType structure contains the definition for a TWS cursor. Cursors are composed of two 16x16 bit arrays, contained in the *foremask* and *backmask* fields. When a cursor is displayed, the "1" bits of the background mask are drawn in black, and the "1" bits of the foreground mask are drawn in white. The "0" bits of both masks are transparent.
The *hotx* and *hoty* fields define the bit in the mask that is the cursor's screen location, called the cursor's hotspot.

7.2. Setting Application Cursors

The application can change the cursor. One cursor is set aside for application use, called the USERCURSOR. This cursor starts out undefined.[5] The first step is to define a shape and hotspot for this cursor using the **SM_DefineCursor** function:

```
CursorType     my_cursor;
/* code to setup the cursor omitted */
SM_DefineCursor(USERCURSOR, &my_cursor);
```

[5] Actually, the shape of this cursor may be predefined by the graphics kernel system. If you select this cursor before defining it, you'll get the graphics kernel cursor.

The CursorType structure contains a mask for the cursor foreground and background, and the cursor-relative coordinates of the hotspot — the single pixel within the cursor rectangle which the system uses to determine the cursor's coordinate.
Then to switch to the custom cursor, call **SM_SetCursor**:

```
SM_SetCursor(USERCURSOR);
```

to restore the normal arrow cursor, call **SM_SetCursor** again:

```
SM_SetCursor(MAINCURSOR);
```

7.3. Changing the System Cursors

Using the same technique as above, an application can change the cursor designs used for moving windows, resizing, the graphics canvas, etc. The cursor manager automatically changes to these cursors when the mouse pointer enters the appropriate region.
Instead of USERCURSOR, substitute one of the following constants to redefine a system cursor:

Constant	*Description*
MAINCURSOR	The normal system cursor, an upward-left pointing arrow;
MOVECURSOR	The cursor displayed when the mouse pointer is in a window title bar;
GRAPHCURSOR	The cursor displayed when the mouse pointer is in the active window's graphics canvas region;
URRESIZECURSOR	
ULRESIZECURSOR	
LRRESIZECURSOR	
LLRESIZECURSOR	Cursors displayed for the upper-right, upper-left, lower-right, and lower-left window resize drag regions, respectively

7.4. Interface Functions *[cursor.h]*

7.4.1. int SM_GetCurrentCursor(void)

Returns the cursor constant (see the table above) for the cursor currently displayed.

7.4.2. int SM_SetCursor(int cursornum)

Sets the display cursor to the one currently installed at *cursornum*. The value of *cursornum* can be any integer expression that evaluates to (0 .. MAXCURSORS-1). Returns 0 if *cursornum* is in range, non-0 otherwise.

7.4.3. int SM_DefineCursor(int cursornum, CursorType *cursor)

Replaces the cursor *cursornum* with the user-defined cursor *cursor.* Once a system cursor has been reset it can be restored to its previous shape only by setting it with another call to **SM_DefineCursor**.

Example

The following shows how an application would set up a custom cursor and display it.

```
    /*
    0000000000000000
```

```
0000000000000000
0000000000000000
0000000000000000
0000000---000000
0000000-1-000000
00000---1---0000
00000-11011-0000
00000---1---0000
0000000-1-000000
0000000---000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
*/
CursorType mycursor = { 7, 7,
{0x00, 0x00,
0x00, 0x00,
0x00, 0x00,
0x00, 0x00,
0x00, 0x00,
0x00, 0x80,
0x00, 0x80,
0x03, 0x60,
0x00, 0x80,
0x00, 0x80,
0x00, 0x00,
0x00, 0x00,
0x00, 0x00,
0x00, 0x00,
0x00, 0x00,
0x00, 0x00},
{0x00, 0x00,
0x00, 0x00,
0x00, 0x00,
0x00, 0x00,
0x01, 0xc0,
0x01, 0x40,
0x07, 0x70,
0x04, 0x10,
0x07, 0x70,
0x01, 0x40,
0x01, 0xc0,
0x00, 0x00,
0x00, 0x00,
0x00, 0x00,
0x00, 0x00,
0x00, 0x00};

int main(int argc, char *argv[])
{

    /*
    ** Define the user cursor to the cursor manager
    */
    SM_DefineCursor(USERCURSOR, &mycursor);
    /*
    ** Make the user cursor the one that is displayed automatically
    ** whenever the cursor is in the active window graph region
    */
    SM_DefineCursor(GRAPHCURSOR, &mycursor);
    /* ... */
}


int MyWindowProc(WindowType *w, EventType *event)
{
    int    oldcursor;            /* Original cursor to save    */

    if ((event->ButtonEvent == LEFTBUTTONPRESS)
        && (event->Region == CONTENTREGION)) {
                /*
                ** If the user presses the left mouse button inside the window,
                ** change the cursor
                */
                oldcursor = SM_GetCurrentCursor();
                SM_SetCursor(USERCURSOR);
                /* ... additional processing ... */
                SM_SetCursor(oldcursor);
        }
    }
```

8. Messaging

The TWS system is designed such that different parts of an application share data through the window structure's user data field. For passing data between different parts of a program which don't share a window structure, a simple messaging system is provided. The messaging system provides a central structure for broadcasting and retrieving messages.

In order to use the messaging system, message senders and receivers must agree on a protocol ("I'll send messages *to* XXX; you look for messages *addressed to* XXX"), and a message dictionary ("If I send a *message* of 1 it means I'm hungry; if I send a *message* of 2 it means bring me a beer."). There can be any number of functions sending or receiving messages, or both.

Retrieved messages are removed from the message queue. The message queue is a fixed length, 128 messages. When the queue is full additional messages are discarded. The posting function is notified (the **SM_PostMessage** function fails) if the message can't be posted. However, there's no way for a receiving function to know that somebody tried to post a message but the posting failed. All a receiving function can know is whether or not there are messages pending. It is the application writer's responsibility to ensure that messages are retrieved in a timely manner so that the message queue doesn't overflow.

Messaging is inherently *asynchronous*; that is, there's no way for a function that posts a message to know exactly when the message will be retrieved, or that it will be retrieved in any particular order relative to other messages in the queue. In fact, there's no way to force messaging to be synchronous. It's up to the application programmer to take advantage of this behavior.

The window system itself does *not* use the messaging system. Messaging is provided as a convenience for the application writer. Messaging can be useful for communicating to/from background procedures.

In general you should use the window *data* fields to provide shared data whenever possible, because it's a more efficient mechanism than messaging. Messaging tends to be more flexible, however.

8.1. Data Types

A message is contained in a data structure described below:

```
typedef int     MessageType;
typedef int     SenderType;
typedef void    *UserDataType;

typedef struct {
    MessageType     message;
    SenderType      from;
    SenderType      to;
    UserDataType    data;
} MessagePacketType;
```

The *message*, *from*, and *to* fields are used to distinguish who messages are intended for. A receiving function can grab all messages of a particular type (ie., with a specific value in the *message* field), all messages from a certain sender (ie., a specific value in the *from* field), and/or all messages addressed to a specific destination (ie., with a specific value in the *to* field). Hybrid combinations are also possible.

The *data* field is a pointer to a memory region. Typically the sending function allocates and fills this memory with the information to be sent. The receiver would then free the memory after receiving the message. The danger here is that if messages aren't retrieved, memory is wasted in undelivered message data. This is another reason to make sure messages are retrieved with dispatch.

8.2. Interface Functions *smmessag.h*

8.2.1. int SM_GetMessage(MessageType t, MessagePacketType *msg)

Returns the next message on the message queue of type *t* in the message packet variable *msg*, which must be allocated. If there are no messages on the queue of type *t* then False is returned and *msg* is unchanged.

8.2.2. int SM_GetMessageFor(SenderType t, MessagePacketType *msg)

Returns the next message on the message queue addressed to *t* in the message packet variable *msg*, which must be allocated. If there are no pending messages for *t* then False is returned and *msg* is unchanged.

8.2.3. UserDataType SM_GetMessageData(MessagePacketType *msg)

Returns the data associated with the message, if any. The application can then cast as appropriate.

8.2.4. SenderType SM_GetMessageDestination(MessagePacketType *msg)

Returns the identifier for the addressee for the message.

8.2.5. SenderType SM_GetMessageSender(MessagePacketType *msg)

Returns the identifier for the sender of the argument message.

8.2.6. MessageType SM_GetMessageType(MessagePacketType *msg)

Returns the message type of the argument message.

8.2.7. int SM_InitMessage(void)

Initializes the messaging system. Must be called before any messaging functions can succeed. Returns True if the message system is successfully initialized.

8.2.8. int SM_PostMessage(MessageType t, SenderType from, SenderType to, UserDataType data)

Puts a message onto the message queue with the supplied data. If the message queue is full or uninitialized then False is returned, otherwise True.

8.3. Example

There are three basic steps to using the TWS messaging system: Initialize the messaging system by calling **SM_InitMessage** (probably in the main function); create a message ‘dictionary’ that defines message types and establishes sender and receiver IDs; write the appropriate functions that post and retrieve messages.

```
#define        MSGCOUNTDOWN   1      /* Message type                      */
#define MSGCOUNTFUNC   1      /* Sender ID                      */
#define MSGRECVFUNC    2      /* Receiver ID                    */
```

Notice that the message type and sender ID are the same actual value. This is harmless.
The example sender simply posts a message after every 10th time it’s called. The message is the date and time the message was posted.

```
#include   <smmessag.h>
#include   <time.h>
void Countdown(void)
{
       static unsigned long  counts = 0L;
```

```
        static time_t          now;

        counts++;
        if ((counts % 10) == 0) {
            now = time(NULL);
            SM_PostMessage(MSGCOUNTDOWN, MSGCOUNTFUNC, MSGRECVFUNC, (UserDataType)now);
        }
}
```

Notice that, in this example, the sender doesn't allocate memory for the message data. Instead a local static variable is used. This is another convention that has to be understood between sender and receiver. Generally an application would not use this technique because all messages would reference the same memory location, so the message data for all messages is changed every time a new message is posted.
The receiving function simply retrieves messages and recovers the data:

```
#include    <smmessag.h>
#include    <time.h>
void Receiver(void)
{
        time_t         then;
        MessageType    msg;

        if (SM_GetMessage(MSGCOUNTDOWN, &msg)) {
            then = (time_t)SM_GetMessageData(&msg);
            fprintf(stderr, “Counter message occurred at %s\n”,ctime(&then));
        }
}
```

9. Icons

An icon is a minimized representation of a window. When an application user selects the window minimize button, the window's icon representation is displayed. While the window is minimized any background processes attached to the window continue normally, but the user can only move the icon on the screen and restore it to its original size.
TWS icons are not stored on disk. Icons are created dynamically by the window system, allocated and de-allocated via TWS functions. The pixmap images displayed within icons (and used elsewhere within TWS) can be stored on disk, however.
Also see the chapter on PixmapTypes in the Gadgets section.

9.1. Creating a Window Icon

By default, a window has no icon and therefore can't be minimized. Before any window can be minimized the application must set the window icon with the **SM_CreateIcon** function. This function creates the icon and sets the title and pixmap that will be used in the icon display.
The size of an icon is determined by the pixmap that's passed to it. TWS adds a little space around the edge of the pixmap, plus enough space below the pixmap to write the icon title. If no pixmap is passed, TWS uses a default size of 72x72 pixels. Usually an application will use a set of same-size icon pixmaps to give the application a consistent look. Also see the Pixmaps gadget chapter for more information on TWS pixmaps.
The position of an icon is determined when it's first used. Icons are initially positioned along the right edge of the workspace, starting at the top. Subsequent icons are positioned down the edge of the screen. The application user can drag icons (or the program can move them in code) to any position on the screen. Each time a window is minimized the icon is redrawn in its previous position.
Minimized windows remain on the window stack, but, unless there are no non-minimized windows on the screen, an icon window can't be the active window. When an icon is restored, it becomes the active window.

9.2. Icons and Closing Windows

When a window that has an icon attached to it is closed, the icon is freed as well. However, the pixmap attached to the icon is *not* freed. This is because any number of window icons may share a single pixmap. The application must free the pixmaps attached to icons as necessary.

9.3. Operations on Icons

The application user can only move and restore icons. An icon is moved by dragging it with the mouse. Click and drag anywhere within the icon border to move the icon. To restore the window from an icon, click within the icon without moving the mouse. Restoring a window from an icon generates a window resize event. The application can also move an icon in code and can change the icon pixmap.

9.4. Data Types

```
typedef struct {
    char        *title;               /* String label for the icon        */
    WindowType  *parentwin;           /* Window the icon belongs to       */
    PixmapType  *pixmap;              /* Icon image                       */
    LabelType   *label;                /* Icon label                      */
    int         x, y;                  /* Icon absolute position          */
    char        *screenunder;          /* Screen under the icon           */
    int         (*dropfunc)();         /* Drop function for drag-n-drop   */
    void        *data;                 /* User data                       */
} IconType;
```

The *title* is the text displayed in the icon's label region. If this string is NULL then the title field from the window is used. In either case, the string is clipped at the label region boundaries. *parentwin* is the window the icon belongs to. Every window has its own icon. The *pixmap* displayed in the icon is also unique to the icon, even if the image displayed is the same as other icons.
The icon position is stored in *x,y*, which is never manipulated by the application. The *screenunder, dropfunc* and *data* fields are currently unused.

9.5. Interface Functions *smicon.h*

9.5.1. IconType *SM_CreateIcon(WindowType *w, char *title, PixmapType *pm)

Creates a new icon and attaches it to the window. The icon title will be set to *title*, and the pixmap image for the icon will be *pm*. If *title* is NULL then the window title will be used. If *pm* is NULL then a blank image is displayed when . Returns a pointer to the new icon.
Example:

```
w = SM_NewWindow(&r, “MyWindow”, DOCUMENT, VOID, VOID);
pm = SM_ReadPixmap(w, “mypixmap.pxm”);
wicon = SM_CreateIcon(w, NULL, pm);
```

Creates an icon whose title is be the window title “MyWindow” and whose image will be the pixmap contained in the file `mypixmap.pxm`.

9.5.2. PixmapType *SM_GetIconPixmap(WindowType *w)

If the window has an icon, returns the pixmap for the icon. Returns NULL if the window has no icon or if the icon has no pixmap.

9.5.3. void SM_MoveIcon(WindowType *w, int dx, int dy)

Move the icon associated with window *w* a delta amount, (*dx, dy*). Positive values for *dx* and *dy* move the icon to the right and down, respectively, while negative values move it to the left and up.

9.5.4. void SM_SetIconPixmap(WindowType *w, PixmapType *pm)

Set the pixmap for the icon for window *w* to *pm*. The pixmap should already be created, and the window should already have an icon. If the window doesn't have an icon then the function does nothing. If the icon already has a pixmap it is simply discarded, so the application should take care to retrieve and free an existing pixmap as necessary.

Part Two

The Graphics State

10. Graphics Functions

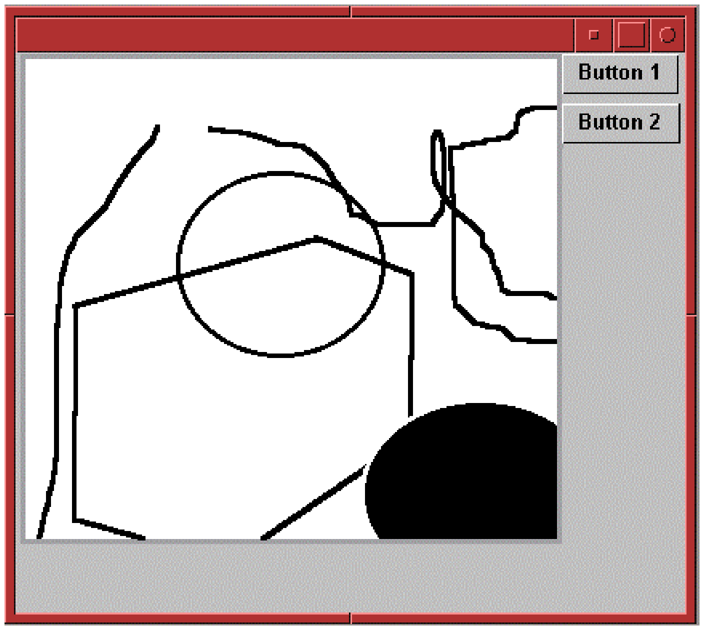
TWS supports a limited set of drawing and graphics functions for drawing points, lines, and other simple 2d geometry, and for setting drawing colors, changing the drawing canvas and other housekeeping chores. The window graphics system combines with the Color Manager system to provide a flexible tool set for drawing and imaging applications.

Of course, the entire TWS window system is graphical in nature, in the sense that the computer video system is run in graphics mode and the entire user interface is produced using graphics kernel routines. The topic of this section is the specific set of TWS functions that permits an application to do “traditional” direct drawing.

10.1. The Graphics State

Before an application can draw graphics in a window a Graphics State must be created for the window. The *graphics state* is a set of parameters that defines locations, colors, drawing modes, line styles and so forth for drawing in a window. The graphics state stores all this information and applies it to all drawing requests to the window. This allows the application to specify green as the drawing color, for example, and know that anything drawn in that window will be green, no matter what other windows are opened, get the focus or whatever.

All drawing is relative to the graphics state’s canvas region and is clipped within it. The canvas, in turn, is relative to the window’s content region. By default the canvas is the same size as the content region, but can be made either larger or smaller.



The illustration shows how the graphics are clipped to the window’s graphics canvas (white region), while the window’s content region (gray region) is larger.

This behavior makes it easy to separate window control graphics like buttons and labels from the drawing graphics. A drawing or painting program doesn’t have to worry about clobbering buttons or other window controls as long as they are outside the graphics canvas[6].

Besides clipping at the canvas borders, the Graphics State keeps track of other information, like the current graphics locator position, drawing color and rasterop mode, background color and so

[6]Actually there’s no mechanism to enforce this, and the system will happily draw buttons or whatever right over your graphics canvas. It’s then up to you to worry about preserving the buttons.

forth. The bottom line is that when a window gets the focus, its Graphics State is just as it was when it lost the focus.
The main purpose of the draw and redraw window management event functions is to provide a way to restore graphics whenever a window is moved, resized, exposed, etc. The window system itself knows how to restore gadgets and menus and so forth, so the application mainly has to worry about drawing anything in the graphics canvas.

10.2. Data Types

```
typedef struct {
    int         gx, gy;             /* Graphics cursor location         */
    int         tx, ty;             /* Text cursor location             */
    ColorType   *palette;           /* Color palette                    */
    int         ncolors;            /* Number of colors in the palette  */
    ColorType   *forecolor;          /* Graphics foreground color        */
    ColorType   *backcolor;          /* Graphics background color        */
    ColorType   *tcolor;             /* Text color                       */
    int         penmode;            /* Xor, And, etc.                   */
    int         linewidth;
    int         linestyle;          /* Dash, solid, etc.                */
    int         textmode;
    FontType    *font;              /* Text font                        */
    RectType    devicecanvas;       /* Drawable region in device coords */
    RectType    canvas;             /* Drawable region in user coords   */
    char        *save_file;         /* Save-under storage file          */
    int         step;               /* Number of save-under buffers     */
    RectType    buffrect[64];
    int         canvasborder;       /* Flag for drawable region border  */
    } GraphStateType;
```

By default a window has no graphics state. The application must create and initialize a graphics state for those windows that need it using the **SM_CreateGraphState** function.
The *palette* variable is where a window's local color table is stored. The foreground, background and text color fields are the colors used for drawing, erasing, and drawing text, respectively. *gx* and *gy* mark the position of the graphics cursor, which is the origin position for relative drawing (**GR_LineTo** for example), while *tx* and *ty* serve the same function for graphics text.
The *canvas* is a rectangle in either an absolute or virtual coordinates. Absolute coordinates are relative to the upper-left corner of the window's content region. They're specified using non-negative integers for the rectangle coordinates and are in units of pixels.
Virtual coordinates are also relative to the upper-left corner of the window content region but are specified using negative integers between -999...-1. TWS treats the absolute values of these coordinates as the fractional distance from the upper left corner to the opposite side, in thousandths of the distance. For example, a position of -500 is abs(-500)/1000 = 500/1000 = 0.5 = 1/2 the distance from one edge of the content region to the opposite edge. Using virtual coordinates, the relative position and size of the canvas stays the same even though the window is resized.
Whether virtual or absolute coordinates are used for the canvas, the *devicecanvas* always contains the absolute size and position of the canvas in device coordinates. This rectangle is maintained internally so it doesn't have to be recalculated for every drawing request.
The *canvasborder* field determines if the canvas will have a border and, if so, what style of border. The relevant constants are in smtypes.h:

Border Constant	Value	Description
SCULPTED	1	A 3D "recessed" border -- the interior of the canvas appears sunken into the screen. The depth depends on the BevelDepth window configuration option.
FLAT	3	A single black line.

BEVELED	4	A 3D "raised" border -- the interior of the canvas appears to rise from the screen. The depth depends on the BevelDepth window configuration option.
CHISELED	5	The appearance of a line "etched" into the screen.

A value of 0 (SMDEFAULT) leaves the canvas unbordered. For the SCULPTED and BEVELED styles, the depth of the effect depends on the BevelDepth configuration option.

10.3. Using Graphics

In general, an application can treat a graphics canvas as though it were the entire screen. The upper left corner is (0,0). A reference outside the canvas is simply clipped. It doesn't really matter where the window is on the screen and it doesn't matter what happens between successive draws; the pen location, colors, and all other parameters within a canvas are preserved.

10.3.1. Graphics and Color

The TWS graphics state and the color manager work together to enable graphics for an application. Colors can be taken from the window color table, or from the system color table. Before the graphics functions can use a window color, of course, the application must allocate and fill one.

Using system colors allows an application to draw using the fixed set of colors the color manager reserves for window borders, backgrounds, titles and so forth. This assures that a graphic will look the same in any graphics mode, and regardless of the window colors. When an application needs more control over colors and more of them the window color table would be used.

The graphics system doesn't know whether a color is a system color or a window color. It's up to the application to be sure to always provide legal colors to the graphics state. *See the chapter on Colors for more information.*

10.3.2. Images

TWS has provisions for working with areas of the graphics display in a fast, hardware-dependent mode. Rectangular areas of a graphics canvas can be captured into a ImageType variable, and can later be written back to the same graphics canvas, or a canvas in a different window.

To capture a portion of the canvas, first determine the amount of memory required using the function **GR_ImageSize**. If the size value returned is greater than 0 then allocate a pointer to an ImageType variable. The image is then captured using the **GR_GetImage** function.

```
long    size;
ImageType      *image;
RectType       r;

/*
** Create the rectangle enclosing the area to be captured
*/
r.Xmin = 1;
r.Ymin = 1;
r.Xmax = GR_GetCanvasWidth(w) - 1;
r.Ymax = GR_GetCanvasDepth(w) - 1;
/*
** Find out how much memory required for an image this size
*/
size = GR_ImageSize(&r);
if ((size > 0) && (size < GRMAXIMAGE)) {
        image = (ImageType *)malloc((size_t) size);
        GR_GetImage(w, &r, image);
}
```

The constant GRMAXIMAGE is defined as the largest block of memory that can be allocated for an image. This value is defined in smgraph.h and is compiler and graphics kernel dependent.
The image can be put back into a graphics canvas, either the same one or a different one. The rectangle where the image is put must be *exactly the same size* as the rectangle when the image was grabbed.
If any part of the source rectangle for the **GR_GetImage** function is outside the canvas border, that part of the image will contain undefined values. When putting an image, any portions of the destination rectangle that are outside the canvas are clipped.

10.3.3. Blocking

When an application draws to a window, TWS goes to a lot of trouble to make sure the window has the focus, that the drawing will be clipped to the window canvas or border, whichever is smaller, that drawing will use the correct colors, line styles, etc., as stored in the graphics state. All of this takes time.
Tight loops that draw repeatedly to the same window can be optimized by setting *blocking* for the window. When blocking is enabled, TWS sets all the drawing parameters in the graphics kernel hardware and software from the window's graphics state. From then on, TWS skips the graphics state setup, assuming it's already done. This results in a significant performance boost.
Blocking imposes some restrictions on an application: for one, the application can't draw to any other window while blocking is enabled. This includes gadgets in the same window. Second, the application must only block the focus window. Third, the application must unblock the window when drawing is done. TWS doesn't explicitly forbid doing any of these things wrong, so it's up to the application developer to program carefully.
Any modifications to a gadget while blocking is enabled will not be reflected on the screen. Some gadgets, in particular Pixmaps, set a redraw flag internally if an application modifies them while blocking is on. When blocking is released, TWS explicitly refreshes any Pixmap gadgets that were modified. See the Iconedit demo program for an example of this.

10.4. Interface Functions *smgraph.h*

A TWS application must always use TWS graphics functions and *not* the graphics functions of the underlying graphics kernel system.[7] Only the TWS graphics routines are aware of the window hierarchy, the graphics state, clipping to the graphics canvas, etc.

10.4.1. int GR_CharWidth(char *c)

Returns the width in display pixels of the argument character, based on the currently loaded system font. Return value is unpredictable if *c* is not a printable character.

10.4.2. int GR_ClearCanvas(WindowType *w)

Erases the window's graphics canvas to the graphics background color. *See also: SM_EraseContent*

10.4.3. int GR_CloseGraphState(WindowType *w)

Deletes the graphics state associated with the window and reclaims memory used by it. This function is usually only called when a window is being closed and is called automatically by the TWS internally when a window is closed. An application *must not* attempt any graphics functions after the Graphics State has been closed. If the window has no graphics state then this function simply returns. *See also: SM_CloseWindow.*

[7]OK, this isn't strictly true either. That is, TWS doesn't *forbid* an application from using graphics kernel routines. I just think it's usually a bad idea. Please let me know if you need a window analog of a graphics kernel function; I'll get it into the TWS system.

10.4.4. GraphStateType *GR_CreateGraphState(WindowType *w, RectType *r, int border)

Creates a new window Graphics State and returns a pointer to it. The Graphics State is initialized with default values. *w* is the window the graphics state will belong to. *r* will be the graphics canvas and is specified in window content coordinates. If *border* is True then the graphics canvas will have a recessed border drawn around it, otherwise not.
If *r* is NULL then the canvas is initialized to the same size as the window content. Otherwise the canvas can be any size and in any position relative to the window content. Graphics output is clipped to the canvas and content, whichever is smaller.
The graphics state is initialized with the following values:

Field	Description	Default Value
gx, gy	Graphics locator/cursor position	0,0
tx, ty	Graphics text locator	0,0
palette	Graphics color table	NULL
ncolors	Number of colors in the color table	0
backcolor	Canvas background color	Window content color
forecolor	Drawing color	White
tcolor	Text color	White
penmode	Drawing mode (xor, etc.)	SMREP
textmode	Text drawing mode	SMREP
linewidth	Width of lines, rects, etc.	1 pixel
linestyle	Solid, dashed lines, etc.	GRNORMAL
font	Text font for graphics text	Current system font
save_file	File for backing store	NULL

10.4.5. int GR_DrawArray(WindowType *w, int x, int y, int l, ColorType *a)

Display an array of color values in the argument window's canvas region starting at position *x,y*.

w	Window to draw in;
x, y	Canvas-relative position where the color array should start;
l	Number of pixels to draw. The array must be at least this long;
a	Array of colors, must be at least *l* long.

The array *a* must contain at least *l* colors. The colors in the array do not have to be all the same. The array is drawn horizontally to the right starting at *x,y*, ending at position *x+l-1,y*. This function is much faster than placing the individual colors using the **GR_DrawPoint** function, especially when supported by the graphics kernel hardware. The graphics state locator position is not affected.

10.4.6. int GR_DrawCircle(WindowType *w, int x, int y, int r)

Draws a round circle of radius *r* centered at position *x,y* in window *w* using the window's graphics drawing mode, drawing color, etc.. The circle is 'hollow.' The graphics state locator is set to the point *x,y*.

10.4.7. int GR_DrawFilledCircle(WindowType *w, int x, int y, int r)

Like **GR_DrawCircle**, except that the interior of the circle is filled with the current drawing background color.

10.4.8. int GR_DrawFilledPolygon(WindowType *w, int npts, int *pts)

Like **GR_DrawPolygon** except the polygon interior is filled with the current drawing background color.

10.4.9. int GR_DrawFilledRect(WindowType *w, RectType *r)

Like **GR_DrawRect** except the rectangle interior is filled with the current drawing background color. The graphics state locator position is not affected.

10.4.10. int GR_DrawLine(WindowType *w, int x1, int y1, int x2, int y2)

Draw a line in the window *w* between points (*x1*,*y1*) and (*x2*,*y2*) using the current window Graphics State for line width, color, etc. The graphics state locator is set to the point *x2,y2*.

10.4.11. int GR_DrawPoint(WindowType *w, int x, int y)

Sets the point (*x,y*) in the window *w* graphics canvas to the current drawing color. The graphics state locator is set to the point *x,y*.

10.4.12. int GR_DrawPolygon(WindowType *w, int npts, int *pts)

Draws an outline of a polygon containing *npts* vertex points. The array *pts* contains at least *npts* pairs of x,y coordinates – that is, *pts[0]* and *pts[1]* are the first x,y coordinates, *pts[2]* and *pts[3]* are the next x,y coordinate, etc. Since by definition a polygon is closed, **GR_DrawPolygon** automatically connects the last point to the first. The Graphics State locator position is updated to the first/last point of the polygon.

10.4.13. int GR_DrawRect(WindowType *w, RectType *r)

Draws an outline of the rectangle *r*. The coordinates of the rectangle are relative to the window canvas. The Graphics State locator position is not changed.

10.4.14. int GR_DrawSolidCircle(WindowType *w, int x, int y, int r)

Like **GR_DrawCircle** except that the circle is filled with the graphics foreground color.

10.4.15. int GR_DrawSolidPolygon(WindowType *w, int npts, int *pts)

Like **GR_DrawPolygon** except that the polygon is the solid foreground color rather than outlined.

10.4.16. int GR_DrawSolidRect(WindowType *w, RectType *r)

Like **GR_DrawRect** except the rectangle is the solid foreground color rather than outlined.

10.4.17. int GR_DrawString(WindowType *w, int x, int y, char *string)

Draws the character string *string* in the specified window, with the left edge baseline of the string at (*x,y*) and using the window's current Graphics State. Like all graphics, the string is clipped at the window boundaries. The graphics text locator position is set to the end of the string.

10.4.18. ColorType *GR_GetBackgroundColor(WindowType *w)

Returns a pointer to the window canvas background color. The pointer is to the actual data in the window's Graphics State structure, not a copy, and should be treated as read-only.

10.4.19. int GR_GetCanvasDepth(WindowType *w)

Returns the depth of the canvas associated with the argument window. If the canvas is a virtual rectangle then the actual depth at the time of the function call is returned.

10.4.20. RectType *GR_GetCanvasRect(WindowType *w)

Returns a pointer to the rectangle structure for the canvas of the argument window. If the canvas is a virtual rectangle then the returned rectangle has the canvas dimensions at the time of the function call, in window content relative coordinates. The returned rectangle is a *static copy* of the canvas rectangle; it may be modified and must not be free'd.

10.4.21. int GR_GetCanvasWidth(WindowType *w)

Returns the width of the canvas rectangle for the argument window. If the canvas is a virtual rectangle then the actual width at the time of the function call is returned.

10.4.22. int GR_GetCursor(WindowType *w, int *x, int *y)

Returns the coordinates for the mouse cursor in graphics canvas-relative coordinates. If the cursor is outside the window *w*'s canvas then the values will be negative or greater than the canvas width or depth.

10.4.23. int GR_GetDevicePoint(int x, int y, ColorType *c)

Returns the color at the absolute screen pixel (*x,y*), where (0,0) is the upper-left corner of the display screen. If *x* or *y* are outside the screen the return value is unpredictable.

10.4.24. ColorType *GR_GetDrawColor(WindowType *w)

Returns a pointer to the graphics drawing color for the window *w*. The pointer is to the actual data in the window's Graphics State structure and should be treated as read-only.

10.4.25. void GR_GetGraphicsLocator(WindowType *w, int *x, int *y)

Returns the current graphics (*x,y*) location in window-relative coordinates.

10.4.26. int GR_GetImage(WindowType *w, RectType *rt, ImageType *twsimage)

Retrieves a rectangular block of pixels from the graphics canvas in window *w*. The rectangle *rt* encloses the region, which is read into the buffer field of *twsimage*.

10.4.27. int GR_GetMouse(WindowType *w, int *state, int *x, int *y)

Returns the position of the mouse cursor relative to the graphics canvas of the window *w*. *state* will contain the mouse button state at the time of the call. Constants defined in smevent.h are:

Constant	Value	Description
SMDEFAULT	0	No mouse buttons pressed.
RIGHTBUTTONACTIVE	2	Right mouse button is pressed
MIDDLEBUTTONACTIVE	4	Middle mouse button is pressed. Can't be returned by a two-button mouse.
LEFTBUTTONACTIVE	1	Left mouse button pressed.

If the window *w* has no graphics canvas then the function returns -1 and the values of *x, y,* and *state* are indeterminate. Otherwise the value of *state* is returned and *x,y* are the cursor coordinates relative to the canvas, where 0,0 is the upper left corner of the canvas region. That doesn't mean the cursor is in the canvas -- the values of *x,y* could be negative or greater than the canvas width or depth.

10.4.28. ColorType *GR_GetPalette(WindowType *w)

Returns a pointer to the Graphics State color palette, which is an array of ColorType structures. Returns NULL if the window has no color palette. Applications would normally treat this array as read-only.

10.4.29. int GR_GetPoint(WindowType *w, int *x, int *y, ColorType *color)

Returns the color at the point (*x,y*) in the graphics canvas region of the window *w*. Returns 0 on success or non-zero if the point is outside the canvas region. Does not affect the graphics locator position.

10.4.30. void GR_GetTextLocator(WindowType *w, int *x, int *y)

Returns the current graphics text *x,y* location in window-relative coordinates.

10.4.31. int GR_HideCursor()

Causes the mouse cursor to disappear. The mouse cursor position will continue to follow mouse movements, so that the cursor may be in a different place when it is shown again. If the cursor is already hidden then nothing happens.

10.4.32. unsigned long GR_ImageSize(RectType *rt)

Returns the number of bytes required to store an image, based on the current hardware graphics mode and graphics kernel system. For real-mode DOS versions of TWS, an image must be less than 64k bytes.

10.4.33. int GR_InsetRect(RectType *r, int dx, int dy)

Reduce or enlarge the argument rectangle by the amount *dx* in the horizontal dimension and *dy* in the vertical dimension. If *dx* or *dy* is positive then the corresponding sides of the rectangle move closer together; if negative then the sides move apart.

10.4.34. int GR_IsBlocked(void)

Returns True if blocking is on, False otherwise.

10.4.35. void GR_LimitMouse(RectType *r)

Restricts the graphics mouse to stay within the boundary described by rectangle *r*, whose values are in device coordinates. If the cursor is not within the rectangle then the results are unpredictable. If any of the values of *r* are outside the graphics hardware resolution then the results are unpredictable. *See also: GR_UnlimitMouse*.

10.4.36. int GR_LineTo(WindowType *w, int x, int y)

Draws a line from the current window graphics locator position to the window coordinates *x,y* using the current window Graphics State. The point *x,y* is relative to the window's graphics canvas. Either the locator position or *x,y* may be outside the window content region; the line is clipped at the boundary. The locator position is updated to *x,y*.

10.4.37. int GR_MouseInCanvas(WindowType *w)

Returns True if the current mouse position is inside the argument window's graphics canvas region, False otherwise.

10.4.38. int GR_MoveTo(WindowType *w, int x, int y)

Sets the window's graphics locator position to the point *x,y* in canvas-relative coordinates. The point may be outside the graphics canvas.

10.4.39. int GR_OffsetRect(RectType *r, int dx, int dy)

Translates the argument rectangle by *dx,dy*. Positive values move the rectangle to the right or down; negative values to the left or up.

10.4.40. int GR_PointInRect(int x, int y, RectType *r)

Returns True if the point (*x,y*) is on or inside the rectangle *r*, False otherwise. Both the point coordinates and the rectangle must be in the *same* coordinate reference frame -- e.g., relative to the same window graphics canvas. Notice that there is no window argument.

10.4.41. int GR_ProtectCanvas(WindowType *w)

Prevents the mouse cursor from displaying while inside the argument window's canvas region. When outside the region the cursor displays normally. If the window canvas is already protected then nothing happens. If another window's canvas region is protected then the new region replaces the existing one. If the window has no graphics state then nothing happens.

10.4.42. int GR_ProtectOff()

Cancels the effects of **GR_ProtectCanvas**. If no window canvas region is protected then nothing happens.

10.4.43. int GR_PutImage(WindowType *w, RectType *rt, ImageType *twsimage)

Draw an image stored in the *twsimage* structure into the graphics canvas of window *w*, bounded by rectangle *rt*. The rectangle must be exactly the right size to accommodate the image or it may not draw correctly.

10.4.44. int GR_SetBackgroundColor(WindowType *w, ColorType *color)

Sets the background graphics color for the window *w* to *color*. The color can be from the system color table or the window color table, but in either case (with one exception described below) must already be allocated and installed.
If the *index* field of the color is -1 (the default when a color is created using **SM_CreateColor** or **SM_InitColor**), **GR_SetBackgroundColor** will substitute the closest color in the system color table for the requested color. The color does not have to be part of the system or window color tables, and none of *color*'s fields modified. The -1 value means that the *color* has no mapping into the system color table.
Example:

```
#include       <smwindow.h>
#include       <smgraph.h>
#include       <smcolor.h>

WindowType     *w;
int    i;

SM_CreateWindowPalette(w, 256);
for (i = 0; i < 256; i++) {
       SM_SetWindowColor(w, i, i, i, i);     /* Create a gray-scale color table    */
}
GR_SetBackgroundColor(w, SM_GetWindowColor(w, 192));
```

See also: GR_SetDrawColor, SM_CreateColor, SM_InitColor, Color Manager.

10.4.45. int GR_SetBlocking(WindowType *w)

Turns on blocking for the window *w*. Loads graphics settings for the window as necessary and prevents any other window's settings from being loaded, resulting in faster drawing for the blocked window (but don't try to draw to other windows while blocked!). If another window is already blocked then returns a non-zero value, otherwise returns 0 on success.

10.4.46. void GR_SetCanvasBorder(WindowType *w, int mode)

Sets the canvas border style to *mode*. Constants for *mode* are defined in smtypes.h:

Constant	*Value*	*Definition*
SMDEFAULT	0	No border.
SCULPTED	1	A recessed beveling.
FLAT	3	A single black line.
BEVELED	4	A raised beveling.
CHISELED	5	A carved channel.

10.4.47. int GR_SetCanvasRect(WindowType *w, RectType *r)

Set the drawing canvas rectangle for the window *w* to the rectangle *r. r* can be either in window-content-relative coordinates or in virtual coordinates (or a combination). Regardless of the size of the canvas rectangle all drawing is clipped at the window content boundaries, or at the canvas boundary, whichever is smaller. Doesn't cause the canvas to be redrawn.

10.4.48. int GR_SetDrawColor(WindowType *w, ColorType *color)

Sets the drawing color for the window *w* to *color.* This sets the color that will be used for subsequent line drawing and other graphic operations but does not affect any colors already drawn. The *color* can be either a system color or window color (with one exception described below), but in either case must already be allocated and set.
If the *index* field of *color* is -1 (the default when a color is created), **GR_SetDrawColor** will substitute the closest color in the system color table for the requested color. The color does not have to be from one of the color tables, and none of *color*'s fields are modified. The -1 value means that the *color* has no mapping into the system color table.
Example:

```
#include        <smwindow.h>
#include        <smcolor.h>
#include        <smgraph.h>

ColorType       *color;

color = SM_CreateColor(37, 156, 29);  /* Allocate and initialize a color    */
GR_SetDrawColor(w, color);                    /* Set to the closest matching color  */
GR_DrawLine(w, 1, 1, 50, 1);          /* Draw a line using the color        */
```

See also: GR_SetBackgroundColor, SM_CreateColor, SM_InitColor, Color Manager chapter

10.4.49. int GR_SetDrawMode(WindowType *w, int mode)

Sets the drawing (rasterop) mode for the argument window to *mode*. The pen's mode is its combinatorial operation with the background, such as replace or XOR. Constants for the mode are found in the file smgraph.h. They are:

Constant	*Value*	*Description*
SMREP	0	Normal drawing; draw color overwrites anything
SMOR	1	Draw color is OR-ed with existing color
SMXOR	2	Draw color is XOR-ed with existing color (rubber-band mode)
SMNAND	3	Draw color is NOT AND-ed with existing color
SMNREP	4	Draw color is NOT REPLACE-ed with existing (transparent)
SMNOR	5	Draw color is NOT OR-ed with existing color
SMNXOR	6	Draw color is NOT XOR-ed with existing color
SMAND	7	Draw color is AND-ed with existing color

Note: Not all rasterop modes will be supported on all graphics kernel systems and all graphics hardware.

10.4.50. int GR_SetFont(FontType f)

Sets the font that will be used for *graphics* string drawing. Does not affect the font used for gadgets or other window elements.

10.4.51. void GR_SetLineStyle(WindowType *w, int style)

Sets the style of line that will be used for drawing lines and geometry. Values for *style* are defined in smgraph.h as GRNORMAL for solid line, GRDASHLINE for a dashed line, and GRDOTLINE for a dot line. *Note: not all graphics kernel system running the TWS system support all line styles for all types of geometry. For example, some may not support multi-pixel lines in other than solid, or may not directly support filled geometry with other than solid lines. TWS will attempt to emulate the expected graphics result in software when necessary.*

10.4.52. void GR_SetLineWidth(WindowType *w, int width)

Sets the width for drawing lines, graphics shape outlines like rectangles and circles, etc. *width* must be >= 0. *Note: Not all graphics kernel systems support multi-pixel wide lines, and some limit the widths (for example, to odd-numbers). TWS emulates multi-pixel lines in software but also may not support all possible widths. Two drawing width constants are provided that are guaranteed to be implemented in all TWS versions. These are GRWIDTHNORM (1 pixel lines), and GRWIDTHWIDE (3 pixel lines).*

10.4.53. void GR_SetPalette(WindowType *w, ColorType *p, int n)

Sets the color palette for the window to the color array *p* which must have at least *n* elements. If the window already has a color palette it is discarded. The color array *p* is copied to the window Graphics State. The new color palette is not immediately loaded into the hardware.

10.4.54. int GR_SetRect(RectType *r, int x1, int y1, int x2, int y2)

Puts the coordinate arguments into the rectangle structure, with *x1,y1* the upper left corner of the rectangle, and *x2,y2* the lower right corner.

10.4.55. int GR_SetTextMode(WindowType *w, int mode)

Sets the rasterop mode that will be used for drawing text to *mode*. Mode constants are the same as for the drawing mode and are defined in smgraph.h.

10.4.56. void GR_ShiftPolygon(int npts, int *pts, int dx, int dy)

Adjust all the points in the polygon by *dx* pixels in x and *dy* pixels in y. Does not draw the polygon.

10.4.57. void GR_ShiftRect(RectType *r, int dx, int dy)

Adjust all the points in the rectangle by *dx* pixels in x and *dy* pixels in y. Does not draw the rectangle.

10.4.58. int GR_ShowCursor()

Displays the cursor. If the cursor is already displayed then nothing happens.

10.4.59. int GR_StringWidth(char *s)

Returns the width of the string *s* in pixels. The value is based on the currently active font and will be different for different fonts.

10.4.60. int GR_TextWidth(char *s, int start, int length)

Returns the width in pixels of a substring taken from string *s* beginning at character *start* and the following *length* characters. If *length* is longer than the string then the width from *start* to the end of the string is returned. The width is based on the currently active font and may be different for different fonts. *See also: GR_StringWidth; SM_StringWidth*.

10.4.61. int GR_UnlimitMouse()

Allows the mouse cursor to roam freely around the screen. If the mouse is not constrained by a prior **GR_LimitMouse** call then nothing happens.

10.4.62. void GR_UnsetBlocking(void)

Turns of blocking. Since blocking can only be in effect for a single window, the window argument isn't necessary. Unsetting blocking causes all gadgets attached to the window to be redrawn if they have been modified. If blocking is off when **GR_UnsetBlocking** is called then nothing happens.

11. Colors

TWS implements a flexible color handling system that supports 4, 8, 15, 16, and 24 bit color systems. Particular attention has been paid to providing good color management for 4 and 8-bit lookup table systems like VGA.

There are two levels of color in the TWS color system: *system* colors and *window* colors. System colors are maintained by the color manager and provide the mapping from the application to the underlying hardware. Window colors are allocated by the application and are typically used by the graphics drawing functions. These colors are under control of the application program. Applications usually try to avoid working with system colors and stick to window color functions.

TWS does not support graphics systems with less than 16 available colors. It does not directly support any monochrome devices (although some transparently map colors to monochrome).

11.1. Look-up Table Color Systems

A lookup table (LUT) color system allows a certain fixed and usually small number of display colors. The color of each display color (and thus its table entry) is specified by its red, green and blue components (RGB value). The range of different colors possible in any table entry depends on how many bits the system allows for each R, G and B component. Most VGA systems support 6 bits for each component, while more advanced graphics systems support 8, 10 or more bits, depending on the device.

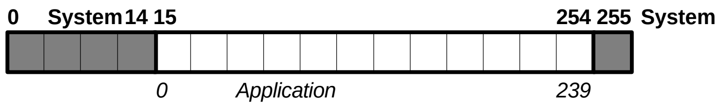
The size of the table varies, too. Standard VGA systems support only 16 entries in the table. So-called “Super VGA” often supports 256 entries, which is by far the most common arrangement among advanced graphics workstations.

TWS insulates the application programmer from much of this diversity. All colors are specified using 8-bit RGB values, and no limit (other than normal memory limitations) is placed on the creation of window color tables. TWS goes a long way to take care of mapping what the application does to the underlying hardware.

11.2. How TWS Manages LUT Colors

11.2.1. Organization of the System Color Table

The TWS color manager divides the hardware color palette into two sections, one where *system* colors are kept and the other for *application* colors. The system colors are used by the window manager for borders, gadgets, text, titles, icons and so forth. The system reserves 16 colors for this purpose — 14 fixed colors and two variable colors. Applications typically use but do not change colors in the system section.



The diagram shows the organization of colors on a system with 256 hardware colors. The application section is the pool that window color table colors are drawn from. The color manager swaps colors in and out of this area as necessary to meet the demands of the active window. Applications do change these colors, but only indirectly through assignments to window color tables.

On a standard 16-color EGA or VGA system, there are no free colors available for the application. In this case the application can only use the fixed system colors.

11.2.2. Window Color Tables

An application can draw using either the system colors (which can't be changed), or with its own custom colors. Before an application can allocate and use its own colors, it must create a graphics state (see the chapter on Graphics for details). The application can then create a color table as large as necessary and populate it with colors:

```
w = SM_NewWindow(&r, "Test Window", DIALOG, NULL, NULL);
GR_CreateGraphicsState(w, NULL, SMDEFAULT);
SM_CreateWindowPalette(w, 381);
for (i = 0; i < 381; i++) {
        r = GetRed(); b = GetBlue(); g = GetGreen();
        SM_SetWindowColor(w, i, r, g, b, SMSHARE | SMCLOSESTCOLOR);
}
```

Window color tables are separate from the system color table. When an application window becomes active, its color table is mapped into the application area of the system color table. There's no restriction on the size of a window color table, although most LUT hardware tables have 256 or fewer entries. In the example above, the application allocates 381 colors. We'll see how TWS handles these "extra" colors shortly.

11.2.3. Using Colors

An application can draw using either system colors or window colors. To retrieve and use a system color, call **SM_GetSystemColor**:

```
...
GR_CreateGraphState(w, NULL, SMDEFAULT);
GR_SetDrawColor(w, SM_GetSystemColor(SMWHITE));
GR_DrawLine(w, 1, 1, 25, 50);
```

To use a window color, call **SM_GetWindowColor**:

```
...
GR_CreateGraphState(w, NULL, SMDEFAULT);
InitWindowPalette(w);
GR_SetDrawColor(w, SM_GetWindowColor(w, 5));
GR_DrawLine(2, 1, 1, 25, 50);
```

In the above examples, specific colors are retrieved. It's also possible to request a color based on RGB values and have the color manager return the closest matching color:

```
...
GR_CreateGraphState(w, NULL, SMDEFAULT);
InitWindowPalette(w);
GR_SetDrawColor(w, SM_GetClosestColor(w, 185, 37, 204));
GR_DrawLine(2, 1, 1, 25, 50);
```

In the case above, the color manager will return the window color closest to the RGB value (185,37,204).
The application can also change gadget colors using either window or system colors. Gadgets usually ought to use system colors so they don't show the "technicolor" effect when active windows are changed.

11.2.4. Color Mappings, Sharing, and Merging

Obviously, if TWS colors are specified as 8-bit RGB values, and the hardware uses 6-bit values, somewhere along the line the TWS colors must be converted to hardware colors. This conversion is called a *mapping*. This mapping occurs when a window's color table is inserted into the system color table.

Another aspect of mapping colors is this: where in the system color table should a window color be stored? The simplest method would be to put the first window color into the first system color entry, the second color into the second, and so forth. However, this isn't what TWS normally does.

To map a color, TWS searches to see if there is already a color in the system color table that matches the window color. If so, that spot in the system color table is reused. The advantages are twofold: one, if the window that set the color is inactive, then the portions of the inactive window that use the shared color aren't changed when the new window becomes active. This reduces the unpleasant "technicolor" effect that's common in all graphical interfaces when active windows change; also, if a window is sharing a color with itself, it leaves more system color resources free for unique colors. Remember that it's possible to specify more colors in TWS (16.7 million) than the underlying hardware can display (262K for VGA). For example, the shades of gray (88,88,88) and (91,91,91) map to the same VGA hardware color (22,22,22). Sharing prevents colors that aren't unique when displayed from ending up as duplicates in the system color table.

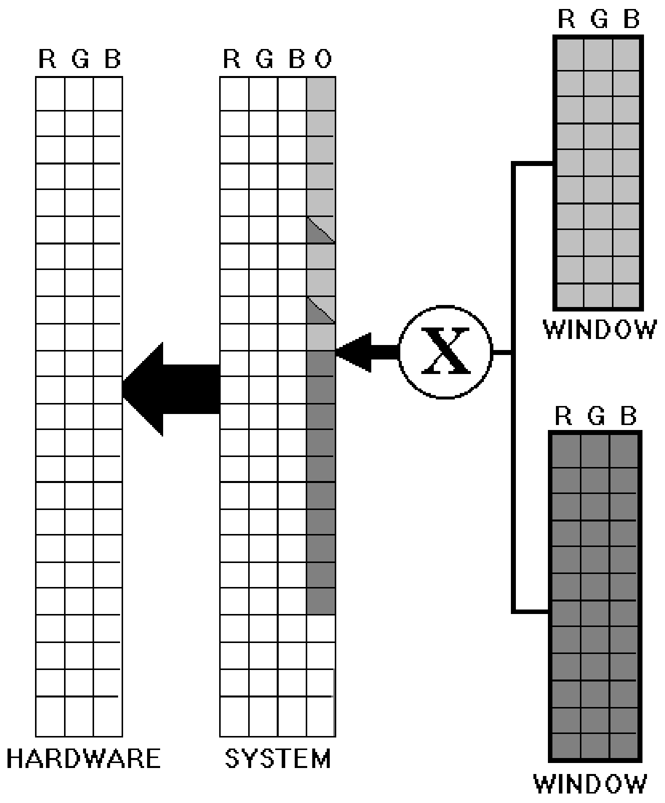
Merging allows an application to create window color tables that have more entries than the underlying hardware. When the TWS color manager discovers that a window that is requesting to set a color has already used up all system color table entries, it will scan through the portion of the window color table that has already been mapped and find the color that most closely matches the color requested. The new color then becomes equivalent to that closest relative. This is transparent to the application, which continues to treat the color as if it were unique.

On graphics kernel modes that only support 16 colors, the system color table takes all the colors and there are no spaces left for window colors. TWS takes not of this if color merging is enabled. On 16-color systems (such as standard VGA), a request for a closest window color will look instead at the system color table.

The SMSHARE flag turns on color sharing, while SMNOSHARE forces TWS to always allocate a unique system color table entry (and hence a unique hardware entry) for each requested color, or fail if it can't. Applications that expect to modify colors extensively would use SMNOSHARE. Merging of colors is specified by setting both the SMSHARE and SMCLOSESTCOLOR flags when the window color is set.

```
SM_SetWindowColor(w, n, r, g, b, SMSHARE);   // Set a window color with sharing
SM_SetWindowColor(w, n, r, g, b, SMNOSHARE); // Defeat sharing
SM_SetWindowColor(w, n, r, g, b, SMSHARE | SMCLOSESTCOLOR); // Sharing and merging
```

There's a slight but sometimes noticeable performance penalty for sharing and merging, but for most applications that do imaging the results are worth it.



The figure above illustrates the flow for TWS color allocations. The **X** symbol represents the mapping from window rgb colors to device rgb colors. The shading shows the ownership of system colors by individual windows. Blocks in the system color table **O**wner column that are dual shaded show colors that are allocated to both windows. Unshaded blocks in the **O**wner column are system colors that are not allocated.
Also see the following graphics functions: **GR_SetDrawColor, GR_SetBackgroundColor, GR_DrawArray, GR_GetBackgroundColor, GR_GetDrawColor.**

11.2.5. Data Types

TWS color information is kept in the ColorType data structure. The system color table and application window color palettes are both arrays of these.

```
typedef struct {
    unsigned char r,g,b;
    int index;
    void *owner;
    int flag;
    int count;
} ColorType;
```

The *r,g,b* values are the color components as requested by the application. *index* is the reference to the system color table for the color. Any number of individual window colors may have the same system color table index. *owner* is a reference to the window which allocated the color and is not usually interesting to the application. The color manager uses this value to determine which colors can be discarded when the focus window requests a color. Colors owned by the focus window are never discarded. The *flag* field is described in the **SM_SetWindowColor** function description below, and the *count* field is used internally and has no meaning to an application (but must never be changed).

11.2.6. Reserved Colors

The TWS system reserves 16 colors for its own use. These are used for window borders, gadgets, workspace content, and text for gadgets and window titles. Some of these colors can be modified when the system is initialized via the configuration file. The application itself usually does not modify these colors. The following table outlines the reserved colors:

Color Constant	Value	RGB
SMBLACK	0	0, 0, 0
SMDARKGRAY	1	96, 96, 96
SMMEDIUMGRAY	2	128, 128, 128
SMLIGHTGRAY	3	192, 192, 192
SMDARKRED	4	128, 24, 24
SMLIGHTRED	5	255, 48, 48
SMDARKGREEN	6	24, 128, 24
SMLIGHTGREEN	7	48, 255, 48
SMDARKBLUE	8	24, 24, 128
SMLIGHTBLUE	9	48, 48, 255
SMDARKYELLOW	10	128, 128, 24
SMLIGHTYELLOW	11	255, 255, 48
SMWHITE	256	255, 255, 255

There are three additional system color constants which do not have a fixed color value. They are:

Color Constant	Value	Default RGB
SMWORKSPACECOLOR	12	128, 128, 128
SMACTIVEWINCOLOR	13	168, 32, 32
SMACTIVEWINHILITE	14	255, 96, 96

The values for these system colors can be modified by the application through the TWS.CFG configuration file. The SMACTIVEWINCOLOR and SMWORKSPACECOLOR colors can be defined directly. The SMACTIVEWINHILITE color, which is the upper-left color for sculpting the active window's border, is calculated by the window manager based on the SMACTIVEWINCOLOR color.

11.2.7. Window Element Colors

The colors used for TWS window elements like borders, title bar and so forth can be changed by the application via the TWS.CFG configuration file, or through the **SM_SetElementColor** function.

Constant	Value	Default Value
INACTIVE_WINDOW_COLOR	0	SMMEDIUMGRAY
ACTIVE_TITLE_COLOR	1	SMWHITE
INACTIVE_TITLE_COLOR	2	SMDARKGRAY
ACTIVE_MENU_COLOR	3	SMACTIVEWINCOLOR
INACTIVE_MENU_COLOR	4	SMMEDIUMGRAY

ACTIVE_MENU_TEXT_COLOR	5	SMWHITE
INACTIVE_MENU_TEXT_COLOR	6	SMDARKGRAY
ACTIVE_WINDOW_BORDER_COLOR	7	SMACTIVEWINCOLOR
INACTIVE_WINDOW_BORDER_COLOR	8	SMMEDIUMGRAY
APPLICATION_TITLE_COLOR	9	SMACTIVEWINCOLOR
APPLICATION_TITLE_TEXT_COLOR	10	SMWHITE
DIALOG_CONTENT_COLOR	11	SMWHITE
DIALOG_BORDER_COLOR	12	SMACTIVEWINCOLOR
DIALOG_TEXT_COLOR	13	SMBLACK
DIALOG_GADGET_COLOR	14	SMMEDIUMGRAY

Window element colors can only be system colors. To change a window element color, use the function **SM_SetElementColor**:

```
SM_SetElementColor(DIALOG_CONTENT_COLOR, SMDARKRED);
```

11.3. Interface Functions *smcolor.h*

11.3.1. int SM_ActivatePalette(WindowType *w)

Loads the window's color table into the system color table and passes it along to the hardware. This function causes the display to immediately use the colors for the argument window. The window does not have to be the focus window. If the window has no color palette then nothing happens.

11.3.2. ColorType *SM_CreateColor(int r, int g, int b)

Allocates a ColorType structure, stores the (*r,g,b*) values in it and returns a pointer. All other ColorType fields are set to default values. This function is useful for quickly casting truecolor rgb values to TWS colors. Also see **SM_InitColor**.

11.3.3. ColorType *SM_CreateWindowPalette(WindowType *w, int n)

Creates a color table with *n* entries and attaches it to the window argument. The initial table colors are undefined. The owner field for each color is set to the window and the flag field is set to *normal.*

11.3.4. ColorType *SM_GetClosestColor(WindowType *, int r, int g, int b)

Returns the color from the window's color table that is closest to the requested rgb color. If the argument window has no local color table then NULL is returned.

11.3.5. int SM_GetElementColorIndex(int element)

Returns the color constant of the system color associated with the *element*. The range of element constant values is described in the preceding section, as well as the system color constants. If the value of *element* is out of range then -1 is returned.

11.3.6. ColorType *SM_GetPalette(WindowType *w)

Returns a pointer to the window's local color palette.

11.3.7. ColorType *SM_GetSystemColor(int n)

Returns a pointer to the *n*th color in the system color table. If *n* is out of range for the system color table then NULL is returned.

11.3.8. ColorType *SM_GetSystemPalette()

Returns a pointer to the table of colors used by the graphics display. *This list should be treated as Read-Only* since any changes to the system palette will affect other system color functions.

11.3.9. ColorType *SM_GetWindowColor(WindowType *w, int n)

Returns the color from the argument window's palette at index *n*. If there are not *n* colors in the window palette, or if the *nth* color hasn't been assigned then the return values are indeterminate.

11.3.10. void SM_InitColor(int r, int g, int b, ColorType *c)

Stores the (*r,g,b*) values into the ColorType struct *c*. The remaining fields in *c* are set to default values and any existing values are lost. Also see **SM_CreateColor**.

11.3.11. int SM_IsColorEqual(ColorType *c1, ColorType *c2)

Returns True if the two colors are the same color, False otherwise.

11.3.12. void SM_ModifySystemColor(int i, int r, int g, int b)

Modify a color in the system color table. *i* is the index into the system table to change. The new color will be the closest hardware equivalent to *r,g,b*, the RGB color values given in 8-bit (0..255) primaries. If the index value is outside the range of valid system colors, the function does nothing.
***Note**: This function should be used with caution, since the TWS system doesn't know that its understanding of the underlying colors is changed. For example, using this function, an application can change the system color* SMDARKRED *to a beautiful chartreuse. TWS will continue to think it's dark red, however.*
See also: **SM_SetElementColor**.

11.3.13. int SM_ModifyWindowColor(WindowType *w, int n, int r, int g, int b)

Change the color value at window palette color *n* to the new value *r,g,b*. Color values must be in the range of 0..255. The window color palette must already be allocated with at least *n* entries. If *w* is the focus window then the screen display is updated immediately, otherwise the colors will be updated when *w* is next active.

11.3.14. int SM_nApplicationColors()

Returns the number of colors the application can support, normally **SM_nSystemColors()** - 16.

11.3.15. int SM_nColorBits()

Returns the number of bits of color supported by the hardware device in use. For standard VGA the value returned is 6. TWS supports up to 8-bits of color. *Note: for most hardware color systems there is no reliable way to determine the number of bits per primary color. In most cases, therefore,* **SM_nColorBits** *simply returns a default value of 6.*

11.3.16. long int SM_nSystemColors()

Returns the number of colors supported by the underlying graphics system. This number is either 16 or 256 for VGA systems, or 32,767 for hicolor systems.

11.3.17. int SM_nWindowColors(WindowType *w)

Returns the number of colors allocated for the window's color palette. This value should be less than or equal to **SM_nApplicationColors()**.

11.3.18. void SM_SetElementColor(int element, int constcolor)

Sets the color of the window *element* to the value of *constcolor*. The values for respective constants are described in the preceding section. If *element* or *constcolor* are out of range then nothing happens.

11.3.19. int SM_SetWindowColor(WindowType *w, int n, int r, int g, int b, int flag)

Set the window color table entry at index *n* to the color *r,g,b* and set the color flag to *flag*. By default the function tries to satisfy the request by finding a system color that matches *r,g,b* and reusing it, allocating a system table entry not already owned by the window if that fails.
The *flag* value can modify this behavior. The flag values and their meanings are:

Flag	Value	Description
SMSHARE	1	The default behavior. Try to find a color in the system table that already matches the requested RGB color and reuse it. If that fails, find an unused color and allocate it. If there are no unused colors then return an error.
SMNOSHARE	2	Must allocate a unique system color table slot for this color, otherwise fail. This would be required if the application is going to change the color values dynamically ("color cycling").
SMCLOSESTCOLOR	4	Just find the color already in the window color table that is closest to the requested color and return it. This is useful when the application wants to set a 'fixed' color palette and force the application to only use the fixed colors.
SMFLAGDEFAULT	SMDEFAULT	Follow the default behavior rules.

Flag values can be combined, but the only the combination SMSHARE | SMCLOSESTCOLOR makes any sense. With this combination, a color request will always succeed.
The *r,g,b* values are in the range [0..255] regardless of the underlying hardware.

11.4. Truecolor Color Systems

Truecolor systems do not use look-up tables. Instead, every screen color is specified using a RGB triplet. Typical systems support 4, 5 or 8 bits for each RGB component, for 4k, 32k, or 16.7M screen colors.
Remember that regardless of the underlying color depth, ColorType values are always specified using 8-bits of precision, so 0 is full off and 255 is full on for each RGB component. TWS colors are mapped to the underlying hardware colors transparently.
When TWS discovers that the graphics kernel mode is a truecolor mode[8] it shifts gears a bit. For one thing, it doesn't create a system color table -- it's not needed. For another, the *flag* field in the ColorType struct and in the **SM_SetWindowColor** function no longer have meaning. For drawing purposes, the system uses the RGB values directly.

[8]For our purposes, "truecolor", or 24-bit-per-pixel colors, is the same as "hicolor", or 15 to 16-bits-per-pixel color.

All of this happens behind the scenes. Many programs will not have to change any source code to move from LUT color systems to Truecolor systems. The window graphics state still supports color palette tables, which can be accessed by index, and the color manager will adjust as necessary. So, for example, you could write a program to display 256-color PCX images, and the same executable would run identically on four or eight bit LUT systems, or 15 or 24-bit truecolor systems. In fact such a program is included in the TWS distribution (WINIMAGE).
Obviously an application can make the most effective use of colors if it pays attention to the actual system capabilities. An application that knows it's running with 15-bit or 24-bit color can offer the user more choices than one that always behaves as if it's using 256 or fewer colors. Likewise, an application that realizes it only has 16 colors to choose from will probably make more intelligent use of them than one that ignores the fact. The point of the color manager is to provide *reasonable* behavior in a large number of cases.

Part Three

Gadgets

12. Gadgets

Gadgets are direct manipulation user interface devices that an application attaches to its windows. Along with menus, gadgets are the primary user interface structures for TWS. Gadgets are built in two parts. The ‘generic’ gadget *GadgetType* is the ‘superclass’[9] of all gadgets, containing data and function hooks common to all gadgets. Each gadget also has specific data and functionality that is unique to it.

12.1. The Generic Gadget

The generic gadget is the parent, or foundation class, of all TWS gadgets. It has the following fields:

```
typedef struct _gadget {
    void        *gadget;               /* Specific gadget fields struct    */
    struct _win *parentwin;            /* Window containing the gadget     */
    RectType    bound;                 /* Sensitive region for the gadget  */
    FontType    font;                  /* Font to draw gadget with         */
    int         width, height;         /* Font width and height            */
    int         face;                  /* Font facing                      */
    short       gcode;                 /* Keyboard code (accelerator)      */
    int         type;                  /* Type of gadget subclassed        */
    unsigned    msg;                   /* Event types gadget responds to   */
    int         nofree;                /* If True gadget not freed on close*/
    int         redraw;                /* If True gadget needs redrawing   */
    ColorType   *backcolor, *forecolor; /* Gadget drawing colors           */
    int         (*drawproc)(struct _gadget *);      /* Draw the gadget      */
    int         (*deleteproc)(struct _gadget *);    /* Delete gadget        */
    int         (*shellproc)(struct _gadget *,EventType *);
    int         (*userproc)(void *);    /* Gadget user procedure           */
    struct _gadget *next, *prev;       /* Links                            */
    } GadgetType;
```

All gadgets attached to windows are of type GadgetType. Specific gadgets are attached to a GadgetType parent at the *gadget* field. Functions to draw and delete the *specific* gadget are attached to the *generic* gadget so that the window manager can rapidly manage gadgets as necessary. The *drawproc*, *deleteproc* and *shellproc* arguments are a pointer to a GadgetType rather than to the specific gadget type.
The argument to the *userproc* function, however, is a pointer to the specific gadget. The user procedure is called by *shellproc*. One of the shell procedures tasks is to extract the specific gadget record from its parent gadget argument.

12.2. Modifying Gadget Attributes and Redrawing Gadgets

When a visual element of a gadget is changed, the gadget display is updated the next time the window is drawn.[10] The application can force redrawing of all gadgets that need it by calling the **SM_RefreshGadgets** function.
The exceptions to this rule are changes that affect the gadget’s boundary. For example, the **SM_AdjustGadgetBound** function *immediately* erases the old gadget, if the gadget’s parent window is active. However, the new gadget is *not* drawn until either the parent window is redrawn or the application calls **SM_RefreshGadgets**.

[9]TWS is not overly ‘object-oriented’ in any rigorous sense. Use of terms like ‘superclass’, ‘foundation class’, ‘subclass’ and so forth is intended to convey the relationship between generic and specific gadgets. If it feels more natural to think of this in simple ‘parent’ and ‘child’ relationships, please feel free.

[10]This is new behaviour. In previous versions of TWS, when a gadget that was part of the active window was changed, the gadget was immediately redrawn.

12.3. Specific Gadgets

A user application deals with generic gadgets rarely or never. An application deals with window, or specific gadgets. A window gadget is a 'subclass' of the generic gadget GadgetType in that when an application creates a gadget, like a pushbutton or string list, that *specific* gadget is attached to a generic GadgetType parent.

12.4. Generic Gadget API functions [smgadget.h]

All specific gadgets have unique attributes that are addressed with their own API functions, like **SM_GetLabelData** for retrieving the user data field of a label. Some attributes are common to all gadgets. For these situations there are a collection of functions that accept any gadget as an argument. These functions are described below.
To use a generic gadget function, pass a pointer to any specific gadget. For example, to change the background color for a button and a label:

```
ButtonType     *button;
LabelType      *label;

/* The button and label are created elsewhere.  Set the colors */
SM_SetGadgetBackcolor(button, SM_GetSystemColor(SMDARKRED));
SM_SetGadgetBackcolor(label, SM_GetSystemColor(SMMEDIUMGRAY));
```

12.4.1. void SM_AdjustGadgetBound(void *g, int dx, int dy, int dwidth, int dheight)

Adjust the position and size of the boundary rectangle for the specific gadget *g*. The gadget is moved a distance *dx,dy*, where positive numbers are to the right and down. The gadget boundary size is modified by *dwidth* and *dheight* pixels, where positive values make the gadget wider/taller, respectively.
A gadget's boundary coordinates can be negative. If the gadget ends up outside the window content, the gadget is clipped and may not be visible. The gadget's width and height must always be greater than 0, however.
Adjusting a gadget's boundary has no affect on the contents of the gadget. If the gadget's parent window is the active window, the old gadget is erased. *See also: SM_SetGadgetFontSize*.

```
LabelType      *label;

/*
** ...
*/
SM_AdjustGadgetBound((void *)label, -15, 10, 25,12);
```

12.4.2. int SM_AttachGadget(WindowType *w, GadgetType *gadget)

Attaches the gadget *gadget* to the window. Assumes the gadget isn't already attached to some other window -- if it is it could be confusing to the window manager. Returns 0 on success, any other value indicates failure.

12.4.3. int SM_CloseGadgets(GadgetType *g)

Removes the gadget *g* and all gadgets that were added to the window after *g*. If *g*'s parent window has the focus then it'll be redrawn without the destroyed gadgets. Typically applications don't need to close window gadgets directly, since the window manager does so itself when a window is closed.

12.4.4. SM_DestroyGadget(void *g)

Removes the gadget *g* from it's parent window and frees all resources used by it. Has the same effect as calling the specific user gadget destructor function for the same gadget. *See also: Specific gadget destructor functions.*

12.4.5. WindowType *SM_GadgetToWindow(void *gadget)

Returns a ‘free’ window whose content and graphics canvas are the same dimensions as the boundary of the gadget passed in. This function is used to create a ‘window’ for drawing graphics inside a gadget. The argument *gadget* is a pointer to any specific user gadget, such as a ButtonType * or a LabelType *. Although **SM_GadgetToWindow** will work with any gadget, it is mainly used for labels and buttons.

The window returned by **SM_GadgetToWindow** is not on the window stack and therefore isn’t managed by the window manager. The window’s content region is the size of the gadget’s boundary. The window has a graphics state whose canvas size is the same size and position as the gadget’s boundary, and, if the parent window of the gadget has a color palette, the palette is copied to this new window. All other window and graphics state parameters have their default (perhaps undefined) values.

The resulting window can be used as an argument to any TWS function that requires a window, as long as the function doesn’t modify the window’s size or position, and doesn’t rely on the window being on the stack. The window mustn’t be closed.

The main use for this function is as part of a GraphProc for a ButtonType or LabelType gadget. The GraphProc is a function that draws the gadget face or content. Since the only way to do graphics in TWS is within a window’s graphics state canvas, **SM_GadgetToWindow** is a convenient way to get a window just the same size as the gadget, draw in it, then free it and return. This would happen every time the gadget is redrawn.

Use the window function **SM_FreeWindow** to free the window when it’s no longer needed.

```
#include       <smgadget.h>
#include       <smcolor.h>
#include       <smwindow.h>
#include       <smlabel.h>
int DrawPalette(LabelType *label)
{
    WindowType  *w;
    int         width, depth;
    float       ddx, dx;
    int         i, x, y;
    ColorType   *c;
    RectType    rt;

    /*
    ** Cast label to a window so we can draw in it
    */
    w = SM_GadgetToWindow(label);

    width = SM_GetContentWidth(w);
    depth = SM_GetContentDepth(w);
    ddx = (float)SM_nWindowColors(w) / (float)width;
    dx = -1.0;
    x = 0;
    /*
    ** Draw the colors in the label, where there’s
    ** approximately the same number of lines for
    ** each color
    */
    for (i = 0; i < SM_nWindowColors(w); i++) {
        c = SM_GetWindowColor(w, i);
        GR_SetDrawColor(w, c);
        while (dx <= 0.0) {
            GR_DrawLine(w, x, 0, x, depth);
            x++;
            dx += ddx;
        }
        dx -= 1.0;
    }
    /*
    ** Free the temporary window
    */
    SM_FreeWindow(w);
    return 1;
}
```

12.4.6. GadgetType *SM_GetFocusGadget(WindowType *w)

Returns a pointer to the focus gadget for the window, or NULL if no gadget has the focus. The window does not have to be the active window or mapped to the screen.

12.4.7. RectType *SM_GetGadgetBound(void *g)

Returns a pointer to the boundary rectangle of the gadget argument. *g* is a pointer to a *specific* gadget, like a button or label. The returned rectangle is a pointer to the gadget structure field, not a copy, and should be considered read-only.
The values of the rectangle coordinates are exactly as they were specified when the gadget was created. This is not necessarily the current gadget position within the window. For example, if the gadget was created with virtual coordinates, the virtual values (negative integers) are returned.

12.4.8. ColorType *SM_GetGadgetBackcolor(void *g)

Returns the background color for the gadget *g*.

12.4.9. ColorType *SM_GetGadgetForecolor(void *g)

Returns the foreground color for the gadget *g.*

12.4.10. int SM_GetGadgetMsg(void *g)

Returns the event type mask that defines what hardware events the gadget will respond to. For example, a button gadget responds to LEFTBUTTONACTIVE events. Event constants are defined in smevent.h. *See also: SM_SetGadgetMsg; Events chapter*.

12.4.11. GadgetType *SM_GetGadgetSuperclass(void *)

Returns the generic gadget field for any specific gadget. Given a pointer to a ButtonType *button*, for example, returns *button->gadget*.

12.4.12. int SM_GetGadgetType(GadgetType *g)

Returns a constant identifying the type of gadget (button, label, etc.). Constants are defined in smtypes.h :

Constant	*Value*
BUTTON	1
CHECKBOX	2
SMSCROLLBAR	16
LABEL	4
EDITSTRING	5
PIXMAP	17
SLIDER	11
STRINGLIST	12
CHECKBOXGROUP	13
TEXTBOX	14
ROTATELIST	18

STATICBORDER	19
HOTREGION	20

12.4.13. WindowType *SM_GetGadgetWindow(void *g)

Returns the parent window of the gadget passed in. The pointer *g* is a pointer to any TWS specific gadget (like a button). This is a pointer to the system structure, not a copy of it, and should be treated as read-only.

```
int ButtonProc(ButtonType *mybutton)
{
        WindowType     *gw;

        gs = SM_GetGadgetWindow(mybutton);
        if (gs == SM_FocusWindow()) {
         .......    ...
```

12.4.14. void SM_SetFocusGadget(void *gadget)

Sets the focus gadget for *gadget*'s parent window to *gadget*. If the gadget's parent window is the active window then the gadget is redrawn.

12.4.15. void SM_SetGadgetBackcolor(void *g, ColorType *color)

Sets the background color for the gadget *g*. This can be any system or window color but system colors are recommended. The application of background colors varies from gadget to gadget.

12.4.16. int SM_SetGadgetDeleteproc(void *g, int (*f)())

Sets the function called whenever the gadget is deleted. The function is an int function whose single argument is a pointer to a GadgetType. The function must remove the gadget from the gadget list and free all resources used by the gadget. *See also: Writing New Gadgets chapter in the TWS Source Code Developers Manual*.

12.4.17. int SM_SetGadgetDrawproc(void *g, int (*f)())

Sets the function that will actually draw the gadget. Currently the window gadget manager ignores this field, so setting it will not change the appearance of the gadget. This behavior will be changed in a future release. *See also: Writing New Gadgets chapter in the TWS Source Code Developers Manual.*

12.4.18. void SM_SetGadgetFont(void *g, char *font)

Set the font used for the gadget's text or label. If the gadget does not output text (a slider, for example), the function is harmless. *font* is the filename, including path if necessary, to a font.

12.4.19. void SM_SetGadgetFontFacing(void *g, int facing)

Sets the facing (bold, italic) that will be applied to the text for the gadget. If the gadget doesn't output text then the function is harmless. *Note: not all graphics kernel systems support font facing attributes.*

12.4.20. void SM_SetGadgetFontSize(void *g, int size)

Sets the size of the font that will be used for the gadget. Only valid if the gadget font is a stroked font and the gadget outputs text; harmless otherwise.

12.4.21. void SM_SetGadgetForecolor(void *g, ColorType *color)

Sets the color used for the gadget's foreground. Normally this is the text color for labels, buttons, and so forth. The application of foreground colors varies from gadget to gadget.

12.4.22. int SM_SetGadgetMsg(void *g, int evntmask)

Sets the event type mask that defines what hardware events the gadget will respond to. Multiple event types can be specified by ORing the appropriate constants, in which case the gadget will respond if any one of the events occurs. Event constants are defined in smevent.h. *See also: SM_GetGadgetMsg; Events chapter*.

12.4.23. void SM_SetGadgetRedrawFlag(void *g, int flag)

Sets the redraw flag for the specific user gadget *g* to the value of *flag*. Any non-zero value for flag is considered True. The argument *g* is a pointer to any specific user gadget, such as a button. *See also: SM_RefreshGadgets.*

12.4.24. int SM_SetGadgetUserproc(void *g, int (*f)())

Sets the application function called when the gadget *g* is activated by the user. The result is the same as if the function *f* was passed when the gadget was created. The function *f* must be an int function whose argument(s) are appropriate for the specific user function attached to the generic gadget *g*. *See also: SM_GetGadgetType; Specific gadgets chapters.*

13. Label

A label is a rectangular area that can contain a text string or graphics. There is no user action associated with a label as there is with a button or checkbox. Labels can't be selected and the event system doesn't notice activity occurring within a label.
A label's bounding box can be bordered or unbordered. If bordered the window bounding rectangle describes the outer bounds of the border, but drawing is done within the border; in effect, pixels are lost on each edge. The size of the border depends on the bevel depth.
The bounding rectangle can be either in *absolute* window coordinates, where the positions represent window content relative pixels, or *virtual* window coordinates, where the positions are given in 1/1000's of the width or depth from the upper left corner of the window content. Virtual coordinates are specified by using negative numbers in the range (-999 ... -1) for one or more of the rectangle's coordinates. Labels specified using *virtual* coordinates keep their same relative positions in the window content region, though their absolute size and shape may change as the window size and shape changes. Virtual coordinates are described in detail in the **Gadgets** chapter.

13.1. Data Types

```
typedef struct _lbl {
    GadgetType *gadget;      /* Generic gadget superclass        */
    int     boldflg;         /* TRUE if string is bolded         */
    int     italflg;         /* TRUE if string is italicized     */
    int     boxflg;          /* TRUE if string is boxed          */
    char    *string;         /* The text to be displayed         */
    int     align;           /* Text alignment left right center */
    void    *data;           /* Application data                 */
    int     (*graphproc)(struct _lbl *); /* Graphics label       */
    int     sculptype;       /* Bordering method                 */
} LabelType;
```

Not all graphics kernel systems support text attributes like bolding and italics, in which case the *boldflg* and *italflg* are ignored. If *boxflg* is non-zero then the boundary of the label is drawn in the style specified by *sculptype*. Constants for supported styles are defined in smtypes.h:

Constant	**Value**	**Description**
SCULPTED	1	Recessed 3D beveled
FOURD	2	Not implemented
FLAT	3	Plain black solid line
BEVELED	4	Raised 3D beveled
CHISELED	5	Chiseled line. Ignores the bevel depth, requires 2 pixels of border all around.

Text alignment constants are also defined in smtypes.h. Possible values are:

Constant	Value	Description
ALIGNLEFT	0	Draw text from left edge of boundary rectangle
ALIGNCENTER	1	Center the text within the boundary rectangle
ALIGNRIGHT	2	Draw text with end of the string on the right edge of the boundary

If the text is wider than the boundary (bordered or not) it is clipped within it. The *data* field is for application data and is provided mostly for symmetry with other TWS gadgets.
The *graphproc* is an application subroutine for drawing graphics within a label boundary. If there's a graphproc attached to a label then the label string is ignored and the graphproc is called instead. The graphproc can do anything (it doesn't have to draw graphics, although if it doesn't there won't be anything displayed in the label), but typically will take the label, cast it to a window using the **SM_GadgetToWindow** function, perform some quick drawing, free the window and return.
A graphproc label is similar to the Pixmap gadget. The main difference is that a Pixmap is a static image while the graphproc label is recreated every time it's redrawn, and so is potentially different each time.

13.2. Interface Functions *smlabel.h*

13.2.1. LabelType *SM_CreateLabel(WindowType *w, RectType *r, char *str, int (*graphproc)(), int align, int boldflg, int italflg, int boxflg, void *data)

Create a new label gadget and return a pointer to it.

w	window the label will belong to;
r................	boundary of the label in window coordinates;
s	string to be displayed in the label;
graphproc ..	If not NULL, the function for drawing graphics within the label bounds. The function must 'cast' the label to a window to draw inside it (see **SM_GadgetToWindow**).
align	Text alignment within the label, either ALIGNLEFT, ALIGNRIGHT or ALIGNCENTER. The constants are defined in smtypes.h
boldflg	If True the label string has the bold attribute;
italflg	If True the label string is italicized;
boxflg	If True the label is outlined
data	Application data

13.2.2. void SM_DestroyLabel(LabelType *label)

Frees a label and all resources allocated to it. If the label's parent window is the focus window then the window is redrawn immediately.

13.2.3. int SM_EraseLabel(LabelType *label)

Clears the label to the window content color. This does *not* remove the label string or graphics procedure, so the next time the window is refreshed the label will be restored.

13.2.4. int SM_GetLabelBoldflag(LabelType *label)

Returns the value of the label's *boldflg* field, either True or False.

13.2.5. int SM_GetLabelBoxflag(LabelType *label) *(macro)*

Returns the value of the label's *boxflg* field, either True or False.

void SM_GetLabelBound(LabelType *label, RectType *r)

Returns the rectangle boundary for the label. The rectangle coordinates are relative to the label's parent window. *See also: SM_GetGadgetBound*

13.2.6. void *SM_GetLabelData(LabelType *label)*(macro)*

Returns the data field of the label. If there is no data attached to the label then NULL is returned.

FontType SM_GetLabelFont(LabelType *label) ***(macro)***

Returns the font used to draw the label. *See also: SM_GetGadgetFont*

13.2.7. int SM_GetLabelItalflag(LabelType *label)

Returns the value of the label's *italflg* field, either True or False.

GadgetType *SM_GetLabelGadget(LabelType *label) ***(macro)***

Returns the superclass gadget for the label. *See also: SM_GetGadgetSuperclass*

13.2.8. char *SM_GetLabelString(LabelType *label) *(macro)*

Returns the string attached to the label. If there is no string (as when the label is a graphics label) then NULL is returned.

13.2.9. int SM_SetLabelBoxflag(LabelType *label, int flag)

Sets the value of the label's *boxflg* field to *flag*, which must be either True or False.

13.2.10. int SM_SetLabelBoldflag(LabelType *label, int flag)

Sets the value of the label's *boldflg* field to *flag*, which must be either True or False.

13.2.11. void SM_SetLabelData(LabelType *label, void *data) *(macro)*

Sets the *data* field for the label. Data is often a pointer to a user data structure but the interpretation is up to the application.

void SM_SetLabelFont(LabelType *label, FontType font)

Sets the font for the label's text. *See also: SM_SetGadgetFont*

13.2.12. void SM_SetLabelGraphproc(LabelType *label, int (*f)())

Sets the graphics procedure for a label to *f*. If a label has a non-NULL value for its *graphproc* field then the procedure is called instead of drawing a string. A label can be changed from a graphics to a text label by passing NULL as the function argument of **SM_SetLabelGraphproc**.

13.2.13. int SM_SetLabelItalflag(LabelType *label, int flag)

Sets the value of the label's *italflg* field to *flag*, which must be either True or False.

void SM_SetLabelBound(LabelType *label, RectType *r)

Sets the rectangle to be used for the label boundary. The rectangle coordinates are relative to the label's parent window. *See also: SM_SetGadgetBound, SM_AdjustGadgetBound*

13.2.14. void SM_SetLabelSculptType(LabelType *label, int type)

Sets the sculpting type to *type*. The type should be one of the constants described above.

13.2.15. int SM_SetLabelString(LabelType *label, char *string)

Sets the string attached to the label to *string*. If there is already a string attached to the label it is freed.

14. Buttons

A button is a window control type that simulates a spring-loaded pushbutton. The application attaches a function to the button that is called whenever the button is 'pushed.' A button is pushed by clicking the left mouse button on it; pressing the left mouse button down causes the button to be 'pressed into' the screen. When the mouse button is released, if the cursor is still inside the button then the button procedure is called, otherwise the button is restored and no action is taken. If the button is defaulted then it is also sensitive to the Enter key being pressed.
A button can have a text label, a graphic facing and/or a Pixmap facing. The text label is assigned when the button is created. The function for applying a graphic is supplied by the application and is attached to the button using the function **SM_SetButtonGraphproc**. To draw a graphic in a button the graphics procedure 'casts' the button to a window using the **SM_GadgetToWindow** function, then use graphics routines to decorate the button face. See the **Gadgets** section and the example code in the distribution.
A Pixmap can be applied to the button face using the function **SM_SetButtonPixmap**. See the Pixmap section for information on creating TWS Pixmaps.
A single button can use any or all of these decorations. If a Pixmap is attached to a button, it is displayed first. A graphics function draws on top of that, and the text is drawn last. Finally, the position of all the visual elements of the button can be controlled so that, for example, a pixmap image of an object can appear next to the text name of it.
A button's position can be either fixed or dynamic (called *virtual* positioning) within the window's content region. For fixed size and position, the button rectangle is specified using window-content-relative absolute coordinates, for example (1,1,50,25). Virtual size and position is specified by using negative coordinates for the button rectangle. A negative coordinate is decoded as thousandth's of the distance from the upper left corner in the X or Y direction. For example, (1,1,-999,25) specifies a button whose upper left corner is 1 pixel from the upper left corner of the window content region, and whose lower right corner is as wide as the content region and 25 pixels from the top edge. The rectangle (-500,-500, -999, -999) fills the lower right quadrant of the content region no matter what the window size is. For virtual positioning the range of allowed values is (-999 ... -1). See the **Gadgets** chapter for more details.

14.1. Data Types

```
typedef struct _but {
    GadgetType  *gadget;              /* 'Superclass' gadget             */
    char        *label;               /* Text for button interior        */
    int         align;                /* Button label horizontal alignment*/
    ColorType   *color;               /* Button face color               */
    ColorType   *pcolor;              /* Button face color when pushed   */
    ColorType   *up_bevel_color;      /* Bevel colors                    */
    ColorType   *down_bevel_color;
    ColorType   *labelcolor;          /* Color for text label            */
    ColorType   *plabelcolor;         /* Color for text label when pushed */
    void        *data;                /* User-specified data for button  */
    PixmapType  *pixmap;              /* Button pixmap                   */
    int         pmx, pmy;             /* Pixmap offset into the button   */
    int         active;               /* if TRUE button is active        */
    int         isdefault;            /* if TRUE button activate by Enter */
    int         pushed;               /* If True button is pushed in     */
    int         (*graphproc)(struct _but *);    /* Draw in the button       */
} ButtonType;
```

The button *label* is a text string drawn in the center of the button. If the string is wider than the button it's clipped within it.
A button has a separate set of colors for the normal and "pushed" states. *color, labelcolor* and the up and down bevel colors are the normal state colors, while *pcolor*, *plabelcolor* and the bevel colors (used on the opposite corners) are the pushed state colors. The *graphproc* is the function that draws a graphic in the button and is NULL by default. *pixmap* also defaults to NULL.
If *active* is false then the button is disabled and won't respond to user input. The default is True.
When *isdefault* is True the button responds to the Enter key as well as mouse button press, and is also drawn with a special sculpting to indicate it's a default button. Finally, when *pushed* is True the button is in a pressed state. The application wouldn't ordinarily notice this state since the button is restored to normal when the button callback function returns.

14.2. Button Callback Function

The user function attached to the button is called each time the button is activated. It is an int function whose argument is a pointer to a ButtonType. The button has a 'pressed in' look while the user procedure is executing. The button is restored to its normal sculpted appearance when the user function returns.
The button callback function can do just about anything, including closing the button's parent window (there's special logic in the TWS event processor for handling this).

14.3. Default Button

If a button is a *default* button then the button is sensitive not only to mouse events within its boundary, but also to the Enter key. If the Enter key is pressed when the button's parent window is active, then the button's user procedure is called.
Visually, a *default* button has an extra bevel drawn around it. If more than one button in a window is a default button then all of them will be drawn with the default button visual; however, only the *first* button's user procedure, determined by the order the buttons were created, will be called. It is usually an error to classify more than one button as a *default* button. A button is made a default button by calling the **SM_SetButtonIsdefault** function.

14.4. Button Colors

A button has two sets of colors; one when the button is off, and the other when it is pushed in or selected. The application can set the color for the button face and text for both situations. Be default, the button text color when off is black and the face color is the window's content color. When pushed the text color is white and the face color is dark gray. The functions **SM_SetButtonColor, SM_SetButtonFaceColor, SM_SetButtonPushedColor,** and **SM_SetButtonPushedLabelColor** set these colors.
For buttons, the generic gadget functions **SM_SetGadgetBackcolor** and **SM_SetGadgetForecolor** modify the button's unpushed face color and label color, respectively.

14.4.1. Transparent Buttons

A button can also be transparent. A transparent button shows an outline and whatever text, pixmaps and graphics the application draws in it, but otherwise the background shows through. This is convenient for placing buttons over imagery. To make a button transparent, set the background color to TRANSPARENT:

```
SM_SetButtonFaceColor(TRANSPARENT);
```

Only a button's background color can be transparent, and the pushed and non-pushed background colors must be set separately (if only one is set to TRANSPARENT then the transparency effect won't work).

Pixmaps in Buttons

An application can decorate a button by applying a pixmap to it. The pixmap can be any size and can be positioned anywhere within the button boundary, with (0,0) as the upper-left corner of the button. The pixmap is clipped at the button boundary. The button beveling is drawn inside the button boundary and after the pixmap is placed so the application should allow space as necessary. The TWS function **SM_SetButtonPixmap** attaches the pixmap, which must already be created:

```
RectType       r;
ButtonType     *button;
PixmapType     *pixmap;
/*
** Set the rectangle for the button.  Make it a little wide for the pixmap
*/
r.Xmin = r.Ymin = 10;
r.Xmax = r.Xmin + GR_StringWidth("New file") + 27;
r.Ymax = r.Ymin + 24;
button = SM_CreateButton(w,
                         &r,
                         "New file",
                         NULL,
                         ButtonUserproc);
/*
** Align the button text to the right instead of middle.  The pixmap will
** be on the left edge
*/
SM_SetButtonAlign(b1, ALIGNRIGHT);
/*
** Create a pixmap from an existing file
*/
pixmap = SM_ReadPixmap("\\pixmaps\\folder.pxm");
/*
** Attach the pixmap to the button.  Offset it from the edge so the button
** bevel doesn't clobber it
*/
SM_SetButtonPixmap(b1, pm, 4, 4);
```

User Graphics in Buttons

For even more flexibility, an application can create a routine to draw inside a button as if it were a graphics canvas. The routine is attached to the button using the **SM_SetButtonGraphproc** function:

```
button = SM_CreateButton(w, &rect, "My Button", NULL, MyButtonProc);
SM_SetButtonGraphproc(b, ButtonGraphProc);
```

The graphics procedure (ButtonGraphProc in the example) is an int function with a ButtonType pointer argument. When called, the argument will reference the button that was pushed. In order to draw, the function must first 'cast' the button to a window using the function **SM_GadgetToWindow**:

```
int ButtonGraphProc(ButtonType *button)
{
       WindowType     *w;

       w = SM_GadgetToWindow(button);
```

SM_GadgetToWindow sets the window and gives it a graphics canvas that's the same size as the button. If the button's parent window had a graphics canvas with a color palette, *w* gets a copy of the color palette as well. At this point the window *w* can be drawn into using the TWS graphics functions. Location (0,0) is the upper-left corner of the button, and all graphics will be clipped to the button boundary.

Before returning, the window *w* must be freed:

```
SM_FreeWindow(w);
```

14.5. Active and Inactive Buttons

The application can make a button inactive, then reactivate it at any time. By default a button is active as soon as it is created (or as soon as its parent window becomes active). When a button is inactive, its face is 'greyed' and it will not respond to input events. The function **SM_DeactivateButton** makes a button inactive, and **SM_ActivateButton** reactivates it.

Button Repeat

Setting a button's *repeat* field causes a pushed button's action to repeat while the button is pushed, following an initial delay of about a half-second. If the mouse button is released the repeat action stops. If the mouse cursor is moved outside of the button while the mouse button is pressed, the button action pauses until the cursor moves back inside the button, or the mouse button is released.
Button repeat is off by default. It can be set using the macro **SM_SetButtonRepeat**.

14.6. Interface Functions *prototypes in smbutton.h*

14.6.1. SM_ActivateButton(ButtonType *b)

Activates an inactive button. When a button is inactive its face is 'greyed' and it doesn't respond to input events. If the button is already active then nothing happens. If the button's parent window is the active window then the button is immediately redrawn.

14.6.2. ButtonType *SM_CreateButton(WindowType *w, RectType *r, char *label, void *data, int (*action)())

Create a new button and return a pointer to it. If the parent window *w* is not NULL then the new button is attached to it. *action()* is an integer function whose single argument is a pointer to a ButtonType. The button rectangle *r* is specified in window content relative coordinates. If any coordinate of *r* is negative then it is treated as a virtual coordinate as described previously.

14.6.3. int SM_DeactivateButton(ButtonType *b)

Makes the button ignore user events. The button face is 'grayed' to signal that it will not respond to user actions. If the button is already deactivated then nothing happens.

14.6.4. int SM_DestroyButton(ButtonType *b)

Remove the button from its parent window and free all memory associated with it. Any user memory associated with the button's *data* field is not automatically recovered -- the application is responsible for that. If the button's parent window is the focus window then the window is redrawn to remove the button from the display.

14.6.5. int SM_GetButtonActive(ButtonType *b) *(macro)*

Returns the status of the button's *active* field. This is True if the button is active, or False if not.

14.6.6. ColorType *SM_GetButtonColor(ButtonType *b) *(macro)*

Returns the button face color. This is a pointer to the button field itself, not a copy. Any changes to the data pointed to by the return value of this function changes the button color field.

14.6.7. void *SM_GetButtonData(ButtonType *b) *(macro)*

Returns the *data* field of the button.

14.6.8. int SM_GetButtonIsdefault(ButtonType *b) *(macro)*

Returns True if the button is a *default* button, False otherwise.

14.6.9. char *SM_GetButtonLabel(ButtonType *b) *(macro)*

Returns the *label* field of the button. This is not a copy of the label but the *label* pointer itself, so modifications to the string returned will modify the button label. Changes will not be apparent until the button is redrawn, however.

14.6.10. ColorType *SM_GetButtonLabelcolor(ButtonType *b) *(macro)*

Returns the color that the button's label is drawn in.

14.6.11. ColorType *SM_GetButtonPcolor(ButtonType *b) *(macro)*

Returns the face color for the button when it's 'pushed'.

14.6.12. PixmapType *SM_GetButtonPixmap(ButtonType *b) *(macro)*

Returns a pointer to the pixmap field of the button, or NULL if there is no pixmap attached to the button.

14.6.13. ColorType *SM_GetButtonPlabelcolor(ButtonType *b) *(macro)*

Returns the label color for the button when it's 'pushed'.

14.6.14. int SM_SetButtonActive(ButtonType *b, int a)

Sets the button's *active* field to *a*, which must be either True or False. Returns *a*.

14.6.15. int SM_SetButtonAlign(ButtonType *b, int alignment)

Sets the horizontal alignment for the button's text element. Like labels, a button's text can be either left or right aligned, or centered in the button. *alignment* must be one of the constants ALIGNLEFT, ALIGNRIGHT, or ALIGNCENTER.

14.6.16. int SM_SetButtonColor(ButtonType *b, ColorType *color)

Sets the button face color to the new value.

14.6.17. int SM_SetButtonData(ButtonType *b, void *data) *(macro)*

Sets the button's *data* field to the new value. *data* is usually a pointer to a user-defined data structure. The memory referenced by *data* must be static.

14.6.18. void SM_SetButtonGraphproc(ButtonType *b, int (*proc)()) *(macro)*

Sets the button's graphics procedure to *proc*. If *proc* is NULL then the button will revert to a standard text button.

14.6.19. int SM_SetButtonIsdefault(ButtonType *b, int d)

Sets the value of the button's *isdefault* field to *d*, which must be either True (non-zero) or False (zero).

14.6.20. int SM_SetButtonLabel(ButtonType *b, char *label)

Changes or sets the button's *label* field to the specified string. If the button already has a label then it is freed and its memory reclaimed before setting the new string. The button makes a copy of the string for itself. If the button's parent window is the focus window then the button is redrawn with the new label.

14.6.21. ColorType *SM_SetButtonLabelcolor(ButtonType *b, ColorType *c)

Sets the color that will be used to draw the button label when the button is not pushed. *c* may be either a system or window color. If the button is part of the current active window, the button is redrawn to reflect the new color.

14.6.22. void SM_SetButtonPixmap(ButtonType *b, PixmapType *pmap, int x, int y)

Attaches the pixmap *pmap* to the button's pixmap field. (*x,y*) is the location within the button where the top-left corner of the pixmap will be. (0,0) is the upper-left corner of the button. If there is already a pixmap attached to the button it's lost. If *pmap* is NULL then any existing pixmap is discarded, so be sure to get and free a button's pixmap before clearing the button pixmap field:

```
pmap = SM_GetButtonPixmap(button);
SM_SetButtonPixmap(button, NULL, 0, 0);
SM_DestroyPixmap(pmap);
```

14.6.23. int SM_SetButtonProc(ButtonType *b, int (*action)()) *(macro)*

Sets the button's *action()* field (the callback function) to the new function.

14.6.24. ColorType *SM_SetButtonPushedColor(ButtonType *b, ColorType *c)

Sets the color that will be used to draw the button background when the button is pushed in. *c* may be either a system or window color.

14.6.25. ColorType *SM_SetButtonPushedLabelColor(ButtonType *b, ColorType *c)

Sets the color that will be used to draw the button label when the button is pushed in. *c* may be either a system or window color.

14.6.26. int SM_SetButtonRect(ButtonType *b, RectType *r)

Changes the button rectangle, essentially changing the size and/or position of the button. All other button parameters are unchanged. If the button is in the focus window then it's redrawn immediately at the new size/position.

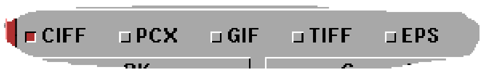
void SM_SetButtonRepeat(ButtonType *b, int flag) (*macro)*

Sets the repeat field for the button to *flag*. The value is interpreted as either True (non-zero) or false (0).

14.7. Examples

The following example programs in the TWS distribution are relevant examples of using buttons:
WINTEST6.C includes an example of a default button;
WINTESTH.C shows placing a pixmap in a button along with text, and varies the button label alignment;
CALC.C demonstrates virtual coordinates for buttons and button callback functions;
ICONEDIT.C also uses pixmaps in buttons, modifies button face colors, and has a number of button callback examples.

15. Checkbox



A checkbox is a toggle-type gadget; it can be either on or off. It can have one of several appearances, including beveled square or circle, or a “button”-look. If the checkbox is on then the checkbox is ‘depressed’ into the screen and its face color is the focus window titlebar color. If off then the face color is the workspace color and the checkbox is ‘raised’ from the screen. Each time the user clicks on a checkbox its state toggles.
When the checkbox is created the application provides a rectangle to enclose the checkbox label and decoration. The rectangle is the sensitive region for the checkbox and the checkbox and its label are clipped within it. A checkbox is toggled by clicking the left mouse button anywhere in the sensitive region; typically, either on the checkbox graphic or on the label.
Checkboxes can be grouped so that only one box of the group can be active at a time; see the section on Checkbox Groups that follows.

15.1. Data Types

```
typedef struct _cb {
    GadgetType  *gadget;                /* Superclass gadget                */
    char        *label;
    int         type;                   /* Visual type                      */
    int         buttonpos;              /* Position for label               */
    int         selected;               /* TRUE if button checked           */
    void        *data;                  /* User-specified data for button   */
    void        *group;                 /* Checkbox grouping, if any        */
} CheckboxType;
```

The *type* field identifies the visual type the checkbox will use. The default visual type is a square decoration next to the label. Alternative styles are defined in smchkbx.h:

Visual type constant	***Value***	***Definition***
SMCBOXSQUARE	0	This is the default visual style as illustrated above.
SMCBOXBUTTON	1	The checkbox’s appearance is the same as the button, but stays pressed in when the checkbox is active.
SMCBOXOVAL	2	As illustrated above, except the visual next to the label is a raised or depressed circle rather than square.

The system constant SMDEFAULT can be used to select the default visual style.
The *buttonpos* field determines where in the checkbox boundary the checkbox graphic will be drawn. If the position is ALIGNLEFT the checkbox graphic will be drawn at the left edge of the boundary with the label to the right of it. If ALIGNRIGHT the checkbox graphic is drawn on the right edge of the boundary and the label is drawn to the left of it. If the visual type is SMCBOXBUTTON, the label is aligned either left or right in the button outline. Note that ALIGNCENTER is valid only for checkboxes using the SMCBOXBUTTON visual style.
If *selected* is True then the checkbox is ‘checked’, otherwise the checkbox is not.
group is a pointer to the checkbox group the checkbox belongs to, or NULL if the checkbox doesn’t belong to a group. See the section of Checkbox Groups that follows.

15.2. Checkbox Callback Procedure

The checkbox callback is an int function whose argument is a pointer to a CheckboxType. The checkbox user function is called each time the checkbox is selected, *after* the system has toggled the *selected* state and *before* the system has redrawn the checkbox graphic to reflect the new state. This allows the user function to modify the state, if desired, before the checkbox is redrawn.

```
...
int CheckboxCallback(CheckboxType *chkbox)
{
        if (SomeOtherProgramCondition) {
                SM_SetCheckboxState(chkbox, True);
        }
        UpdateProgramStatus(SM_GetCheckboxState(chkbox));
}
```

In the example, the checkbox callback tests some other program condition to see if the checkbox's state should be allowed to go False. Then the state of the checkbox is stored by the application.

15.3. Interface Functions *smchkbx.h*

15.3.1. CheckboxType *SM_CreateCheckbox(WindowType *w, RectType *bound, char *label, int buttonpos, int state, void *data, int (*f)())

Create a new checkbox and return a pointer to it.

w	parent window for the checkbox;
bound	sensitive region boundary for the checkbox in window-relative coordinates.
buttonpos .	either ALIGNLEFT or ALIGNRIGHT (defined in smtypes.h), determines whether the checkbox will be to the left of the bounding box or to the right of it;
state	the checkbox's initial state. Should be either True (for initially on) or False (initially off);
data	a pointer to user data;
f	the user function called whenever the checkbox is selected.

15.3.2. int SM_DestroyCheckbox(CheckboxType *c)

Removes the checkbox *c* from the control list of its parent window and frees memory allocated to it.

15.3.3. void *SM_GetCheckboxData(CheckboxType *c) *(macro)*

Returns the *data* field of the checkbox *c.*

15.3.4. char *SM_GetCheckboxLabel(CheckboxType *c) *(macro)*

Returns the character string *label* from the checkbox *c.*

15.3.5. int SM_GetCheckboxState(CheckboxType *c) *(macro)*

Returns the selected state from the checkbox c. This is True if the checkbox is on, False otherwise.

15.3.6. int SM_GetCheckboxType(CheckboxType *c) *(macro)*

Returns the value for the checkbox visual type.

15.3.7. int SM_SetCheckboxData(CheckboxType *c, void *d) *(macro)*

Sets the *data* field of the checkbox *c* to the pointer *d*. The original field value is discarded.

15.3.8. int SM_SetCheckboxLabel(CheckboxType *c, char *s)

Sets the *label* field of the checkbox *c* to the string *s*. The string is duplicated into the checkbox. The previous label, if any, is discarded. If the checkbox is in the focus window then the checkbox is immediately redrawn to display the new label.

15.3.9. int SM_SetCheckboxProc(CheckboxType *c, int (*f)()) *(macro)*

Sets the user procedure called when the checkbox is selected to the function *f*. *f* must be an int function whose argument is a pointer to a CheckboxType.

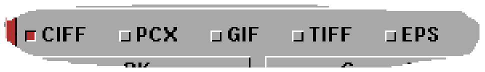
15.3.10. int SM_SetCheckboxState(CheckboxType *c, int state)

Sets the checkbox *selected* field to *state. selected* can be either True or False. If the checkbox is in the focus window then the checkbox is immediately redrawn to reflect the new state.

15.3.11. void SM_SetCheckboxType(CheckboxType *c, int type)

Sets the checkbox visual type to *type*. Valid constants for the type are defined above. If the checkbox is in the focus window then the checkbox is redrawn to reflect its new visual.

16. Checkbox Group



A CheckboxGroup is a collection of checkboxes. The group can have any number of individual checkbox gadgets, but only one of them can be active at any time.
Checkbox groups can be bordered or not, and titled or not. If the checkbox group variable *outline* is True then the checkbox group boundary and title, if any, is drawn; otherwise neither are drawn. If the checkbox group title is NULL then no title is drawn in either case. There is no way to draw a title for a checkbox group without also drawing the border.
All checkbox operations from the previous section can be applied to the individual checkboxes in a group. This section only describes data and functionality specific to groups.
Note: This type of control is sometimes called “radio buttons.”

16.1. Data Types

```
typedef struct _group {
    GadgetType      *gadget;            /* Superclass                    */
    CheckboxType    *selected;          /* Active checkbox               */
    int             selfadjust;         /* If true group boundary autoadjust*/
    int             outline;            /* If true group bounds is drawn    */
    char            *title;             /* Group title                   */
} CheckboxGroupType;
```

The *selected* field points to the individual checkbox that is selected; all other checkboxes in the group are off. If *selected* is NULL then no checkboxes in the group are on.
selfadjust, if True, causes the checkbox group boundary to automatically adjust whenever a new checkbox is added to the group. This is the boundary that is drawn if the *outline* field is true. The CheckboxGroup gadget calculates the boundary as the smallest axis-aligned rectangle (a rectangular parallelepiped, to be purely technical about it) that encloses all the checkbox boundaries in the group. Four pixels are added all around the boundary. The boundary can only get bigger.

If *outline* is True then the checkbox group boundary and title, if any, are drawn around the group of checkboxes. Note that no tests are made that the boundary rectangle encloses checkboxes *only*. Particularly for *selfadjust*ed boundaries, the boundary rectangle may pass through other window gadgets unless the individual checkboxes are positioned carefully.

16.2. CheckboxGroup Callback Procedure

There is no callback procedure for the checkbox group. The individual checkbox callback functions *are* executed when checkboxes are selected. When a user selects a new checkbox, the previous checkbox selection state is toggled off and it's callback function is called. Then the new checkbox is toggled on, then its callback function is called.
The value returned by the second checkbox callback function (the one activated) is the one returned to the application via the event processor.

16.3. Interface Functions ***smchkbx.h***

16.3.1. CheckboxType *SM_AddCheckbox(CheckboxGroupType *g, RectType *box, char *label, int buttonpos, int selected, void *data, int (*f)())

Creates a new checkbox, adds it to the end of the checkbox group list *g*, and returns a pointer to the individual checkbox created. The parent window for the checkbox is taken from the checkbox group.

g...............	Checkbox group to attach new checkbox to.
box............	Boundary for the individual checkbox in window coordinates.
label	Character string label for the new checkbox.
buttonpos ..	ALIGNLEFT or ALIGNRIGHT, for determining position within the *box* of the checkbox button.
selected	True if the new checkbox is the one that's on, False otherwise. If more than one checkbox is added to the group as on, the latest one has precedence.
data	User data.
f................	User function called whenever the individual checkbox is selected.

16.3.2. CheckboxGroupType *SM_CreateCheckboxGroup(WindowType *w, RectType *r, char *label, int draw, int (*f)())

Creates a new checkbox group and returns a pointer to it. The checkbox group and all checkboxes in the group will be attached to window *w*.

w	Parent window for the checkbox group and all checkboxes in the group;
r................	Boundary rectangle for the group. If the *Xmax* or *Ymax* field of the rectangle is 0 then the checkbox group will set the *selfadjust* field and the boundary will be automatically adjusted to enclose all checkboxes added to the group. In any case the *Xmin* and *Ymin* fields are interpreted as the upper-left corner of the group bound.
label	Text label for the group. If the *draw* field is True and the *label* is not NULL, the text label is drawn using the system font in the top middle of the group boundary rectangle.
draw..........	If True, the group boundary and label will be drawn;
f................	Checkbox group callback procedure. Not currently used. Set to NULL for future compatibility.

16.3.3. int SM_DestroyCheckboxGroup(CheckboxGroupType *g)

Destroys the checkbox group *g* and all checkboxes grouped with *g.*

16.3.4. CheckboxType *SM_GetCheckboxSelected(CheckboxGroupType *g)

Returns a pointer to the checkbox of the group *g* that is on.

16.3.5. int SM_RemoveCheckbox(CheckboxType *cb)

Removes the checkbox *cb* from its group, if any.

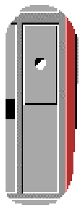
16.3.6. int SM_SetCheckboxSelected(CheckboxGroupType *g, CheckboxType *c)

Makes the checkbox *c* the selected checkbox of group *g*. The checkbox must be in the group; if it isn’t, the *selected* checkbox from group *g* is unchanged and False is returned.

16.3.7. int SM_UngroupCheckboxGroup(CheckboxGroupType *g)

Destroys the checkbox group *g* without destroying the individual checkboxes associated with *g*. The individual checkboxes are ungrouped but not otherwise disturbed.

17. Slider



A slider is a control for selecting within a fixed range of values. Visually the control portion of the slider rests inside a fixed-length ‘channel.’ The control’s position within the channel indicates the value of the control. The length of the slider channel and the values at the minimum and maximum positions are all defined by the application. The length and position of the slider thumbbar is calculated by the system based on application-supplied parameters. A slider can be either vertical, as illustrated, or horizontal.
The size of the thumbbar relative to the size of the slider channel is automatically adjusted to show the ratio of the slider’s range to the range within a ‘page,’ described below. The user controls a slider by either clicking the left mouse button on either end of the thumb bar, or by dragging the thumb bar control in its channel. If all the slider’s range is in the page then the thumbbar is the same size as the channel and no adjustment is possible.

17.1. Data Types

```
typedef struct _sli {
    GadgetType *gadget;                 /* Superclass gadget                */
    RectType thumbbar;                  /* Slide thumbbar region            */
    int     type;                       /* Horizontal or Vertical           */
    int     min;                        /* Slider value at minimum          */
    int     max;                        /* Slider value at maximum          */
    int     scale;                      /* Size of jump in user units       */
    int     position;                   /* min <= position <= max           */
    int     twidth;                     /* Width in pixels of thumbbar      */
    int     pixels;                     /* Number of pixels free movement   */
    int     state;
    double  inc;                        /* Units/pixel for thumbbar moves   */
    void    *data;                      /* User-specified data for button   */
} SliderType;
```

The *channel* rectangle is supplied by the application, but the *thumbbar* rectangle is calculated by the slider based on the min, max, and scale parameters. The width of the *thumbbar* is designed to provide a visual clue of the ratio of the range of values controlled by the slider (*max-min*) to the number of values in a single ‘page’ (*scale*). *position* is the data value for the current slider position. *scale* is the number of user units in a single ‘page.’ *twidth* is the free dimension of the thumb bar, and *pixels* is the number of pixels in the channel not covered by the thumb bar. These values are calculated internally.

17.2. Slider Callback Function

The slider’s application callback function is called whenever the *position* field of the slider changes. The function is an int function whose argument is a pointer to a SliderType.
Dragging the slider’s thumbbar may or may not result in a call to the function, since the *inc* value may dictate that it takes a move over multiple pixels to effect a single unit change in the slider. In most cases however dragging the slider thumbbar results in a torrent of calls to the user function. If the user function doesn’t process these calls very quickly then the slider will seem to

behave sluggishly. For example, using a slider to scroll a large image would take far too long if done on a pixel-by-pixel basis.
The slider's *state* field is used to restrict processing to particular slider actions. If the slider thumbbar is being dragged by the user, the state's value is SLIDERTHUMBSTATE. If the slider has just been 'jumped' down the state's value is SLIDERJUMPDOWNSTATE and so forth. These constants are defined in smslider.h.

State constant	***Value***	***Definition***
SMSLIDERTHUMBSTATE	1	The user is dragging the slider thumbbar. The slider's position value has been updated but may change again very quickly. If processing a change in position is very time consuming (e.g., redrawing a large graphic), the callback function will probably return immediately if this is the slider's state. Processing that can be done very quickly, like modifying a string in a label, could be handled.
SLIDERTHUMBDONESTATE	4	The user has finished dragging the slider thumbbar. The slider is finished processing its event.
SLIDERJUMPUPSTATE	2	The user clicked the slider channel above the thumbbar position. The slider position has been updated and the slider thumbbar redrawn.
SLIDERJUMPDOWNSTATE	3	The user clicked the slider channel below the thumbbar position. The slider position has been updated and the slider thumbbar redrawn.

17.3. Interface Functions *smslider.h*

17.3.1. SliderType *SM_CreateSlider(WindowType *w, int type, RectType *box, int min, int max, int size, int initpos, void *data, int (*f)())

Create a new slider and return a pointer to it.
The slider is somewhat more complex to set up than other gadgets. In particular, the meanings of the *min*, *max*, and *size* arguments bear some explanation.
The *min* argument is the value that is to be the slider's *position* value (see the Data Type section above) when the slider thumbbar is at its leftmost position (HORIZONTAL slider) or topmost position (VERTICAL slider). The *max* value is the slider's *position* value at the other end.
However, the slider will also work properly if the *min* value is greater than the *max* value. For example, potentiometer-type sliders usually have their minimum value at the bottom; you slide the pot up to increase the value. For this type of control set *min* to the potentiometer's maximum value, and *max* to the potentiometer's minimum value.
The *size* argument is the number of application data items in a 'page.' This is an idea from window scrolling. For example, if you're using a vertical slider to scroll some text (see the Text gadget, for example), you'd set *size* to the number of lines that fit in your window (and *min* would be 0 and *max* would be the total number of lines in your text buffer). When the user clicks in the slider channel above or below the thumbbar, the *position* field is adjusted by *size* either up or down. For the previous potentiometer example, *size* would be set to 1. It must *not* be less than 1.

w is the window the slider is attached to. *type* is the type of slider, either HORIZONTAL or VERTICAL (constants defined in smslider.h). The *box* is the rectangle for the slider's channel in window coordinates. *initpos* is the initial slider position value. *data* is a pointer to user data. *f* is the application function called whenever the slider's value is changed. If the slider is being dragged by the user then the function is called repeatedly for each value change as the thumb bar is moved. The function must be an int function whose single argument is a pointer to a SliderType.

17.3.2. void SM_DestroySlider(SliderType *s)

Delete the slider *s* by removing it from its parent window's control list and recover memory used by it The SliderType pointer *s* is the pointer returned by the **SM_CreateSlider** function.

17.3.3. void *SM_GetSliderData(SliderType *s)

Returns the *data* field for the slider.

17.3.4. int SM_GetSliderMax(SliderType *s)

Returns the value of the *max* field of the slider, which is the value of the slider when the thumbbar position is at the very bottom|right of the slider channel.

17.3.5. int SM_GetSliderMin(SliderType *s)

Returns the value of the *min* field of the slider, which is the value of the slider when the thumbbar position is at the very top|left of the slider channel.

17.3.6. int SM_GetSliderPosition(SliderType *s)

Returns the value of the *position* field of slider *s.*

17.3.7. int SM_SetSliderChannel(SliderType *s, RectType *r)

Changes the slider's channel to the rectangle *r*, in coordinates relative to *s*'s parent window. If the slider's parent window has the focus then the slider is redrawn.

17.3.8. int SM_SetSliderData(SliderType *s, void *d)

Sets the *data* field of the slider *s* to the pointer *d*. Usually gadget data are pointers to user-defined data structures. The original value of the *data* field is discarded.

17.3.9. int SM_SetSliderMinMax(SliderType *s, int min, int max)

Sets the *min* field of the slider *s* to *min*, and the *max* field to *max.* The previous values are discarded. If the slider *s* is in the focus window then the slider thumb bar is recalculated and the slider is redrawn.

17.3.10. int SM_SetSliderPosition(SliderType *s, int p)

Sets the *position* field of the slider *s* to the value *p.* The previous value is discarded. If the slider *s* is in the focus window then the slider thumb bar position is recalculated and the slider is redrawn.

17.3.11. int SM_SetSliderScale(SliderType *s, int scale)

Sets the slider's *scale* field. If the parent window of *s* is the focus window then the slider is redrawn with a recalculated thumbbar. Returns *s->scale*.

17.4. Example

The simplest use of a slider is as a value selector within a fixed range of values. The user moves the slider thumbbar to the desired position and the slider callback function sets the data value accordingly. This type of operation is often combined with a label to display the data value in the slider.

When using a slider this way, remember there's a relationship between the size of the slider channel and the values the slider can choose. In particular, if there are more data values between *min* and *max* than pixels in the channel, the slider will skip over some values in the range. The slider is therefore not suitable for precision selection within a large range of values.

For this example we'll assume the slider is to select a value between 0 and 100, inclusive. We'll set the slider channel comfortably large so that every individual value can be selected, and we'll create a label over the slider to display the current slider value.

```
WindowType     *w;
RectType       rt;
SliderType     *v;
LabelType      *label;

rt.Xmin = rt.Ymin = 4;
rt.Xmax = rt.Xmin + 50;
rt.Ymax = rt.Ymin + SM_GetSystemFontHeight() + 6;
label = SM_CreateLabel(w,
                       &rt,
                       "",
                       NULL,
                       ALIGNLEFT,
                       False,
                       False,
                       True,
                       NULL);
rt.Ymin = rt.Ymax + 4;
rt.Ymax = rt.Ymin + 150;              /* Leave plenty of room for 100 'items'      */
rt.Xmax = rt.Xmin + 10;
v = SM_CreateSlider(w,
                    &rt,
                    VERTICAL,
                    100,              /* 'Minimum' slider value                    */
                    0,       /* 'Maximum' slider value                    */
                    1,       /* 'page' size (pixels per item)             */
                    0,       /* Initial item                              */
                    (void *)label,/* User data                                 */
                    DoSlider);        /* Callback function                         */
```

Notice that the slider 'minimum' value is larger than the 'maximum' value. The minimum and maximum application values are related to the minimum and maximum pixel values in the slider (the *y* values in the case of a vertical slider, or *x* values in the case of a horizontal slider). In other words, the slider normally puts the minimum value at the top of a vertical slider, and values increase going down. Most people are more comfortable with increasing values going up, so we simply reverse the values. The slider can deal with it either way.

There is no 'page' of items corresponding to the slider display, so we set it to its minimum value of 1. We also attach the label to the data field of the slider so we can access it from the slider itself.

Next write the slider callback function:

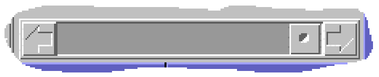
```
int DoSlider(SliderType *slider)
{
       LabelType      *label;
       char           valstr[5];

       label = (LabelType *)SM_GetSliderData(slider);
       sprintf(valstr,"%d",SM_GetSliderPosition(slider));
       SM_SetLabelString(label, valstr);
```

```
        return 1;
}
```

All it does is extract the label from the slider's data field, create a string based on the slider's position value, and update the label. As the user slides the thumbbar up and down the value in the label will change in real-time. In a real application you would probably also capture the slider position value and store it in an application variable.

18. Scrollbar



The scrollbar is similar to the slider gadget. Besides the slider channel, however, the scrollbar also has buttons at the ends of the channel. Pressing one of the buttons changes the scrollbar position by one unit, whereas clicking inside the channel on either end of the thumbbar changes the position by one page. A scrollbar can be either vertical or horizontal.
Like the slider, the size of the thumbbar relative to the size of the scrollbar channel is automatically adjusted to show the ratio of the scrollbar's range to the range within a 'page.'
The user controls a scrollbar by either clicking the left mouse button on either end of the thumbbar, by dragging the thumbbar control in its channel, or by clicking one of the buttons. If all the slider's range is in the page then the thumbbar is the same size as the channel and no adjustment is possible.
Clicking on one of the scrollbar unit buttons changes the *position* value by one unit either up or down. If the position is already at the min or max position then the *position* value is unchanged. If the *position* value doesn't change then the scrollbar's callback function is not called.

18.1. Data Types

```
typedef struct _sb {
    GadgetType  *gadget;
    void        *data;
    int         type;
    SliderType  *slider;
    ButtonType  *upbutton;
    ButtonType  *downbutton;
} ScrollbarType;
```

The Scrollbar takes advantage of the previously defined slider and button gadget types for most of its functionality. The *type* field identifies the scrollbar as horizontal or vertical, and the *data* field contains application data.
The application can't directly manage the scrollbar's slider or buttons; the scrollbar does that internally.

18.2. Scrollbar Callback Function

The callback function supplied when the scrollbar is created behaves just about the same as the slider callback function. In fact, the Scrollbar callback function is an int function whose argument is a pointer to a *slider* rather than a scrollbar. There is an additional possible state value that might be returned if the slider is part of a scrollbar: The SLIDERDONESTATE (value 0) is returned if the user clicked on one of the slider buttons. This state is never returned by sliders.

18.3. Interface Functions *smscroll.h*

18.3.1. ScrollbarType *SM_CreateScrollbar(WindowType *w, RectType *r, int type, int min, int max, int size, int init, void *data, int (*f)())

Creates and initializes a new Scrollbar and returns a pointer to it.

min ....The value for the scrollbar's *position* field when the scrollbar's slider thumbbar is at its leftmost or topmost position.

max...........	The value for the scrollbar's *position* field when the scrollbar's slider thumbbar is at its rightmost or bottom position. The actual value for *max* can be either greater or lesser than *min*.
type...........	Orientation for the scrollbar, either HORIZONTAL or VERTICAL (defined in smslider.h);
size	Number of application data items per 'page'. This value is used to adjust the slider thumbbar size. For example, if the scrollbar is used to control a window that contains some text, the *size* field would be set to the number of lines that fit in the text display. This would enable the slider portion of the scrollbar to drag the text so that the last text line was at the bottom of the 'page'. *size* must be at least 1;
r................	The rectangle enclosing the scrollbar. The slider and buttons are set inside this rectangle;
w	Window the scrollbar is attached to;
f................	The scrollbar's callback function. The function is called whenever the scrollbar's *position* value changes, either through the slider or one of the buttons. Clicking on the scrollbar does *not* call the function if the position value doesn't change.

18.3.2. void *SM_GetScrollbarData(ScrollbarType *sb)

Returns the application data field for the scrollbar.

18.3.3. int SM_GetScrollbarMax(ScrollbarType *sb)

Returns the *max* value the scrollbar can have.

18.3.4. int SM_GetScrollbarMin(ScrollbarType *sb)

Returns the *min* value the scrollbar can have.

18.3.5. int SM_GetScrollbarPosition(ScrollbarType *sb)

Returns the current scrollbar value.

18.3.6. int SM_GetScrollbarScale(ScrollbarType *sb)

Returns the page size, called the *scale* value internally, for the scrollbar *sb*. This is the number of data items that constitute a page "jump" when the user clicks above or below the scrollbar thumbbar. It has the same meaning as the corresponding slider field.

18.3.7. void SM_SetScrollbarChannel(ScrollbarType *sb, RectType *r)

Changes the size of the scrollbar based on the rectangle *r*. The slider channel, thumbbar, and buttons are resized accordingly.

18.3.8. void SM_SetScrollbarData(ScrollbarType *sb, void *data)

Sets the application data field for the scrollbar. The existing data value, if any, is discarded.

18.3.9. void SM_SetScrollbarMinMax(ScrollbarType *sb, int min, int max)

Sets the scrollbar's *min* and *max* fields. Automatically adjusts the thumbbar size if necessary to reflect the new range. Note that the *position* field is *not* affected.

18.3.10. void SM_SetScrollbarPosition(ScrollbarType *sb, int pos)

Sets the scrollbar's current value. The scrollbar's slider thumbbar is moved accordingly. *pos* should be in the range of the scrollbar, between *min* and *max.*

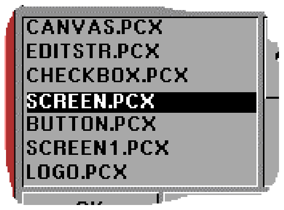
18.3.11. SM_SetScrollbarProc(ScrollbarType *sb, int (*f)(SliderType *))

Sets the application callback function for the scrollbar *sb*. The function *f* is an int function whose argument is a pointer to a SliderType, which will be the slider component of the scrollbar.

18.3.12. void SM_SetScrollbarScale(ScrollbarType *sb, int scale)

Changes the ‘page’ size for the scrollbar. Must be greater than 0. Automatically adjusts the scroll-bar’s slider’s thumbbar size as necessary.

19. Stringlist



A stringlist displays a list of strings supplied by the application and allows the user to select one. A string is selected by positioning the cursor over it and clicking the left mouse button. If another string is already selected, it is unselected in favor of the new string.
If the list of strings doesn't fit within the stringlist bounds, the list can optionally be made scrollable. Scrollable single-column stringlists have a vertical slider to the right of the list. Multiple-column stringlists have a vertical slider to the right and a horizontal slider below the list. Moving the slider positions changes the portion of the stringlist displayed, but doesn't change the selection. If the stringlist is the focus gadget it also responds to keyboard input as described below.
The currently installed system font and font color is used for the string display. If the list is scrollable or has multiple columns, the attached sliders are drawn inside the stringlist border.

19.1. Data Types

```
typedef struct _sl {
    GadgetType  *gadget;              /* 'Superclass' gadget              */
    int         nstart;               /* first string to display          */
    int         ndisplay;             /* Number of strings to display     */
    int         nselected;            /* Index of selected string         */
    int         nitems;               /* Total number of list items       */
    char        **list;               /* String list                      */
    int         destroyflag;          /* If true gadget can free stringlist*/
    int         scrollflag;           /* If true gadget can free stringlist*/
    int         allowmulti;           /* If true then allow multiple selection */
    int         ncolumns;             /* Number of columns in list        */
    char        **cols[10];           /* Pointers to column heads         */
    int         firstcol;             /* Leftmost column index            */
    int         columnsep;            /* Pixel separation between cols    */
    int         colfit;               /* Number of displayable columns    */
    SliderType  *hscrollbar;          /* Stringlist scroll, if requested  */
    SliderType  *vscrollbar;          /* Stringlist scroll, if requested  */
    void        *data;                /* User data pointer                */
} StringlistType;
```

Field	Description
nstart	The first string in the list to display in the stringlist rectangle, always displayed at the top. The application supplies the initial value for this when the stringlist is created. This is an index into the list, with 0 the first string. If the list is scrollable, the user will be able to scroll back to the beginning of the list regardless of the initial value. This value is updated as the user scrolls the list.
ndisplay	Number of strings that fit in the boundary rectangle. This value is calculated when the stringlist is created.
nselected	The index of the string currently selected. When the stringlist is first created, this value is -1, meaning none of the strings are selected.

nitems	Number of strings in the list. The application supplies this value when the stringlist is created. The stringlist gadget always assumes this value is correct, so if there are actually fewer strings than this the stringlist will display garbage. If there are more strings then they won't be displayed.
list	The list of strings. The stringlist does *not* copy the list, but simply copies the pointer supplied by the application. Thus the list must be static data relative to the lifetime of the stringlist gadget. Changes in the list will be reflected in the stringlist gadget display (but not until the next time the gadget is redrawn).
destroyflag	Supplied by the application when the stringlist is created, if True then TWS will free the list when the stringlist gadget is destroyed. If False then the stringlist destructor will ignore the list and it'll be up to the application to free it, if necessary.
scrollflag	If True then a vertical slider will be attached to the stringlist. If the list is longer than the space allowed in the boundary rectangle, the user will be able to scroll through the list. If the entire list fits in the boundary then the slider is drawn anyway, but does nothing.
allowmulti	If True, the stringlist will allow multiple strings to be selected. If false, the stringlist can return only a single string.
ncolumns	The number of columns to display the list in. Supplied by the application. The gadget will divide the list as necessary (this does not affect the data in the list itself). If ncolumns is greater than 1 the stringlist adds a horizontal scrollbar at the bottom of the stringlist rectangle. If all the columns will not fit in the stringlist boundary then the user will be able to scroll the columns across. If all the columns fit then the horizontal scrollbar will still be drawn but does nothing.
cols	Pointers to beginning of columns. Maintained internally.
firstcol	Leftmost column in the stringlist boundary rectangle. Maintained internally.
columnsep	Number of pixels separating columns. Arbitrarily set to 10 pixels.
colfit	Number of columns that fit in the boundary rectangle. Only calculated if ncolumns is greater than 1. Maintained internally.
hscrollbar vscrollbar	Stringlist sliders. Maintained internally.
data	Application data.

19.2. Stringlist Callback Function

The stringlist user function is called whenever a string from the list is selected. It is an int function whose argument is a pointer to a StringlistType. The function is called after the string selected by the user has been recorded and highlighted by the system.

19.3. Keyboard Interface

If a stringlist is the focus gadget, it responds to keyboard input as well as mouse input. The stringlist responds to the following keys:

Key	***Action***
Up / Down arrow	Move the string selection up or down by one. If the selection is already at the top or bottom of the list, nothing happens. The list scrolls as necessary and, if sliders are attached to the list, the slider positions are updated as well.

PgUp / PgDn	Jump the selected string up or down by a single “page.” Stops at the top and bottom of the list. If the list has sliders attached, they are updated as necessary.
Left / Right arrow	For multiple-column lists, jumps the selected string over by one column. No effect for single-column lists.
Home / End	Positions the selection at the first or last item of the list, as appropriate. Scrolls the list if necessary and adjusts the slider(s) if appropriate.

A stringlist focus gadget can always be scrolled with the keyboard interface, though not using mouse input unless the stringlist was originally created as a scrollable stringlist. When a key is pressed in the focus stringlist, if the selected string isn’t visible, the list is first scrolled to bring the selected string into the list box.
After each recognized keystroke, the stringlist’s callback function is called.

19.4. Selecting Multiple Strings

If the *allowmulti* field is set then the stringlist gadget will keep track of more than one selected string. Whether or not a stringlist gadget will allow multiple selections is determined when the stringlist is created.
Multiple selection introduces subtle changes in the stringlist’s operation. In a single-selection stringlist, the highlighted string is equivalent to the selected string. Moving the highlight, either through mouse or keyboard control, also moves the selection. For multiple selection, the highlighted string is not necessarily one of the selected strings.
The highlighted string for a multiple selection stringlist is represented by an outline instead of a reverse-video box. Selected strings are represented in reverse video. Also, the following mouse and keyboard actions are added or modified:

Key	***Single-selection action***	***Multiple-selection action***
Left mouse button	Set the selection and the highlight	Set the selection and the highlight. Any other existing selections are cleared.
Shift + Left mouse button	None	Extends the selections from the previous *highlight* position to the current position. Does not affect any existing selections.
Ctrl + Left mouse button	None	Adds or deletes the string at the cursor position to the selection list. If the string is already on the list, it is removed, otherwise it is added.
Space	None	Same as Left mouse button.
Shift + Space	None	Same as Shift + Left mouse button.
Ctrl + Space	None	Same as Ctrl + Left mouse button.

For multiple selection stringlists, the functions **SM_GetStringlistSelection** and **SM_GetStringlistSelectString** return the index or string value for the *highlighted* string, not one of the selected strings. To retrieve a list of indices for the selected strings, use **SM_GetStringlistSelectionList**. This function returns an array of integer indices into the the stringlist string array. The list is sorted in ascending order. The function **SM_GetStringlistSelectionCount** returns the number of selected strings. The value returned is accurate for either single or multiple selection stringlists.

```
StringlistType *slist;
char           **list;
RectType              box;
```

```
WindowType             *w;
int                    i, count, *selections;
/*
**...
** Set up list for multiple selections
*/
slist = SM_CreateStringlist(w, &box, list, cnt, 0, 1, MyCallback, 0, 0, True, NULL);
/*
** ...
** Done with stringlist, get what the user selected
*/
count = SM_GetStringlistSelectionCount(slist);
selections = SM_GetStringlistSelectionList(slist);
for (i = 0; i < count; i++) {
    /*
    ** Process the strings
    */
    DoSomethingWith(list[selections[i]]);
}
```

19.5. Interface Functions *smstrlst.h*

19.5.1. StringlistType *SM_CreateStringlist(WindowType *w, RectType *r, char **list, int nitems, int nstart, int ncols, int (*f)(), int destroyflag, int scrollflag, int multiflag, void *data)

Create a new stringlist gadget attached to window *w* and return a pointer to it.

w	parent window for the stringlist;
r	the bounding rectangle for the stringlist in window-relative coordinates;
list............	the static array of strings to be displayed;
nitems	number of strings in *list*;
nstart	index (starting at 0) of the string from *list* to be displayed first in the box;
ncols	number of columns to display the list in. If greater than 1 **SM_CreateStringlist** calculates the number of strings for each column and the number of columns that will fit in the boundary rectangle *r* (also taking into account the slider, if any). A horizontal slider is added to the stringlist for scrolling left and right between columns.
f................	application function to be called whenever the stringlist is selected. The function gets a single argument, which will be a pointer to the stringlist.
destroyflag.	If non-zero, the stringlist destructor function will free the strings in the stringlist, then the stringlist itself. This assumes that both the individual strings and the list itself were dynamically allocated. If 0, the stringlist destructor ignores *list*.
scrollflag	If non-zero then the stringlist will have a vertical slider attached to it, which is used to scroll through the list. The slider is drawn to the inside of the stringlist boundary *r* which reduces the space available for displaying strings.
multiflag	If True then the stringlist will allow multiple selections, otherwise only one string at a time can be selected.
data	Application data.

19.5.2. int SM_DestroyStringlist(StringlistType *s)

Free memory allocated to the stringlist argument and remove it from its parent window's control list. Does *not* free memory used by array of strings.

19.5.3. int SM_GetStringlistNDisplay(StringlistType *s)

Returns the number of strings that fit in the stringlist rectangle. This value is calculated when the stringlist is created and is the same as the *ndisplay* field of the stringlist structure.

19.5.4. int SM_GetStringlistSelection(StringlistType *s)

Returns the index of the item selected in stringlist *s*. This value is the *nselected* field of the stringlist structure.

19.5.5. int SM_GetStringlistSelectionCount(StringlistType *s)

Returns the number of strings selected in the stringlist *s*. Works for both single and multiple selection lists. Returns 0 if no strings are selected.

19.5.6. int *SM_GetStringlistSelectionList(StringlistType *s)

Returns an integer array containing the indices of the strings selected from stringlist *s*. Valid only for stringlists with the *allowmulti* field set. The returned array is internally *malloc*-ed and is of size **SM_GetStringlistSelectionCount**. The application should free this array when it's no longer needed.

19.5.7. char *SM_GetStringlistSelectString(StringlistType *s)

Returns a pointer to the string that is currently selected. This is a pointer to the string in the original list (which is in the application). Modifying this string will modify the stringlist display when it's next drawn.

19.5.8. int SM_RedrawStringlist(StringlistType *s)

Unconditionally redraw the stringlist *s*. Does not check that the stringlist's parent window is the focus window.

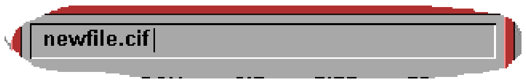
19.5.9. void SM_SetStringlistList(StringlistType *s, char **list, int nitems)

Gives an existing stringlist a new list of strings to display. Recalculates columns and sliders as necessary and, if the stringlist is in the focus window, redraws the stringlist. The selection is cleared (no string selected).

19.5.10. int SM_SetStringlistSelection(StringlistType *s, int n)

Sets the selected string in the stringlist *s* to the n^{th} element in the string array. If *n* is out of range then the existing selection is unchanged and False is returned. On success True is re-turned. The selected-string highlighting is modified to reflect the new selected string if the *s*' parent window is the focus window.

20. Editstring



The Editstring is a one-line, bordered gadget containing a text string which can be edited. When active, it traps keystroke events, including the Enterkey, and left mouse button events. Printable characters are added to the string at the insertion point (the beginning of the string initially). The left and right arrow, end, and home keys, move the edit cursor thus changing the insertion point for input characters. Clicking the mouse cursor inside the editstring box while the editstring is active moves the edit cursor to the mouse cursor position. The delete and backspace keys work as expected.

If the string is longer than the boundary rectangle, the string scrolls left and right in response to the arrow keys, backspace key, and text entry. The home and end keys also scroll the string as necessary to display the beginning and end of the string.

Pressing the Enter key while an editstring is active deactivates it. When the editstring is inactive it only responds to the left mouse button, which activates the editstring if the cursor is inside the editstring boundary. Within a focus window only one editstring should be active at any time; if more than one is, the first one created will process all keystrokes.

Most gadgets trap[11] the input events they're interested in. However, the editstring does *not* trap the Enter key event. This allows the event manager to propagate the Enter key to other gadgets that might be interested in it. The main reason for this is that the Enter key triggers the default button, which is often used to close application windows.

20.1. Data Types

```
typedef struct _etext {
    GadgetType *gadget;               /* Superclass gadget               */
    char    *string;                  /* String to be edited             */
    int     maxlength;                /* Allocated string space          */
    int     cursorpos;                /* Index in string where cursor is */
    int     headpos;                  /* Index where string is drawn     */
    int     boldflg;                  /* TRUE if string is bolded        */
    int     italflg;                  /* TRUE if string is italicized    */
    int     insert;                   /* Insert/overwrite mode flag      */
    int     xalign;                   /* Horizontal string alignment     */
    int     start_highlight;          /* First index of highlight block  */
    int     end_highlight;            /* Last index of highlight block   */
    int     active;                   /* True if this string is active   */
    void    *data;                    /* User data                       */
    int     (*action)();              /* User callback function          */
    int     group;                    /* Editstring group                */
       } EditstringType;
```

Editstring data field	*Description*
string	The initial string for editing passed in when the editstring is created. The editstring makes a copy of the string.

[11]*Trapping* an event simply means that, once a gadget has actively responded to an event (the event was one the gadget was interested in, and occurred within the gadget bounds, and the gadget performed some action in response to the event), the gadget shell function tells the gadget manager to stop checking any other gadgets. This is an efficiency feature – if you click the mouse on a button, you can't have also clicked it on a checkbox. Why waste time testing it?
There are time you want events to 'propagate' through the system, however. This is one of those times.

maxlength	Provided by the application when the editstring is created. This is the maximum number of characters the string can hold. If 0 then the length of the initial *string* is used as maxlength.
cursorpos	Index in the string where the next input character will be inserted. Managed internally.
headpos	String index for the beginning of the display. Managed internally.
backcolor	Background color for the stringlist when active. Set to White internally.
textcolor	Color of the string, set to Black internally.
boldflg	If True the string is bolded. Not all graphics kernel systems support text attributes. Defaults to False.
italflg	If True the string is drawn using an italic facing. Not all graphics kernel system support text attributes. Defaults to False.
insert	Insert/overwrite flag. Only insert mode is currently supported.
xalign	Text alignment within the editstring boundary. Defaults to ALIGNLEFT. Other alignments are not currently supported.
start_highlight end_highlight	Not currently used.
active	True if the editstring is active. Defaults to True when the editstring is created.
data	User data.
action	Callback function.
group	Identifier for a collection of editstrings, which allows only one from the group to be active. Defaults to 0 (no grouping).

20.2. Editstring Callback Procedure

The editstring callback function is an int function with *two* arguments: a pointer to an EditstringType and a pointer to an EventType. The function is called for every event processed by the editstring, except for the event that activates it, and is called *before* the event is processed by the editstring. This gives the callback function the opportunity to modify the event before the editstring gets a look at it. Some simple examples are converting characters to uppercase, adding a keyclick, or filtering illegal characters for validated data input.
The editstring callback function should return 0 if the editstring should continue to process the input, or non-zero if not.

20.3. Editing a string

When the Editstring is created the application specifies either the initial string to be edited, a maximum length of any string in the gadget, or both. If no maximum length is specified (set the *maxstr* argument to 0), then the gadget will limit all strings to the length of the initial string. If *maxstr* is greater than 0, its value sets the maximum number of characters that the string can hold.
In either case, the gadget will not allow more characters than *maxstr*.

20.4. Interface Functions *smeditst.h*

20.4.1. EditstringType *SM_CreateEditstring(WindowType *w, RectType *box, char *str, int maxstr, void *data, int (*f)())

Create a new editstring gadget and return a pointer to it.

w	Parent window for the editstring;
box...........	Boundary rectangle for the editstring in window coordinates. The string is clipped within this box.
str	String to be edited. Must be static data;
maxstr	Maximum length of the string that can be edited. *Must be >= the length of the initial string, if any!* If 0 then the length of *str* is used as the maximum length.
data	User data.
f................	Callback function.

20.4.2. int SM_DestroyEditstring(EditstringType *e)

Free the editstring and all resources allocated to it. If the editstring's parent window has the focus then it's redrawn.

20.4.3. int SM_GetEditstringActive(EditstringType *e)

Returns the *active* field for the editstring, either True or False.

20.4.4. int SM_GetEditstringCursorpos(EditstringType *e)

Returns the edit cursor position for the string in the editstring. The cursor position is the index into the character string being edited, with 0 as the first character.

20.4.5. int SM_GetEditstringGroup(EditstringType *e)

Returns the ID of the group the editstring belongs to. If 0 then the editstring doesn't belong to a group. Within a group only one editstring is allowed to be active -- activating one editstring in a group automatically deactivates any others in the same group.

20.4.6. char *SM_GetEditstringString(EditstringType *e) *(macro)*

Returns the string being edited. The return pointer is string in the Editstring *e* itself. If the string is modified by the application the Editstring gadget won't know it, and Chaos will result.

20.4.7. int SM_SetEditstringActive(EditstringType *e, int a)

Makes the editstring active if *a* is True, or inactive if *a* is False. If the editstring's parent window has the focus then it is redrawn to indicate the change in state. Returns *a*.

20.4.8. int SM_SetEditstringCursorpos(EditstringType *e, int pos)

Sets the edit cursor position within the editstring to the string character at index *pos* within the string, where the first character is 0. The edit cursor is placed in front of this character. If *pos* is greater than the number of characters in the string then the position is set to the end of the string. Returns the actual *cursorpos*; either *pos* or the position at the end of the string.

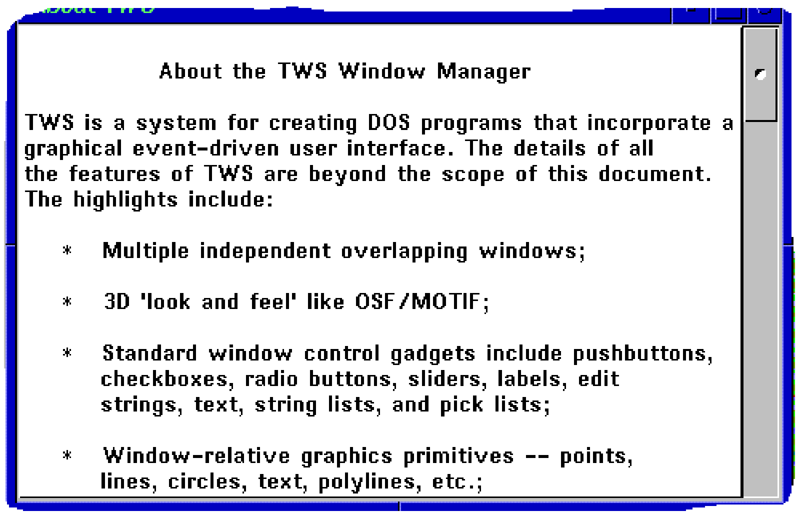
20.4.9. void SM_SetEditstringGroup(EditstringType *e, int group)

Sets the editstring group to *group*, which should be a non-zero integer identifier. All editstrings to be in the same group should be set to the same value. To remove an editstring from a group without deleting the editstring itself, set the group value to 0.
The function doesn't check that only one of the editstrings in the group is active so the application must ensure this.

20.4.10. int SM_SetEditstringString(EditstringType *e, char *s)

Changes the string pointed to by the editstring to *s*. The previous string is discarded. *s* must be a pointer to static data. The cursor position is set to 0 and, if the editstring's parent window has the focus, the editstring is redrawn to reflect the change. Returns the length of *s*.

21. Textbox



About the TWS Window Manager

TWS is a system for creating DOS programs that incorporate a graphical event-driven user interface. The details of all the features of TWS are beyond the scope of this document. The highlights include:

* Multiple independent overlapping windows;

* 3D 'look and feel' like OSF/MOTIF;

* Standard window control gadgets include pushbuttons, checkboxes, radio buttons, sliders, labels, edit strings, text, string lists, and pick lists;

* Window-relative graphics primitives -- points, lines, circles, text, polylines, etc.;

The Textbox is a simple ASCII text display gadget that provides ease and simplicity rather than great flexibility. Given the name of a disk file this gadget opens the file, reads the text in, builds slider bars for scrolling the text if necessary and displays the whole thing. The amount of text that can be displayed is limited to about 32k. An attempt to display a larger file will fail.
Text lines can either clip at the Textbox boundary, or be wrapped.

21.1. Data Types

```
typedef struct _textboxtype
{
    GadgetType  *gadget;                /* 'Superclass' gadget              */
    FontType    *font;                  /* Display font                     */
    char        *filename;              /* Filename                         */
    TextlistType *list;                 /* String list                      */
    int         nlines;                 /* Number of text lines that fit    */
    int         tlines;                 /* Total number of lines            */
    int         border;                 /* Box border flag                  */
    int         topline;                /* Points at top line in box        */
    int         bottomline;             /* Points at last line in box       */
    int         wordwrap;               /* If TRUE then wrap lines          */
    SliderType  *vslider;               /* Slider, if necessary             */
} TextboxType;
```

Just about all the fields of a Textbox gadget are calculated, except for the boundary rectangle and font.

21.2. Textbox Callback Procedure

Although a user function is a parameter when the Textbox is created, it is not currently used. There isn’t any way for the user to “select” a Textbox so no way for the system to call a callback function.

21.3. Interface Functions *smtext.h*

21.3.1. int SM_BuildTextList(char *fname, TextlistType **tl)

Given a file name *fname* and a pointer to a pointer to a TextlistType *tl*, opens the file, allocates the textlist, and builds the textlist using the text from the file. This is the same function called by the **SM_CreateTextbox** function.
Returns -1 if unable to open the file, otherwise returns the number of lines of text read.

21.3.2. TextboxType *SM_CreateTextbox(WindowType *w, RectType *r, char *filename, FontType *font, int borderflg, int wrapflag, int (*func)())

Creates a new Textbox attached to window *w* by reading the contents of the file *filename* and returns a pointer to it.

w	Window the Textbox will be attached to;
r................	Bounding rectangle for the Textbox in window-relative coordinates;
filename.....	Name, including path as necessary, of file to read (no wildcards allowed);
font	Font for displaying the text;
borderflg....	True if the Textbox boundary should be bordered, False otherwise;
wrapflag	If True then text will be wrapped at the boundary rectangle borders. Otherwise text ix clipped at the borders.
func...........	Application callback function, currently unused. Set to NULL for future compatibility.

21.3.3. int SM_DestroyTextbox(TextboxType *tb)

Frees a Textbox and all memory associated with it. Does not affect the text file.

21.3.4. int SM_SetTextboxRect(TextboxType *tb, RectType *r)

Sets the boundary rectangle for Textbox *tb* to the rectangle *r*, specified in window-relative coordinates (relative to the parent window of *tb*). If *tb*'s parent window is the focus window then the Textbox is recalculated and redrawn to reflect the new boundary rectangle. If necessary, a slider may be added to (or removed from) the Textbox.

21.3.5. int SM_TextboxLineDown(TextboxType *tb)

Causes the text in *tb* to scroll down one line, unless the bottom line of the text is already displayed.

21.3.6. int SM_TextboxLineUp(TextboxType *tb)

Causes the text in *tb* to scroll up one line, unless the top line of the text is already displayed.

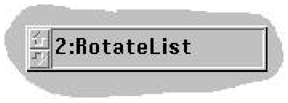
21.3.7. int SM_TextboxPageDown(TextboxType *tb)

Causes the text in *tb* to scroll down one page (number of lines that fits in the Textbox boundary), unless the bottom of the text is already displayed.

21.3.8. int SM_TextboxPageUp(TextboxType *tb)

Causes the text in *tb* to scroll up one page (number of lines that fits in the Textbox boundary), unless the top of the text is already displayed.

22. Rotatelist



The Rotatelist displays a sequence of strings, one at a time. The display includes a label area which shows the currently selected string, and two buttons for going to the next or previous string on the list. An attempt to go to the next string when the last string is displayed wraps around to the first string. Likewise, trying to go back from the first string wraps to the last string -- that is, the list is circular, hence the name Rotatelist.
The list of strings displayed by the Rotatelist is provided by the application; the gadget does not make a copy of it. It must therefore be static data relative to the gadget's lifetime. That last string as far as the gadget is concerned is the string preceding the first NULL string in the list. The strings are displayed using the system font defined at the time the list is drawn.

22.1. Data Types

```
typedef struct {
    GadgetType  *gadget;              /* Superclass gadget               */
    char        **list;               /* List of strings to display      */
    int         nlist;                /* Number of items in the list     */
    int         current;              /* Current string                  */
    int         align;                /* Left, right, center alignment   */
    ButtonType  *upbutton;            /* Scroll up                       */
    ButtonType  *downbutton;          /* Scroll down                     */
    LabelType   *label;               /* Display of selected string      */
    int         (*f)();               /* Application function            */
    void        *data;                /* Application data                */
} RotatelistType;
```

The *list* is the list of strings supplied by the application. These must be static since the Rotatelist doesn't copy them. The list is terminated by a NULL pointer. The number of items on the list *nlist* is calculated when the Rotatelist is created. *current* is the index for the string displayed in the gadget. *align* controls how the displayed string is positioned in the display and is one of the text alignment constants defined in smtypes.h: ALIGNLEFT, ALIGNRIGHT, or ALIGNCENTER.
The *upbutton, downbutton,* and *label* are TWS interface gadgets that are explicitly controlled by the Rotatelist gadget -- the application cannot "see" these gadgets. *f* is the gadget callback function, which must be an int function whose argument is a pointer to a Rotatelist. Finally, *data* is the application data field.

22.2. Rotatelist Callback Procedure

The Rotatelist callback function is called whenever the list selection is changed by pressing one of the buttons. It is an int function whose argument is a pointer to a Rotatelist. It's called after the selection change and after the buttons have been redrawn. The return value from the callback function is returned to the application event processor.
The callback procedure should not change the list selection value.

22.3. Interface functions *smmulti.h*

22.3.1. RotatelistType *SM_CreateRotatelist(WindowType *w, RectType *r, char **list, int align, int (*f)(), void *data)

Allocates and initializes a new Rotatelist gadget attached to window *w* and returns a pointer to it.

w .... Window to attach the Rotatelist to. If *w* is NULL then the gadget is allocated and initialized but not attached to any windows.
r.......... A rectangle for the outer boundary of the list. The selected string display label and control buttons are drawn inside this rectangle and relative to it -- the buttons are each about half the height of the rectangle, and the label is drawn just inside the border. Specified in window content coordinates.
list....... The *list* is the array of strings the Rotatelist uses. It must be a contiguous array of null-terminated strings and must be static data relative to the Rotatelist. The array itself must end with a NULL pointer element.
align.... Specifies the alignment of the selected string within the display label. Constants provided in smwindow.h are ALIGNLEFT, ALIGNCENTER, or ALIGNRIGHT.
f.......... Application function called whenever the selected string changes. May be NULL.
data .... Application-specific data. May be NULL

Example:

```
#include              <smwindow.h>
#include              <smmulti.h>
#include              <stdio.h>
:
:
static char           *strings[] = {“Black”,”Blue”,”Green”,”Yellow”,”Red”,(char *)NULL};
RectType              r;
WindowType            *w;
RotatelistType *rl;
:
:
r.Xmin = r.Ymin = 50;
r.Xmax = r.Xmin + 150;
r.Ymax = r.Ymin + 2*SM_GetSystemFontHeight();
rl = SM_CreateRotatelist(w, &r, strings, ALIGNLEFT, NULL, NULL);
```

22.3.2. int SM_SetRotatelistList(RotatelistType *rl, char **list)

Sets the list of strings for the Rotatelist *rl* to *list*. *list* must be static data in the application program relative to the lifetime of the gadget. The selected string is set to the first string in the new list and, if the Rotatelist’s parent window is the focus window, the Rotatelist gadget *rl* is redrawn. Returns 0 (the function cannot fail).

22.3.3. char *SM_GetRotatelistString(RotatelistType *rl)

Returns a pointer to the string currently selected in the Rotatelist *rl*. If the list in *rl* is empty then NULL is returned.

22.3.4. int SM_GetRotatelistSelected(RotatelistType *rl)

Returns the index of the string currently selected in the Rotatelist *rl*. This function cannot fail because the default index is 0 even for an empty list. The string may be NULL or undefined, however, depending on the values in the list of strings attached to the Rotatelist.

22.3.5. void *SM_GetRotatelistData(RotatelistType *rl)

Returns the data field of the rotatelist, generally a pointer to application data.

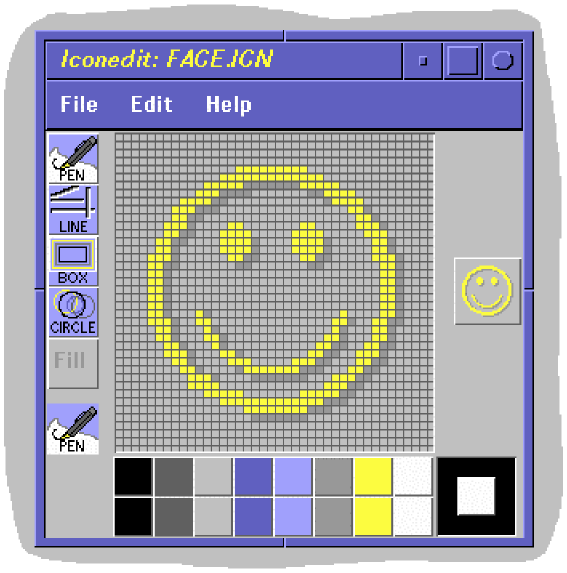
22.3.6. void SM_SetRotatelistData(RotatelistType *rl, void *data)

Sets the data field of the rotatelist *rl* to *data*. If there is already a value in the data field, the existing value is discarded.

22.3.7. int SM_SetRotatelistSelected(RotatelistType *rl, int n)

Sets the selected string in the rotate list to the one at index *n*, where 0 is the first string. If *n* is greater than the number of strings in the list then the current selection is unchanged and a non-zero value is returned; otherwise 0 is returned on success.
Setting the Rotatelist selection automatically calls the Rotatelist callback function after the new selection is set.

23. Pixmaps



A **pixmap** is a rectangular region of pixels. Pixmaps are used to decorate and highlight, as facings for buttons (as in the column of buttons on the left in the illustration) and labels, and as part of icons.
A pixmap has no functionality associated with it, but otherwise is treated much as any other gadget. In addition, pixmaps can be saved to disk and read back later. Since the pixmap is an array of colors, the appearance of a pixmap depends on the hardware color system and the color palette of the focus window at the time the pixmap is displayed.
Pixmaps are normally treated as static data (for example, as window icons or decorations for dialog boxes -- the pixmaps at the bottom left and center right in the illustration above) but there are also provisions for dynamically modifying a pixmap by directly setting its colors. However, since pixmaps are severely restricted in size (more about this later), they're not really suited for general graphics purposes.

23.1. Data Types

```
typedef struct {
    GadgetType  *gadget;             /* Superclass gadget               */
    int         *pixmap;             /* Array of color indices          */
    void        *image;              /* Screen image                    */
    int         redraw;              /* If True then pixmap was modified */
    int         border;              /* If True draw border around pixmap*/
    int         blocking;            /* If True then don't save image    */
                                     /*     after every pixel change     */
} PixmapType;
```

Notice that there are two types of pixmap "images" stored in the pixmap struct. One is a map of the color indices from the color table that was active when the pixmap was created, and the other is a screen image of the pixmap. These are not (necessarily) the same values. The screen image is stored so the pixmap can be redrawn quickly. The color indices image is needed in case the application modifies the pixmap while it's inactive, or the pixmap is saved to disk.
The blocking flag is an optimization feature. Normally, to keep the screen image current, it is saved after every modification to the pixmap. This can slow the application considerably if the pixmap is being updated rapidly. Setting the blocking flag prevents the screen pixmap image

from being saved. Clearing the blocking flag automatically saves the pixmap image if the pixmap's parent window has the focus.

23.2. Pixmaps and memory usage

Pixmaps are expensive gadgets. Every pixmap requires two large data sets. One is the matrix of color indices describing the pixmap image. This is the "real" pixmap as the artist designed it. This representation is needed whenever the pixmap must be regenerated on the screen, such as when the window is initially opened or when the pixmap has been obscured.
The other data set is a screen image of the pixmap -- that is, a matrix of pixel values. The pixmap image provides very fast redrawing of the pixmap, such as whenever a window is moved.

23.3. Creating a Pixmap

A pixmap can't be created "ready to wear" -- it has to be set up in two steps: first, the pixmap gadget is created using the **SM_CreatePixmap** function. Given a window and a boundary rectangle, **SM_CreatePixmap** allocates a new pixmap and attaches it to the window. The *content* of the pixmap at this point is empty.
The next step is to add colors to the pixmap. The function **SM_SetPixmapPixel** is used to set the individual pixels within a pixmap. **SM_SetPixmapPixel** is passed the pixmap, the X and Y coordinates within the pixmap to be set, and a ColorType to set the pixmap to. Only the color table index from the ColorType is used. If the pixmap's parent window is the focus window, the pixmap is updated on the screen as well.

```
PixmapType            *pm;
RectType              r;

r.Xmin = r.Ymin = 1;                    /* Create a 32x32 pixmap */
r.Xmax = r.Ymax = 32;
pm = SM_CreatePixmap(w, &r, False);
for (y = 0; y < 32; y++) {
        for (x = 0; x < 32; x++) {
                SM_SetPixmapPixel(pm, x, y, SM_GetSystemColor(5));
        }
}
```

A special note about the ColorType * argument to the **SM_SetPixmapPixel** function: only the *index* of the color passed is used (for LUT systems), and the index is assumed to be from the *system* color table. If the application sends in window colors the pixmap may display with the wrong colors.
The example above creates a pixmap and fills it with a solid color. More likely your application will either draw in the pixmap or read it from disk.

23.4. Reading and Writing Pixmaps from Disk

The other way to create a pixmap is to read it from a disk file. The function **SM_ReadPixmap** is passed a window and a file name, and it reads the pixmap file, creates a new pixmap, loads the pixmap with the image from the file, and returns the new pixmap. The pixmap as returned by **SM_ReadPixmap** has no parent window and its "position"[12] is at (0,0). Once the application has the pixmap, it must be mapped to a window.
A pixmap gadget created in this manner is freed normally when the window is closed.

```
PixmapType            *pm;

pm = SM_ReadPixmap("mypixmap.pxm");
if (pm) {
```

[12]Yes, but relative to what? The answer is, nothing at all. An example of "coordinate-free" positioning -- a position with no reference frame! The mapping will set the frame and thus establish the "real" position of the pixmap.

```
        SM_AttachGadget(w, SM_GetGadgetSuperclass(pm));
        SM_MovePixmap(pm, 10, 5);
}
```

The example above verifies that the pixmap was read successfully (**SM_ReadPixmap** returns NULL on failure), then attaches it to a window using **SM_AttachGadget**. Finally, the gadget is moved into position by **SM_MovePixmap**. If the window w is the focus window the user will see the pixmap displayed twice; once when it's attached, then again when it's moved. We could prevent this by moving it first (and probably would in a real application).

The colors in a pixmap file must be in the TWS 8-bit RGB format. If the pixmap was written using the TWS function **SM_WritePixmap** then this is the case. The pixmap gadget itself contains only the LUT indices for the colors in the pixmap. These are interpreted by the pixmap draw and write functions as indices into the system color table. This policy has two effects: for one, the actual RGB colors written to a pixmap file depend on the current color palette, which in turn depends on the current active window, among other things. If a pixmap is written to disk while some other window with a different graphics color palette is active, the pixmap may store colors different from those intended.

On the plus side, using system colors allows pixmaps that don't change when windows load their own palettes, because the pixmaps can access reserved system colors. This is very desirable for icons and other user interface uses.

Application developers can control the system color table outside the reserved system colors by simply filling a window color table and making the window active. In this way both fixed system colors and window colors can be applied to pixmaps.

23.5. Interface Functions *smpixmap.h*

23.5.1. void SM_ClearPixmap(PixmapType *pm, ColorType *c)

Sets all the pixels in a pixmap to the color *c*.

23.5.2. PixmapType *SM_CreatePixmap(WindowType *w, RectType *r, int Borderflag)

Allocates and returns a pointer to a new pixmap whose dimensions are determined by the rectangle *r*. The resulting pixmap is attached to window *w* and its contents are empty. If *borderflag* is True then the pixmap will display with a raised bevel border; otherwise the pixmap has no border.

23.5.3. PixmapType *SM_ReadPixmap(char *filename)

Allocate a new pixmap, read its contents from the file *filename* and return a pointer to the pixmap. The pixmap is not displayed and is not attached to any window. Its upper-left corner is at position (0,0). If *filename* doesn't exist, can't be opened or doesn't contain a valid pixmap then NULL is returned.

23.5.4. void SM_MovePixmap(PixmapType *pm, int dx, int dy)

Change the position of a pixmap by *dx,dy* pixels. Positive values move the pixmap down and to the right, negative values up and to the left. If the pixmap is part of the focus window then it's redrawn immediately.

23.5.5. int SM_GetPixmapBlocking(PixmapType *pm);

Returns the blocking flag for the pixmap, either True or False.

23.5.6. void SM_SetPixmapBlocking(PixmapType *pm, int blocking);

Sets the pixmap blocking flag to *blocking*. If the value is False blocking is disabled, any other value enables blocking. When blocking is enabled then setting pixels in the pixmap will not be drawn on the screen.

23.5.7. void SM_SetPixmapPixel(PixmapType *pm, int x, int y, ColorType *c);

Set the value of the pixmap pixel *x,y* to the color *c*. The color is assumed to be from the system color table. For LUT color systems the pixmap is only concerned with the index value of *c*.

23.5.8. int SM_DestroyPixmap(PixmapType *pm);

Free a pixmap and all resources associated with it. If the pixmap's parent window has the focus it's redrawn immediately with the pixmap removed.

23.5.9. ColorType *SM_GetPixmapPixel(PixmapType *pm, int x, int y, ColorType *c);

Return the color value from the pixmap at position *x,y*. The color is stored in *c* and contains the color index and RGB values taken from the system color table.

23.5.10. int SM_GetPixmapWidth(PixmapType *pm);

Returns the width of the pixmap in pixels.

23.5.11. int SM_GetPixmapDepth(PixmapType *pm);

Returns the depth of the pixmap in pixels.

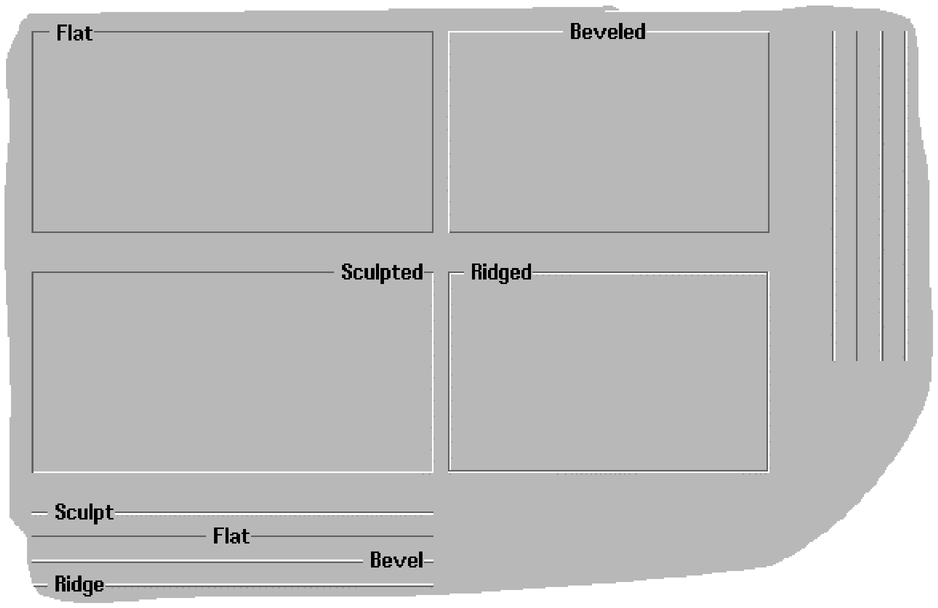
23.5.12. int SM_WritePixmap(PixmapType *pm, char *fname);

Write the contents of the pixmap to the file named *fname*, which can contain drive and path information but no wildcard characters. If the file exists its contents are replaced by the pixmap data without warning.
The pixmap is written to the file in a device-independent way that includes the RGB values of the original pixmap in TWS 8-bit format. This allows the most flexibility in mapping the pixmap to the existing color resources when it's read back in.
Returns 0 on success, any other value indicates failure.

23.5.13. void SM_CopyPixmap(PixmapType *dest, PixmapType *source);

Make a duplicate of the *source* pixmap in the *dest* pixmap. The destination pixmap must already be created at least large enough to hold the entire contents of the source pixmap. If the destination is smaller than the source then some source pixels will be lost, but this is not considered an error.

24. Borders



Borders are beauty gadgets, designed to provide visual organization and decoration to TWS applications. They have no functionality and cannot have the input focus. They serve roughly the same purpose as the lines and rectangles that organize text and graphics in a printed document. Since they are gadgets, however, the application doesn't need to create a graphics state, and the window manager takes care of positioning, sizing, and clipping them once they're created.

24.1. Border Types and Styles

There are three border types. Each type of border can be one of five styles. Types and styles are identified by constant values defined in smborder.h. The types are:

Type	***Value***	***Description***
twsBORDERHLINE	1	A horizontal line.
twsBORDERVLINE	2	A vertical line.
twsBORDERBOX	3	A rectangular box.

Each of the border types can be one of five styles. The styles are:

Style	***Value***	***Description***
twsBORDERFLAT	1	Simple solid color outline.
twsBORDERBEVEL	2	A rectangle with a 'raised' appearance. The leading edge is drawn in white, the trailing edge in the border's specified color.
twsBORDERSCULPT	3	A rectangle with a 'recessed' appearance. The leading edge is drawn in the border's specified color, the trailing edge is drawn in white.

twsBORDERRIDGE	4	The appearance of a raised ‘bead’ around the rectangle. This style requires two lines; the leading line is drawn in white, the trailing line in the border’s specified color. The total border width is twice the specified width. This is identical to the twsBORDERBEVEL style when applied to horizontal or vertical lines.
twsBORDERLRIDGE	5	The appearance of a recessed ‘groove’ around the rectangle. This style requires two lines; the leading line is drawn in the border’s specified color, the trailing line in white. The total border width is twice the specified width. This is identical to the twsBORDERSCULPT style when applied to horizontal or vertical lines.

The box and horizontal line border types can have an optional title or border label. When applied, the label is drawn at the top of the box, and can be positioned in the center of the box or line, or at the left or right edge. Constants that control this are defined in smborder.h:

Label Position	***Value***	***Description***
twsBORDERLEFT	1	Position the label about 10 pixels from the left edge of the border, and along the top edge for boxes.
twsBORDERRIGHT	2	Position the label about 10 pixels from the right of the border, and along the top edge for boxes.
twsBORDERCENTER	3	Center the label on the border, along the top edge for boxes.

Vertical borders will not display labels, but it is not an error to provide one.

24.2. Interface functions *[smborder.h]*

24.2.1. BorderType *SM_CreateBorder(WindowType *w, RectType *r, char *label, int position, int type, int style, int thick, ColorType *color)

Create a new border gadget and return a pointer to it. Values are:

w	Parent window for the border gadget;
r................	Rectangle in window-relative coordinates describing the outside boundary of the border. For multi-line thick borders, the extra lines are drawn inside this rectangle
label	Border label. Ignored if the border is a vertical line;
position......	Constant defining the position for the label, defined in smborder.h. Ignored if the label is a vertical line;
type...........	Type of border -- horizontal or vertical line, or box. Constants defined in smborder.h;
style	Style of border -- sculpted, beveled, flat, beaded, or grooved, according to constants defined in smborder.h;
thick	Thickness of the border. Must be at least 1. Additional lines of thickness are drawn inside the border. The *xxx*RIDGE styles require two lines per thickness, so their actual thickness is *2*thick*.
color	Color of the border. For 3D border styles, the highlighted edge is always in white, the shadowed edge is in *color*. May be any system or window color.

It’s important that the rectangle *r* be large enough to enclose all parts of a horizontal or vertical line border. These are not single lines. For lines, the border is not drawn and will not cause underlying window elements to be obscured — only the border lines themselves are drawn. However, the lines *are* clipped within the specified rectangle.

24.2.2. int SM_DestroyBorder(BorderType *border)

Frees the border *border* and removes it from the window.

ColorType *SM_GetBorderColor(BorderType *border) *[macro]*

Returns the border's color. This is a pointer to the border's color field and should normally be treated as read-only.

24.2.3. char *SM_GetBorderLabel(BorderType *border) *[macro]*

Returns the label for the border. This is a pointer to the label in the border structure itself, not a copy, and should be treated as read-only.

int SM_GetBorderPosition(BorderType *border) *[macro]*

Returns the label position constant from the *border*. Values are:

Label Position	*Value*	*Description*
twsBORDERLEFT	1	Position the label about 10 pixels from the left edge of the border, and along the top edge for boxes.
twsBORDERRIGHT	2	Position the label about 10 pixels from the right of the border, and along the top edge for boxes.
twsBORDERCENTER	3	Center the label on the border, along the top edge for boxes.

int SM_GetBorderStyle(BorderType *border) *[macro]*

Returns the border style constant from the *border*. Values are:

Style	*Value*	*Description*
twsBORDERFLAT	1	Simple solid color outline.
twsBORDERBEVEL	2	A rectangle with a 'raised' appearance. The leading edge is drawn in white, the trailing edge in the border's specified color.
twsBORDERSCULPT	3	A rectangle with a 'recessed' appearance. The leading edge is drawn in the border's specified color, the trailing edge is drawn in white.
twsBORDERRIDGE	4	The appearance of a raised 'bead' around the rectangle. This style requires two lines; the leading line is drawn in white, the trailing line in the border's specified color. The total border width is twice the specified width. This is identical to the twsBORDERBEVEL style when applied to horizontal or vertical lines.
twsBORDERLRIDGE	5	The appearance of a recessed 'groove' around the rectangle. This style requires two lines; the leading line is drawn in the border's specified color, the trailing line in white. The total border width is twice the specified width. This is identical to the twsBORDERSCULPT style when applied to horizontal or vertical lines.

int SM_GetBorderThick(BorderType *border) *[macro]*

Returns the border thickness, in pixels. For the ridged border styles, the actual number of pixels used to draw the border is 2* the border thickness.

int SM_GetBorderType(BorderType *border) *[macro]*

Returns the border type constant, which identifies the border as a line or box. Constant values are:

Type	*Value*	*Description*
twsBORDERHLINE	1	A horizontal line.
twsBORDERVLINE	2	A vertical line.
twsBORDERBOX	3	A rectangular box.

24.2.4. RectType *SM_GetBorderRect(BorderType *border)

Returns a pointer to a RectType structure containing the border's rectangle in window-relative coordinates. If the border was specified in virtual coordinates, they are resolved first, so the returned rectangle is the actual coordinates for the border at the time the function is called. When finished, the application must free the rectangle.

```
RectType       *borderrect;
BorderType     *border;
/*...*/
borderrect = SM_GetBorderRect(border);
/*
** Additional processing....
*/
free(borderrect);
```

24.2.5. void SM_SetBorderLabel(BorderType *border, char *label)

Sets the title string used for the border. If the border is part of the active window, the border is redrawn to reflect the change. The existing label, if any, is discarded and its memory reclaimed.

void SM_SetBorderPosition(BorderType *border, int position)

Sets the border label position to *position*, which must be one of the position constants defined above. If the border is part of the active window then the border is redrawn to reflect the change.

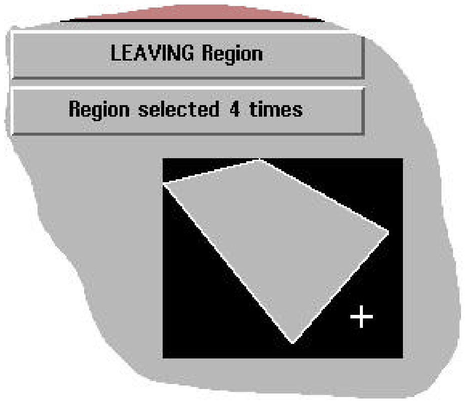
24.2.6. void SM_SetBorderRectangle(BorderType *border, RectType *r)

Sets the size and position of the border by setting the rectangle. The rectangle *r* can be virtual. If the border is part of the active window, its redrawn to reflect the new border size.

24.2.7. void SM_SetBorderThick(BorderType *border, int thick)

Changes the thickness of the border *border* to *thick*. If the border is part of the active window it's redrawn to reflect the change. Note that multi-line borders draw the extra lines to the inside of the border rectangle.

25. Hotregion



The Hotregion gadget is a polygonal area that is sensitive to an extended number of mouse and keystroke events. However, unlike all other TWS gadgets, the Hotregion has no visual representation at all. When an event occurs within the Hotregion, the gadget simply calls the appropriate application callback function for that event.
The Hotregion then is unique in the following ways:

- It's sensitive area is defined by a closed polygon, rather than a rectangle;
- There's no intrinsic visual for a Hotregion;
- It responds to an extended set of mouse and keystroke actions, including: Cursor enters region; cursor leaves region; mouse button clicked in region; and cursor dragged in region. Each of these events has its own callback function.

Since a Hotregion has no visual, it can be set over a graphics image without disturbing that image.

25.1. Creating a Hotregion

The hotregion is the only TWS gadget whose boundary is a polygon rather than a rectangle. This allows the hotregion to enclose complex areas. The polygon may be non-convex or complex; TWS uses the “odd-even” rule for determining polygon boundaries. The polygon must be closed — that is, the last point and first point must be the same.
The function **SM_CreateHotregion** creates a hotregion:

```
int                   npts;
PointType             *pts;
WindowType            *w;
HotregionType  *hotregn;
int                   (*userfunc)(HotregionType *),
                      (*enterfunc)(HotregionType *),
                      (*leavefunc)(HotregionType *),
                      (*dragfunc)(HotregionType *);
/* ...create polygon points... */
hotregn = SM_CreateHotregion(w, npts, pts, enterfunc, leavefunc, dragfunc, userfunc);
```

The values *npts* and *pts* define the polygon region where the hotregion is sensitive. They're the number of points in the polygon and the list of points, respectively. The point coordinates are relative to the window content region. The hotregion callback functions are int functions that take a pointer to a HotregionType as their argument.

25.2. Using Hotregions

Like all other TWS gadgets, the hotregion polygon is specified in window content coordinates. However, many applications will want to use hotregions to add interactivity to graphics and imagery. For example, an application might load an image of an office into a window, then set hotregions over various parts of the scene (books, the phone, printer, etc.) to make the image interactive.

TWS provides a number of convenience functions for converting coordinates to and from device, window, graphics canvas, and other frames. The function **SM_PtCanvasToContent** takes an x,y in window graphics canvas coordinates and returns a corresponding x,y in window content coordinates. See the Utilities chapter for details.

Another help is that the TWS event structure contains the cursor coordinates in device, window content and graphics canvas coordinate frames.

25.2.1. Hotregion Callback Functions

The callback functions for a Hotregion are int functions whose argument is a pointer to the Hotregion gadget an event occurred in. There are five callback functions for a hotregion, corresponding to four different event types:

- A function to draw the hotregion. This function is called whenever the gadget would have to be drawn. Most TWS gadgets supply their own draw functions, but Hotregions must be drawn by the application[13]. The draw function is an int function whose argument is a pointer to the hotregion to be drawn. Remember that the polygon points in a hotregion are in window content coordinates;

- A function to respond to an ENTER event. This function is called when the mouse cursor goes from outside the hotregion to inside of it. An application may want to change a region’s appearance, change the cursor or provide some other visual clue to the user when the cursor enters a hotregion;

- A function to respond to a LEAVE event. This function is called when the mouse goes from inside to outside the hotregion. Often an application will restore a region’s appearance to some default when this happens;

- A function to respond to a DRAGMOTION event. This function is called when the user drags the mouse inside the hotregion. It isn’t possible to drag the mouse outside the hotregion — if the cursor crosses the hotregion boundary while being dragged, a LEAVE event is generated and the region will ignore button drags until the mouse is again within the region (but see the **SM_HotregionMove** function);

- A function to respond to BUTTONPRESS events. The application can use this function to allow the user to ‘select’ the hotregion.

Any of these functions may be NULL in which case the region ignores the event. If the DRAW function is NULL then the hotregion won’t be visible, but will still respond to events.

[13]Or not. The application doesn’t *have* to draw anything at all. The hotregion will respond to events just as well when it’s unseen.

25.3. Interface Functions *[hotregn.h]*

25.3.1. HotregionType *SM_CreateHotregion(WindowType *w, int npts, int *pts, void *data, int (*drawfunc)(), int (*enterfunc)(), int (*leavefunc)(), int (*dragfunc)(), int (*userfunc)())

Create a new hotregion. The *pts* array is composed of (x,y) pairs of integers in the window content coordinate frame. *npts* is the number of coordinate points in the array *pts*, such that for all i < *npts*, (i*2) is the x coordinate, and (i*2+1) is the y coordinate.
Each of the functions is an application function whose argument is a pointer to a HotregionType. The system calls the appropriate function on certain Hotregion events. If any function is NULL then that event is effectively ignored.

25.3.2. void SM_HotregionMove(HotregionType *hr, int dx, int dy)

Translate the hotregion a distance (*dx,dy*). Positive numbers are to the right and down, respectively.

25.3.3. int SM_DestroyHotregion(HotregionType *hr)

Remove a hotregion and free its memory. Like all TWS gadgets, hotregions are destroyed automatically whenever their parent window is closed.

25.3.4. void *SM_GetHotregionData(HotregionType *hr) *(macro)*

Returns the data pointer of a hotregion.

25.3.5. int SM_GetHotregionNpts(HotregionType *hr) *(macro)*

Returns the number of points in the hotregion polygon.

25.3.6. int *SM_GetHotregionPts(HotregionType *hr) *(macro)*

Returns the polygon array for the hotregion.

Part Four

Utilities

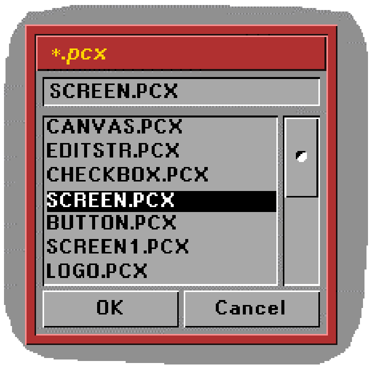
26. Standard Dialogs

A *standard dialog* is simply a modal dialog function supplied with the TWS distribution. These are provided as a simple and standard way of performing a number of routine user interface tasks.
First of all, a *dialog* is a window that contains information for the user which he may select or respond to. Typically, dialogs are differentiated from ordinary program windows in several ways: They have a specific purpose, such as informing the user of an error or allowing a file selection; they are often not application-specific (any number of applications might want to allow the user to select a file); and they are typically on the screen for just a short time.
A *modal* dialog box won't let the user proceed with the program until he has responded to the dialog. A *non-modal* dialog is more like a regular window, and will allow the user to move on to other things and come back to the dialog. Writing a non-modal dialog is simply a matter of creating a window, inserting the appropriate gadgets, and attaching the action function to an OK button.
Modal dialogs are a bit different. They have to prevent the user from changing the active window and must interrupt the application processing. In a way, they're somewhat antithetical[14] to the whole event-driven paradigm. However, it'd be hard to design lots of things without them.
A TWS a standard dialog is called just like a regular C function. The application branches off to the dialog function, and when the user has responded to the dialog the function returns. The value returned by the dialog function indicates what the user did. For example:

```
rsp = SM_YesNoDialog(“Do you want to quit now?”, “Yes”, “No”);
```

displays the quoted message in a window, along with a button for OK (that is, yes) and a button for Cancel (that is, no). The value returned depends on which button the user pressed. The function won't return until the user presses one of the buttons, and by the time it does return the dialog window has been closed and the system state is as it was before the function was called.
The following dialog functions are provided:

26.1. Picklist



A picklist is a dialog window containing a Label, a Stringlist, a Slider, and two Buttons. The purpose of the picklist is to allow the application user to select a string from a list of strings The index into the list of strings of the string selected is returned.

26.1.1. Usage

```
#include       <smpickls.h>
int     n;
```

[14]Every now and again I like to drop in a five-dollar word, just so you'll think you're getting your money's worth.

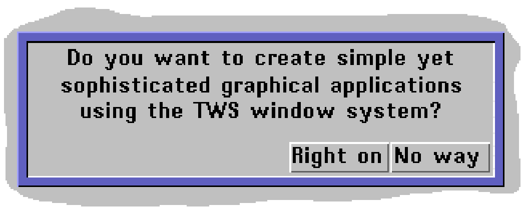
```
n = SM_Picklist(char *title, RectType *box, char **list, int nitems, int nstart, int
selected, int destroyflag)
```

title	Title for the picklist window;
box............	Rectangle for the *stringlist* portion of the picklist. The rest of the picklist window is built around this box.
list.............	Static array of strings to be displayed. Same as the *Stringlist* data type;
nitems	Number of items in the *list* array. Same as the *Stringlist* data type;
nstart	Index of the first element in *list* to display in the *box.* Same as the *Stringlist* data type;
selected	Index of the initially selected string in *list*.
destroyflag .	If true, the strings in *list* will be freed when the picklist window is closed, and the list itself is freed. This assumes both the strings and the list of strings were dynamically allocated.

26.1.2. Returns

Index of string user selected, or -1 if Cancel.

26.2. YesNoDialog



Do you want to create simple yet sophisticated graphical applications using the TWS window system?

Displays a message in a window along with two buttons. The left button returns 0, the right button 1. The button labels are supplied by the application along with the message, and the buttons are both built large enough to hold the longest button string (i.e., the buttons are the same size). The window is centered on the screen, it's width is 1/2 the screen width. The text message is word-wrapped as necessary to fit in the window, and the window depth is adjusted to accommodate the wrapped lines and buttons.

26.2.1. Usage

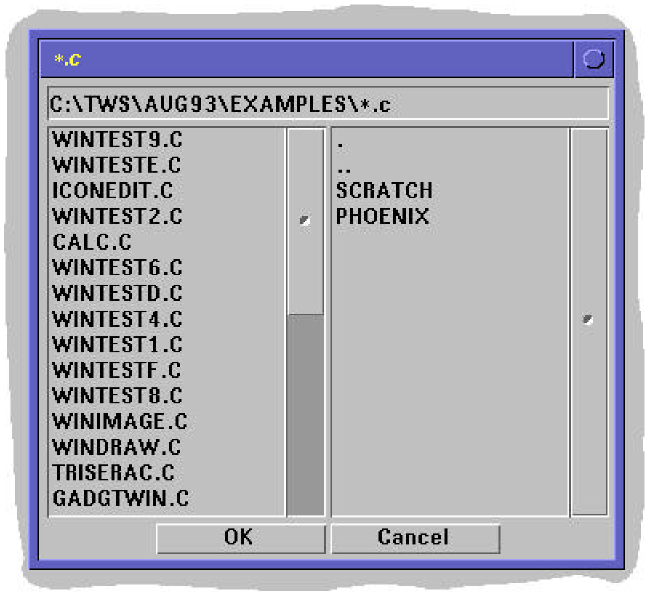
```
#include       <smyesno.h>
int     n;
char    *msg, *yesbuttonstring, *nobuttonstring;
n = SM_YesNoDialog(msg, yesbuttonstring, nobuttonstring);
```

msg................	Character string (NULL-terminated) message to be displayed.
yesbuttonstring.	Character string displayed in the YES button (left)
nobuttonstring..	Character string displayed in the NO button (right)

26.2.2. Returns

0 if left button pressed 1 if right button pressed.

26.3. File Selection Dialog



Displays lists of files and subdirectories. The user can select a file in the current directory by clicking on its name in the file list, or can change to a subdirectory by clicking on a directory name, or on '..' to go up the tree. When a file is selected it's appended to the path and displayed in an editstring. When the directory is changed or the filename string is unselected, the filename in the editstring is replaced by the file template.
The user may also enter a name by directly typing in the editstring. If a new directory path is entered, the dialog will read it and load its contents into the lists. If a new file template (a file name portion that includes wildcard characters) is entered a new list of files is presented based on the template.

26.3.1. Usage

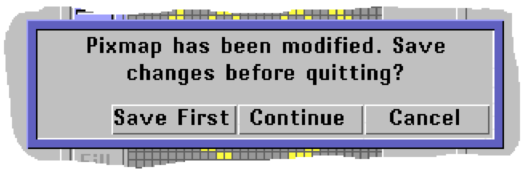
```
#include       <filedial.h>
char    *filename;
char    *template;
filename = SM_GetFilename(template);
```

template ....Character string, a DOS file template including path and wildcards as necessary. The dialog will only display files matching the template. However, all subdirectories will be displayed.

26.3.2. Returns

A malloc'd string containing the fully qualified file name selected (i.e., drive, path, file name and extension), or NULL if the user presses the Cancel button. The application should free the string when it's no longer needed.

26.4. Three-state Dialog


Pixmap has been modified. Save changes before quitting?

Similar to the YesNoDialog except there are three buttons instead of two. This dialog uses the same text wrapping and window sizing behavior. The application supplies the message and the text for the three buttons.

26.4.1. Usage

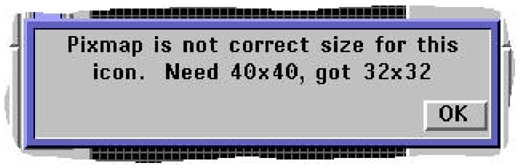
```
#include       <smync.h>
int     n;
char    *msg, *yeslabel, *nolabel, *cancellabel;
n = SM_YesNoCancelDialog(msg, yeslabel, nolabel, cancellabel);
```

msg.................A null-terminated character string containing an information message to the user.
yeslabel...........A character string containing the label for the leftmost button;
nolabel............A character string label for the center button;
cancellabel.......A character string label for the right button.

26.4.2. Returns

0 for the left button pressed; 1 if the center button pressed; -1 if the right button pressed;

26.5. One-state Dialog



Pixmap is not correct size for this icon. Need 40x40, got 32x32

A message dialog with only a single button. The dialog function simply returns a 0 when the user clicks the button, which simply displays “OK”.

26.5.1. Usage

```
#include       <smok.h>
int     n;
char *msg;
n = SM_OKDialog(msg);
```

msg.................A null-terminated character string containing an information message to the user.

26.5.2. Returns

0 when the user presses the button.

27. Utilities

These are functions that provide general assistance in some way, such as converting points to different coordinate systems (different windows), screen dumps and so forth. As such this section is a hodgepodge of stuff without any underlying data structures or semantics.

27.1. Interface functions *smwutil.h*

27.1.1. void SM_PtCanvasToContent(WindowType *w, int xo, int yo, int *x, int *y)

Takes a point (*xo,yo*) in graphics canvas coordinates and returns the same location in window content region coordinates, all for the same window *w*.

27.1.2. void SM_PtDeviceToContent(WindowType *w, int xo, int yo, int *x, int *y)

Takes a point x_o,y_o in device coordinates and returns the corresponding *x,y* location relative to the window *w*. If the original point is not within the window content then the resulting *x,y* could be negative or greater than the content boundary.

27.1.3. void SM_PtContentToCanvas(WindowType *w, int xo, int yo, int *x, int *y)

Takes a point x_o,y_o in window coordinates (relative to the window content region) and returns the corresponding point relative to the same window's graphics canvas. If the window doesn't have a canvas region then the return value is unchanged.

27.1.4. void SM_PtDeviceToCanvas(WindowType *w, int xo, int yo, int *x, int *y)

Takes a point x_o,y_o in device coordinates and returns the corresponding location in window *w*'s graphics canvas. If the window doesn't have a graphics canvas then the returns value is unchanged.

27.1.5. void SM_PtCanvasToDevice(WindowType *w, int xo, int yo, int *x, int *y)

Takes a point x_o,y_o relative to the graphics canvas region of window *w* and returns the corresponding global device coordinates in *x,y*.

27.1.6. void SM_PtContentToDevice(WindowType *w, int xo, int yo, int *x, int *y)

Takes a point x_o,y_o relative to the content region of window *w* and returns the corresponding global device coordinates in *x,y*.

27.1.7. int SM_SaveAsPCX(char *fname, RectType *r)

Given a rectangle *r* in device coordinates, writes the enclosed region of the screen to a PCX-format graphics file names *fname*. If *fname* exists already it is overwritten. The graphics region saved includes the rectangle boundary. The color format (16 or 256-color) of the PCX image is based on the graphics kernel mode when the file is written.
Returns 0 on success, any other value indicates failure (probably disk full).
Note: Truecolor graphics kernel modes cannot be saved as PCX files. See also: SM_SetPrintscreenProc.

27.1.8. int SM_SaveWindowAsPCX(WindowType *w, char *fname)

Saves the window *w* as a PCX file named *fname*. If *fname* exists it's overwritten. The entire window, including borders, is saved. The color format (16 or 256-color) of the PCX image is based on the graphics kernel mode when the file is written.
Returns 0 on success, any other value indicates failure (probably disk full).

Note: Truecolor graphics kernel modes cannot be saved as PCX files. See also: SM_SetPrintscreenProc.

27.1.9. int SM_SaveScreenAsPCX(char *fname)

Saves the entire graphics display screen to a PCX file named *fname*. If *fname* already exists it is overwritten. The color format (16 or 256-color) of the PCX image is based on the graphics kernel mode when the file is written.
Returns 0 on success, any other value indicates failure (probably disk full).
Note: Truecolor graphics kernel modes cannot be saved as PCX files. See also: SM_SetPrintscreenProc.

27.1.10. int SM_SaveAsPostscript(char *fname, RectType *r)

Writes the contents of the rectangle *r* to a disk file in a monochrome PostScript™ format. Display colors are converted to appropriate shades of gray. The resulting file can be sent to a PostScript™ printer directly. Returns 0 on success, any other value indicates failure (probably disk full). Unlike the PCX screen dump facilities, truecolor graphics modes can be saved. *See also: SM_SetPrintscreenProc.*

SM_SetRectangle(static RectType *r, int x, int y, int width, int height)

Sets a TWS rectangle format (Xmin, Ymin, Xmax, Ymax) from a Xmin, Ymin, width and height set of arguments. The *x* and *y* arguments are always set into the rectangle verbatim. A positive *width* or *height* argument is used to calculate an appropriate Xmax or Ymax, as in Xmax = Xmin + width - 1. A negative value is set into the rectangle verbatim. The result is that **SM_SetRectangle** can correctly set either normal or "virtual" coordinate rectangles.

Part Five

Tutorial

28. Tutorial

In the next few pages we'll try to put some of the previous tools and techniques together into some real applications. Writing event-driven programs for interactive applications is hardly ever "easy" (there are always so many darn *details* to look after!), but with TWS you have a collection of tools that ought to make the job far more manageable.

All of the code presented is included in the distribution. There are many more example and demo files in the distribution than are presented here. A table near the beginning of this document lists all the example code files and what is emphasized in each one. It'll be helpful if you have the example programs to refer to as you go through the following brief tutorial.

28.1. Getting Organized

Throughout this manual we've talked about some important concepts like event-driven programs, object-centered design and so on. Let's stop now and define just what the heck all this means.

28.1.1. Event-Driven Programming

First of all, remember that in an event-driven environment, the user performs actions based on the interface the program presents to him. The application then responds to the user's actions. This is just the opposite of the classical approach, in which the program makes demands of the user (enter *this* value *now*), and the user responds to the program. In general then an event-driven program won't be a linear path from beginning to end. Instead it'll be a collection of relatively independent modules which have a predefined connection to the user interface.

Here's an illustration. Suppose a CD player operated the way an old-fashioned linear program did. First of all, the CD player would have only three buttons on it (On/Off, Yes, and No) and maybe a panel so the player could show you messages. When you turned the CD player on (i.e., "ran the program") the CD player would ask you to insert a disk and press the "Yes" button. If you pressed the button without putting the disk in it would ask you again, and keep asking you until you either put in a disk or turned off the power.

When you put the disk in it would ask if you want to play track one. Your only choice would be to press Yes or No or turn off the player. If you answer no it might ask if you want to play track two, and so forth through the selections.

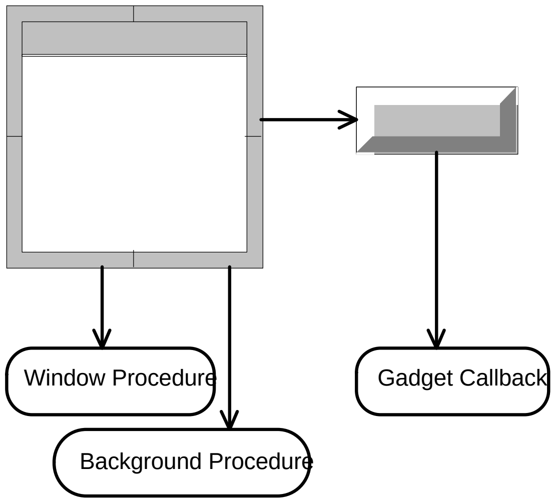
An event-driven program works more like a real CD player. When the Play button is pressed, the player must determine the system state. Is a CD inserted? Check and see. Is the disk already playing? Check and see. Play the first track or select a track at random? Check if the Random Selection button is active or if a specific track button was pressed, otherwise do some 'default' action. And so on.

Few users would opt for a CD player that worked like the first scenario but it would probably be a lot easier to build. The same goes for an event-driven program. The goal is to produce applications that are easier for users but, until you get used to it, it's a somewhat more complicated approach.

28.1.2. "Object-centric" Programming

Event-driven programming seems to go hand-in-hand with an "object-centric" approach to application development. The TWS user interface tools can be used in many different ways, but the most common approach integrates program data and user interface components (gadgets and windows) into program *objects*. Windows contain collections of these objects and might be considered *containers* for a program's basic objects. In object-centric lingo, we would say that a TWS GUI object *encapsulates* its own behavior (the visual appearance of a button, for example), data (through the button's data pointer field), and operations on the data (through the button's internal code and application callback function) into a single package, the TWS gadget or window.

TWS is “object-centric” rather than “object-oriented” because TWS lacks some true object-oriented features, notably inheritance and polymorphism.
Callback procedures are central to TWS applications. A callback procedure is a function written by the application programmer. The callback procedure can examine and modify the data attached to its parent object. Of course, in the end a callback procedure is just another function in your application.



28.1.3. Designing a TWS Program

The key to successful TWS programs is the same as for any programming. Applying the TWS GUI library to your projects only affects somewhat the programming *style*, not the fundamental *concepts* of software development. As always, your projects will proceed more smoothly if you apply the principles you already know:

28.1.3.1. Understand the problem

Take a little time to explore the problem you’re trying to solve in software. What should the program do, and how will it get done? What should it *not* do? What features are most important right away? What features will likely be added later? Make notes.

28.1.3.2. Find the objects in the problem

What are the *things* in the problem? How can your things be represented in code? Images? Documents? Files? What operations need to be performed on these things? How are different things in the program related?

28.1.3.3. Sketch organization of objects

As you’re getting a handle on the program’s objects, sketch their attributes, operations, and relationships. Think about a user accessing and manipulating your objects.

28.1.3.4. Use windows intelligently

Lots of times a program object will be presented to the user as a window or dialog, with gadgets providing access to the object’s internal data and functionality. Once the organization of objects in your program starts developing, you can start sketching window and gadget layouts.

28.1.3.5. Plan for errors

Remember that in an event-driven program the user almost always has many different choices he can make. Usually some of those choices will be incorrect in some way.

28.1.3.6. Iterate

Creating software is always an interative process. Be open to new understanding and insights which will change and improve the program.

28.2. The Basics

This first few lessons don't do anything particularly interesting. They basically show setting up the TWS framework, #include files, initialization and so forth. Bear with me.
The simplest TWS program simply establishes the graphics environment, turns on the application, and sits in its event loop. No menus, no fancy graphics. Not very interesting. But this basic structure will be repeated over and over in your real applications, so let's look at ***wintest2.c***...

```
#include <stdio.h>                    /* Required for NULL definition     */
#include <smwindow.h>
#include <smevent.h>
#include <smtypes.h>

main(int argc, char **argv)
{
    EventType  ev;
    int        msg;

    SM_Init(0, 0);               /* Initialize graphics system       */
    SM_OpenApplication("TWS Windows", NULL); /* Create application screen    */
    while (True) {
        SM_GetNextEvent(&ev);
        msg = SM_ProcessEvent(&ev);
        if (msg == KEYPRESS) {
            SM_Exit(NULL);
        }
    }
}
```

This is a complete, working TWS program that does nothing but display its own name. It's singular distinction is that it includes everything any TWS application program needs to work properly. Let's look at those things in detail.

28.2.1. Header files

Each component of the TWS library has a header file associated with it. For example, applications that use Buttons will `#include smbutton.h`, those that use pixmaps will `#include smpixmap.h` and so forth. Every TWS application uses windows (the workspace is a window), events, and fonts, so those headers are required for every application. The smtypes.h header file contains additional global types and constants and should also be included.
The ordering of header files is unimportant. Most header files in turn `#include` whatever additional header files they need internally, so the application doesn't have to worry about multiple dependencies.

28.2.2. System initialization

The function **SM_Init** puts the video system into the appropriate graphics mode, initializes internal TWS data structures, reads the TWS.CFG configuration file, loads fonts, and in general prepares the hardware and software to support a GUI. Much of this is very hardware-dependent, particularly the graphics and mouse hardware.
The parameters to **SM_Init** are the code values used by the underlying graphics kernel system to set the graphics card mode and mouse device support. For example, for a MetaWINDOW

graphics kernel, you could pass the macros `VESA1024x768X` and `MsCOM1` for a VESA video card in 1024x768x256 color mode, and the Microsoft Mouse driver on Com 1, respectively.
If a parameter is 0, **SM_Init** uses the device value from the TWS.CFG configuration file. In this case, the value in the configuration file must be the exact numeric value for the device. For example, the numeric value for `VESA1024x768X` is 0x1a33 (6707), which is the value that must be in the configuration file. If the value in the configuration file doesn't match a value supported by the graphics kernel, or if the configuration file is missing, then the application will not run. TWS itself has no intrinsic defaults for either the mouse or graphics device. The application must supply this information.

*This is a **very** important point for TWS programmers. There are a large number of graphics hardware devices and all professional graphics kernel systems, like MetaWINDOW, support a large number of them. However, in almost all cases it's impossible for the software to automatically determine which graphics hardware (Diamond Stealth? ATI Wonder?) is installed. Therefore, a graphics application should **always** provide some method for the user to specify which hardware is in use, and what video mode to run. This capability is unique to the graphics kernel system and is **not** included in TWS itself.*

Once system initialization has been done, the application can create TWS data items but still can't display anything on the screen. The video will be in graphics mode and the mouse will be started. The application could start building menus, windows, etc., but nothing can be displayed until the application itself has been initialized.

28.2.3. Application initialization

Before any windows can be displayed, the application initialization function **SM_OpenApplication** must be called. This establishes the workspace window that all other TWS windows must be drawn in; sets the application title bar and menu, if any; and sets the window stack pointers appropriately. The workspace is drawn.
Now the application can start displaying its windows and graphics.

28.2.4. Event loop

The heart of any TWS application is the event loop. "Loops" I should say, since many complex applications will have event loops in various parts of the program, depending on what's going on. Since the concept of event handling is so fundamental to TWS programming, we ought to take some time to discuss it in detail.

28.2.4.1. Event-driven design

Most beginning students of computer programming first learn to write "process-driven" programs. A process-driven program performs its work sequentially, requesting input from the user when the program is ready to accept it, and ignoring the user at all other times. If the user does something unexpected the program either stops or just waits until the user does the right thing.
Event-driven programs work differently. At any given time, an event-driven program may allow any number of different actions by the user. In general the program doesn't care which of these actions is taken or in which order they occur, and must be ready to do something reasonable no matter what course the user chooses. Instead of spending most of its time processing and very little time listening for user input, an event-driven program's priority is to pay attention and respond to user actions.
The advantages of event-driven programs from the user perspective are many and well documented. Programmers often find event-driven programming difficult and confusing at first, especially if they have to develop their own event handling systems. The advantage of a library

like TWS is that most of the grunt work has been done. However, you must still design your application and decide how it will react to every possible user input event.

28.2.4.2. What is an event?

From the above discussion it might be assumed “event” is synonymous with “keystroke”, or perhaps “mouse-button-press-or-keystroke.” In fact there’s no fixed definition for an event. In general, an event is an occurrance that a software system should notice. This concept of events lets us define specific event instances in the context of the application. Events can and often are keystrokes and mouse button presses, but they can also be the arrival of a specific time and date, the opening of a specific file, the amount of free RAM reaching a certain level, or anything else that makes sense for the application at hand.

One characteristic that all events share is that they may occur at any time, asynchronously and unpredictably[15]. The goal of event-driven program design is to be prepared to do something reasonable for every possible event.

Since TWS is a user interface library, it responds to keyboard and mouse events, which the user can cause, and timer events, which the user can’t cause but are nevertheless important for building a user interface. It also defines other events, like when a window is moved or the active window changes, that are called *window manager events* elsewhere in this manual.

28.2.4.3. Event handling and callback functions

The task of doing something in response to an event is termed “handling” the event, and the code that does so is called an *event handler*. TWS provides a large measure of default handling, and also gives an application the opportunity to establish its own event handlers. Sometimes the mechanism that lets the application set its own handlers is called a *hook*.

There is more than one way to accomodate application vs. system event handling. In TWS, different types of events are accomodated differently. “Hardware” events, like keystrokes and mouse buttons, are not usually handled directly at the application level. For example, you don’t have to watch for all mouse button events, see where the button was pressed, etc., in order to make gadgets work. TWS does all of that internally. In fact, if a user interface gadget *does* handle a hardware event, the rest of the application will never see it.

Applications handle events through the *callback function* mechanism. These are the application routines that are attached to windows and user interface gadgets. When a user activates a gadget by clicking on it or whatever, the callback function is automatically invoked.

28.2.4.4. Events vs. messages

A *message* is a notification from one part of the system to some other part(s) of the system that something has occurred. The differences between messages and events are that: 1) messages can be generated from any program level, including application code; 2) there is not a fixed set of messages for all applications; 3) there is no direct mechanism for “handling” messages, nor is there any default action in response to specific messages; 4) TWS itself ignores messages completely.

The TWS function **SM_ProcessEvent** returns a message. A message is simply an integer value that has been given some meaning. Some meanings are defined by TWS, but an application may define its own.

The *message* returned by **SM_ProcessEvent** depends on the *event* it processed. For example, most of the TWS example programs look for a message of KEYPRESS, and exit if that message is received. A message of KEYPRESS means that the event passed to **SM_ProcessEvent** was a keystroke that was not processed by any gadget or window. An application can expand on this with its own messages, which are passed back from application callback procedures through the event loop. This concept is described in more detail later.

[15] Yes, even the specific-day-and-time is an unpredictable event. Whose to say that time will ever come?

Quitting the Application

A TWS program *must* call **SM_Exit** before exiting. This function encapsulates the graphics kernel shutdown routines, as well as TWS-specific termination routines. While there are very few guarantees in life, I can guarantee that if a TWS program ends without calling **SM_Exit**, the computer will have to be rebooted!

28.3. Adding an Application Menu

Since almost all applications will use menus the next thing we'll do is add a menu to the program (file ***wintest3.c***).
This program is almost identical to the first one. To add menus to our program we make the following modifications:

- #include the file *smmenu.h*
- For each different menu and submenu in the application declare a *MenuType* * variable;
- Build each menu by adding items to it;
- Attach the top-level menus to their windows when the windows are created.

A menu is created by the **SM_CreateMenu** function. Every menu, including submenus, must be created before items are added to it. Then **SM_AddMenuItem** is called for each item in the menu:

```
menu = SM_InitMenu();
SM_AddMenuItem(menu, SUBMENU_ITEM, "File", NULL, NULL);
SM_AddMenuItem(menu, ACTION_ITEM, "Quit", NULL, NULL);
```

Note that both of the menu items attached to *menu* have neither action functions nor submenus. You would be unlikely to do this in your own programs, but it works for educational purposes (or to "stub" your menus before you've written the menu functions). Items are added to menus left to right or top to bottom based on the order they're added to the menu.
How do we know whether a menu will be a horizontal bar or a pull-down? The menu that is directly attached to the window is always a horizontal bar menu. Any submenu of that menu (and all subsequent submenus) are pull-downs:

```
SM_OpenApplication("TWS Windows", menu); /* Create application screen    */
```

Since *menu* is directly attached to the application workspace it will be a horizontal bar menu.
If you move the cursor over one of the menu labels and press the left mouse button, you'll see a rectangular depression over the selected menu item. Nothing else will happen. Technically it is a menu, though it's a pretty boring example.
Time to put a window on the screen.

28.4. Windows At Last!

In the next example we'll finally put something on the screen that looks like a real windowing GUI. You'll be surprised how little extra work we have to do (see file ***wintest4.c***).
The additions necessary to get a simple (and empty) window on the screen are:

- Add a *WindowType* * variable;
- Specify the size and position of the window's content region by building a rectangle with the appropriate dimensions. That means we also need a *RectType* variable;
- Create the window by calling **SM_NewWindow**, assigning the returned window pointer to our window variable;
- Open the window on the screen with **SM_OpenWindow**.

Besides adding the appropriate variable declarations, here's all the extra code necessary:

```
r.Xmin = r.Ymin = 100;
r.Xmax = r.Ymax = 300;
w = SM_NewWindow(&r, "Test Window", DOCUMENT | NOBACKING, NULL, NULL);
SM_OpenWindow(w);
```

These statements are added after we've opened the application and before the event loop starts. The meaning of the various parameters and numbers, if not obvious, are explained in the previous Windows reference section.
So now we've got a window – so what? What can we do with it? We can move it around on the screen, enlarge it to its maximum size and restore it (but not minimize it, since we didn't create an icon for the window), resize it by dragging the borders, and close it. All of which is happening inside the event loop we added in the second example – the event processor finally has something to keep it busy!

28.5. So Far...

Let's stop here and summarize. We've now seen the rudiments of a basic TWS window application. Here are the steps:

- Be sure the #include the necessary headers. All programs need *smwindow.h, smtypes.h,* and *smevent.h* as a bare minimum;
- Initialize the TWS system;
- Create the application menu. You may also create other menus for use by other program windows, which we'll do in a later example;
- Open the application;
- Create and open application windows as necessary;
- Enter the event loop;
- Be sure there's a way to call **SM_Exit** to exit the program.

Of course, there are still a lot of things we'd like to do that we haven't seen yet. We haven't built any gadgets or done any graphics; we haven't built any submenus or added any functionality to the window menu; we haven't added any window callback functions or background procedures; shoot, we haven't done any *application code* yet! All of which we'll get to – this just looked like a good milestone.

28.6. Simple Drawing

In this example we'll write a simple drawing program. Again, this is about the simplest program that could possibly be called a drawing program, but nonetheless it does draw lines in a window. The code is in the file ***windraw.c***.
The main new thing here is the addition of a window callback function. This is a function 'attached' to an application window that is called repeatedly by the event processor whenever the window has the focus. The window callback function is attached to the window using the **SM_SetWindowProc** call:

```
SM_SetWindowProc(w, TWSDraw);
```

Since this is our first window function it's reproduced in its entirety below:

```
int TWSDraw(WindowType *w, EventType *ev)
{
    static int  oldx = -1, oldy = -1;
    int         x, y, nx, ny;
    int         state;
    /*
    ** We know if we get here that the window is the focus window, but we
    ** don't know if the event occurred in this window.  We can check it...
    */
    if ((ev->Type & BUTTONPRESS) && (ev->Region == CANVAS_REGION)) {
        /*
        ** While we're drawing we want to XOR the line so it can appear to
        ** move along with the cursor
        */
        GR_SetDrawMode(w, SMXOR);

        GR_GetMouse(w, &state, &x, &y);
        /*
        ** If the old point is negative then we haven't been here before
        ** so set the anchor point to the current point
        */
        if (oldx == -1) {
            oldx = x;
            oldy = y;
        }

        GR_DrawLine(w, oldx, oldy, x, y);

        /*
        ** While the mouse button is down we want to trap it and draw a
        ** rubber-band line to follow it around
        */
        while (state & LEFTBUTTONACTIVE) {
            /*
            ** If the new cursor position is different from the previous
            ** position then erase the previous line and draw a new one
            */
            GR_GetMouse(w, &state, &nx, &ny);
            if ((nx != x) || (ny != y)) {
                GR_DrawLine(w, oldx, oldy, x, y);   /* Erase old line       */
                GR_DrawLine(w, oldx, oldy, nx, ny); /* Draw new line        */
                x = nx;                 /* Save new cursor position         */
                y = ny;
            }
        }

        /*
        ** At this point the left mouse button has been released.
        ** Set the line mode to overwrite and draw the line again
        */
        GR_SetDrawMode(w, SMREP);
        GR_DrawLine(w, oldx, oldy, x, y);
        oldx = x;
        oldy = y;
    }
    return True;
}
```

A window callback function must have two arguments – a window pointer and an event pointer. When called the window pointer will point to the function's parent window, and the event pointer will point to the system event that just occurred.

If a window is active its window callback function is called on every pass through the event loop. Often a window function is only interested in certain things, such as a particular type of event. The first thing it should do therefore is see if an interesting event has occurred and, if not, return immediately. TWSDraw does that in the first if-statement.

Once we're assured that an event we're interested in has occurred, we can process the event. In this case, we're looking at the left mouse button state. If it's pressed, we're going to draw a 'rubber-band' line from the endpoint of the last line drawn to the tip of the cursor. When the left mouse button is released we'll draw the line in place and update the new endpoint. The result is a series of connected line segments in the window.

There's nothing particularly remarkable about the code itself but a few comments are in order on the function's organization. Note the 'static' keyword for the *oldx* and *oldy* variables. It's not unusual for a window function to use lots of static variables since, for user response reasons, these functions seldom complete their work in a single call. Instead, window functions (like background procedures) usually chop a task into bite-size chunks. After each chunk the function 'remembers' where it is and returns so the event manager can see if the user has done anything. On the next pass through the event processor, the window function continues where it left off. We use this piecemeal approach to minimize the lag time between the user doing something and the system responding to it.

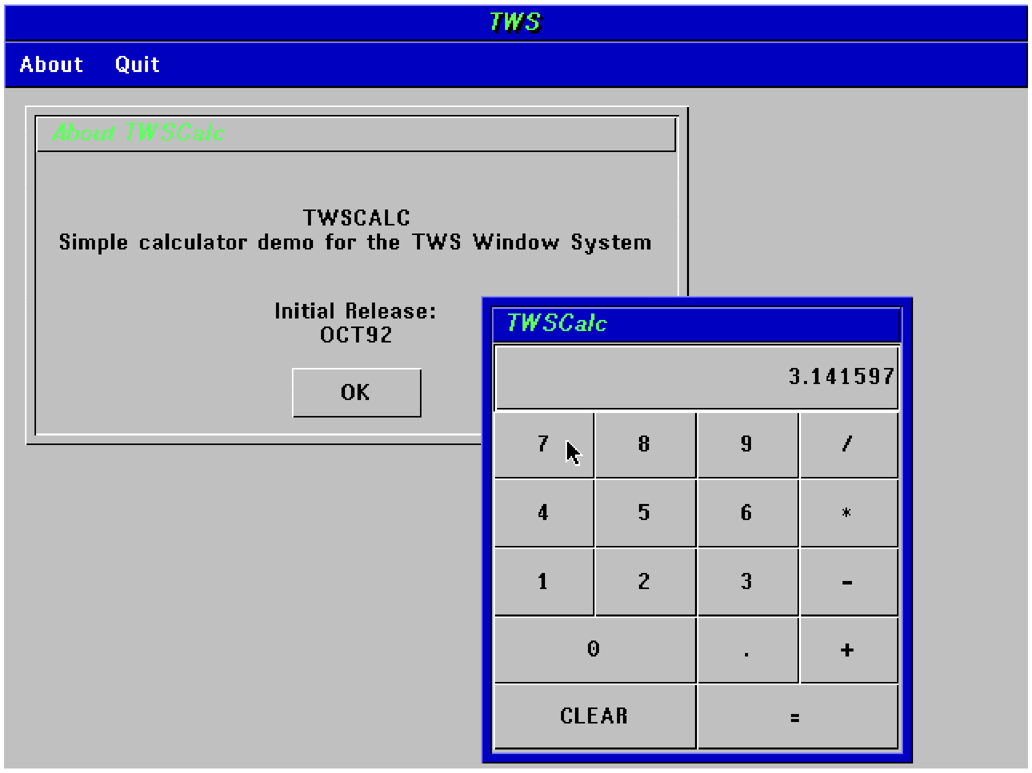
We used the **SM_GetMouse** function for the inner loop. Why not put an event loop here? We could have. In this case we are not interested in events in general, only in the mouse state, so it's more efficient to query the mouse directly.

However, if we had a background procedure installed, we might have used an event loop here, probably with an event grab. That would have allowed the background procedure to continue to execute while lines were being drawn, but would have created a bit of a lag in the drawing itself. The amount of lag would depend on what the background process was doing.

The function returns True to indicate that it fully processed the event passed to it. This value becomes the return value for **SM_ProcessEvent** and will be passed back to the event loop in main. If we wanted to we could send simple 'messages' back to the main event loop this way.

Finally, try moving or resizing the window after drawing a few lines. The lines disappear! That's because we didn't provide any way for the lines to be redrawn whenever the window has to be redrawn – all TWS knows to do is clear the window. Also, when the window is enlarged, the line drawing is not clipped at the window border, but within a smaller rectangle. That's because the graphics state canvas is *not* resized along with the window borders. Again, we could write code to make this happen, if we wanted to. Maybe later...

28.7. A Gadget Example: TWSCalc



TWSCALC
Simple calculator demo for the TWS Window System

Initial Release:
OCT92

In addition to window functions, gadgets are the most common method of controlling an application. The TWSCalc application demonstrates the use of gadgets. The file ***calc.c*** implements a simple 4-function calculator built totally of buttons and labels.
The first interesting thing in this program is the function **BuildButtons**. Notice that all the buttons are built using *virtual* coordinates. If you want the calculator to be larger or smaller, all you have to do is drag one of the resize handles. The calculator buttons will resize themselves to fit the new window.
Next, notice that all the numeral buttons have the same callback function and all the operand buttons have the same callback function. There's no rule that says each gadget has to have a *unique* function.
Finally, notice that all the buttons have the calculator display label stored in their user data fields. Why? Let's jump down to the **EnterNumber** function, which is the user function attached to the numeral buttons. It's short so we'll reproduce it here:

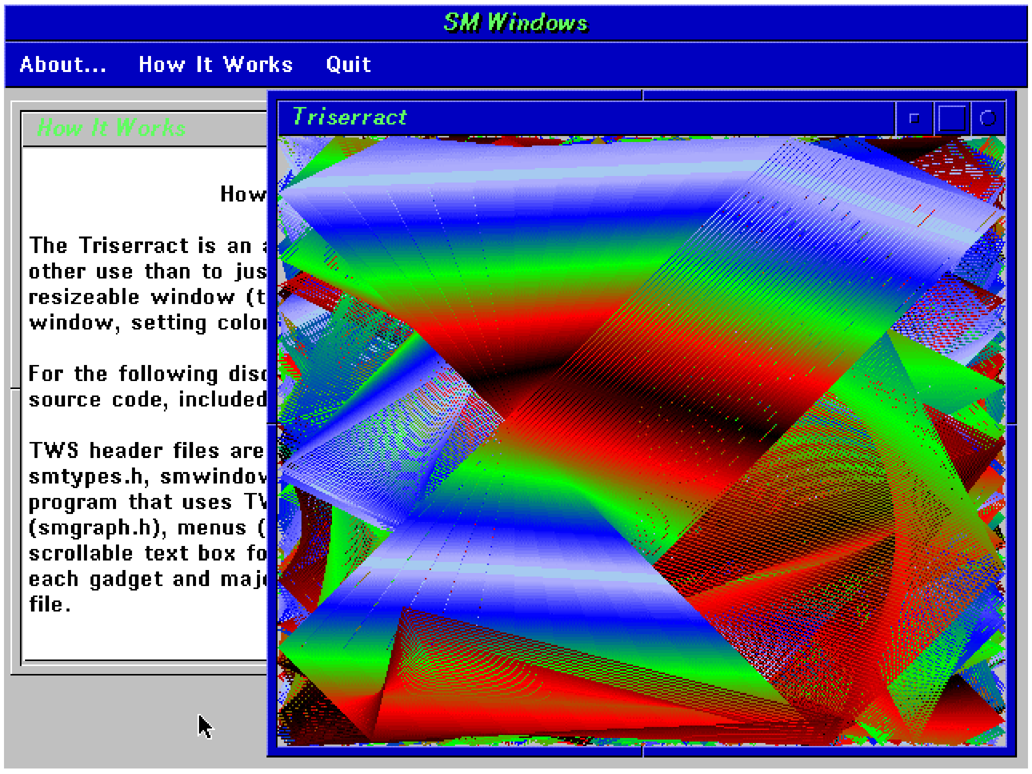
```
int EnterNumber(ButtonType *b)
{
    int     i;
    char    *s;
    LabelType   *l;

    /*
    ** The button label is the numeral
    */
    s = SM_GetButtonLabel(b);
    currinput[strlen(currinput)] = *s;

    /*
    ** Retrieve the display label, attached to the button data field,
    ** and change the label string. This automatically redraws the label
    ** because in the program the calculator window is always focus
    */
    l = (LabelType *)SM_GetButtonData(b);
    SM_SetLabelString(l, currinput);
    return True;
}
```

The argument to a button callback function is a pointer to a button. By storing a pointer to the label in the button's user data field, we can retrieve the label, change the string and redisplay it every time a button is pressed.
The other buttons all work about the same way. The actual details of how the calculator itself works aren't very interesting. The point here is to see TWS in action.
Finally, take a look at the **About** function at the end of the file. This is an example of a simple dialog window for displaying program messages, warnings, and errors. Note that, again, I've used virtual coordinates for the labels, and the strings are centered. The "OK" button closes the window. This isn't a modal dialog because you can click in the *TWSCalc* window and activate it while the *About TWSCalc* window is displayed – you can even open multiple copies of the *About...* window!

28.8. Hands-off Graphics: TRISERACT



How

The Triserract is an
other use than to jus
resizeable window (t
window, setting colo

For the following disc
source code, included

TWS header files are
smtypes.h, smwindov
program that uses TV
(smgraph.h), menus (
scrollable text box fo
each gadget and maj
file.

The final example, in ***triserac.c***, is the Triseract program. This program demonstrates several TWS features: color palette control, background processing, the text gadget, and window resize and close procedures.

A *triseract* is a nonsensical word for a triangle that deforms itself in size, shape and position over time. It's a simple and colorful (on a 256-color system) pattern that I sometimes crank up as a screen saver. It also has two other text windows: one gives a brief description of the program (how it works), the other a brief overview of the TWS system and how to register.

After setting up the window system and building the application menus, the program sets up the color palette for the triseract window. The palette setup will be different if the current graphics mode supports 256 colors or 16 colors. The color assignments are arranged so that a 'rainbow' of color is set up.

The TRISERACT program uses a background procedure for most of the work. A background procedure is similar to a window callback function, except that a window callback is only called if the parent window has the focus. A background procedure is called even if its parent window isn't the focus window -- even if the parent window isn't open!

Before we can use background procedures we have to initialize the background processing system:

```
SM_InitBackgroundProcs();
```

Next the **Lines** function is registered as a background procedure:

```
id1 = SM_RegisterBackgroundProc(w, Lines, ALLEVENTS | NOTOBSCURED);
```

The flag ALLEVENTS | NOTOBSCURED tells the background procedure that the function Lines should be called for every event, but only if its parent window *w* is not obscured by any other window. That *doesn't* mean the window has to have the focus, only that no other window, focus or not, is in front of it.

Next special resize and close procedures are attached to the triseract window:

```
SM_SetResizeProc(w, ResizeWin);
SM_SetCloseProc(w, CloseWin);
```

Why does the Triseract window need special attention? When the window is resized, we want the window's graphics canvas to fill the window. Remember that by default the graphics canvas is fixed in size. The **ResizeWin** function sets the graphics canvas to the same size as the window's content.

CloseWin just removes the background procedure function. If we didn't do that when the window is closed the background function would certainly barf, since its parent window would no longer exist! This is a rule you should always remember: ***background functions must <u>always</u> be removed when their parent windows are closed!*** We could call this "The First Law of Background Procedures".

The **Lines** function just draws a triseract triangle. Note: triangle *singular*. Each call to **Lines** only draws a single triangle. This is "The First Law of Event-Driven Programming": whatever you have to do, keep it short and get back to the event loop fast! Other than this important observation there's nothing remarkable about this function.

The **About** and **HowItWorks** functions are not too remarkable either. Both open separate windows and fill them with text (in the form of text gadgets). Again, we have resize procedures so that the text bounds are kept the same size, more or less, as the window content.

28.9. Additional Examples

The few preceding examples have been developed to show off the basics of programming using the TWS system. The TWS distribution contains a number of additional examples you'll want to look at.

Index