Shipping Your Application

Going without the SUI file

There comes a time in all software where the application needs to go to the end user. At such a time, it is likely that the interface is frozen and thus, you do not want users to be able to access the property editor or to be able to move and resize widgets interactively. When the flexibility of the runtime tools are no longer required or desired, it is possible to "internalize" all of the interface information in a .sui file by following these steps:

1.) Change the call in your application from **`SUIT_init()`** to **`SUIT_initFromCode()`**

2.) Rename your .sui file to suitinit.c

3.) Compile and link in suitinit.c

This has the effect of "shrinkwrapping" the application, in the sense that all the interactive tools that make up SUIT are turned off.

Restoring the Interactive Tools

In the event that the interactive tools are required again, you can reverese the above process:

1.) Change the call in your application from **`SUIT_initFromCode()`** to **`SUIT_init()`**

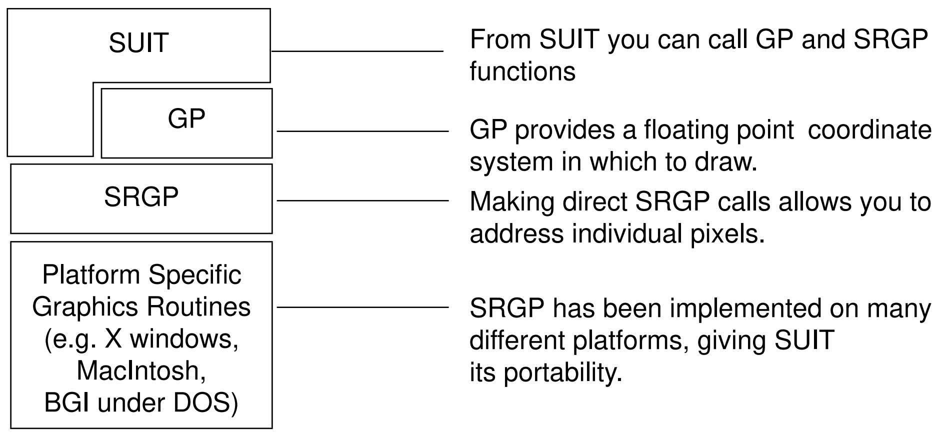
2.) Rename your suitinit.c file to <program name>.sui

3.) Compile and link without suitinit.c

The SUIT Software Layers

Drawing In Pixels: SRGP

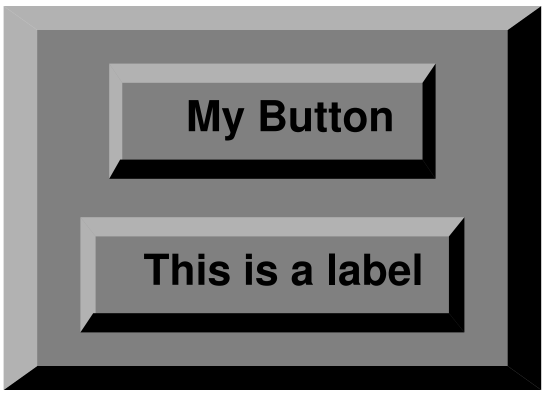
In the section describing Windows and Viewports, we make a case against using straight pixel coordinates for widget graphics. Clearly, though, there are occasions where being able to address the screen at the pixel level is appropriate. To do this, you can make function calls directly to SUIT's underlying raster graphics facility, the Simple Raster Graphics Package, SRGP. GP is a "thin veneer" over SRGP that performs the necessary conversion between the floating point numbers that the programmer is using to the integer pixel coordinates that SRGP uses. Below is a diagram that shows the different layers of graphics software that SUIT uses.



Documentation for the SRGP layer is available via anonymous ftp from Brown University (ftp address wilma.cs.brown.edu).

Shrink To Fit

The boolean property SHRINK_TO_FIT that governs labels, buttons and type in boxes causes a widget's viewport to shrink around the size of the text in the LABEL property, plus whatever MARGIN there is. It is not possible to interactively resize a widget that obeys the SHRINK_TO_FIT property. The viewport of the widget shrinks about its center point, so the four sides of the widget move, but the center of the widget remains unchanged. In our example, if SHRINK_TO_FIT is set to true for both the button and label, these widgets would assume their set locations inside the bulletin board and then would resize about their respective centers to create a widget that looked like the one shown below. (The label widget has had its HAS_BORDER property changed to TRUE for purposes of illustration).



Here, the SHRINK_TO_FIT property is toggled to TRUE for both the label and the button, making their viewports as small as possible, while still being able to see the text.

Notice that their viewports have shrunk about the center of the widget.

Stacker Widgets

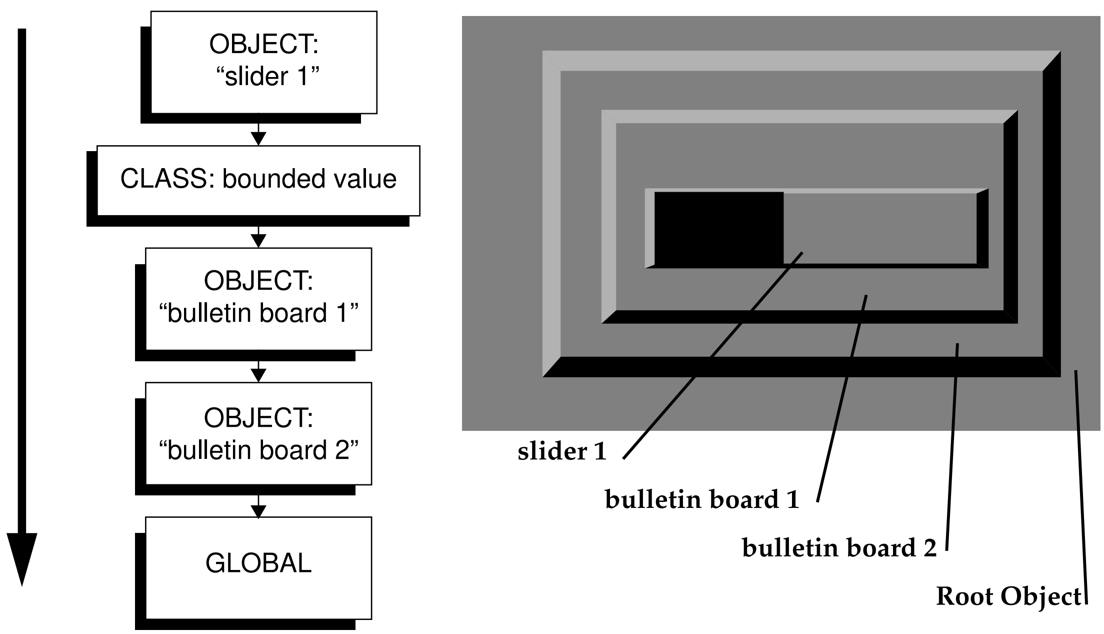
There is a special hierarchical widget called a "stacker" that comes in two display styles, horizontal and vertical. These are essentially bulletin boards that perform some very simple geometry management by arranging their children to be packed vertically or horizontally. They are like all hierarchical widgets with respect to property lookups and children.

Employees

Employees are children that are registered for a particular display style. A good example of the use of employees is the bounded value widget. The scrollbar display style uses an elevator widget and two arrow button widgets. Clearly, these widgets need not exist for any other display style other than the scroll bar, so instead of children, we make these widgets employees of the scroll bar display style. For purposes of property lookup and composition, employees are treated exactly like children.

Property Lookup Order

Property lookup on a hierarchical widget is done in a slightly different order from the lookup on "flat" objects. The lookup begins the same: it starts at the OBJECT level of the widget in question and if the property lookup fails, it then goes to the appropriate CLASS level. Failure to find a property at the CLASS level would usually mean looking for a property at the GLOBAL level, but with a hierarchical widget, property lookup instead continues at ***the object level of the parent***. If you still fail to find the property, the lookup continues at the OBJECT level of the next immediate parent (its grandparent), until you reach the root object, where a lookup at the GLOBAL level takes place. For example, imagine an object called "slider 1" which is a child of "bulletin board 1" which in turn is a child of "bulletin board 2" which is a child of the root object. The property lookup would go like this:



Registering a Bulletin Board as a New Class

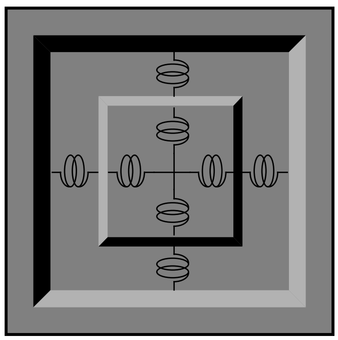
Bulletin boards belong to the "bulletin board" class of widgets. If you set a property at the "bulletin board" class, all bulletin boards will inherit that property, if they don't specify that property at the OBJECT level. This can present a problem. Because bulletin board widgets are used to create new widgets, it is often useful to associate a new class name with a widget created out of a bulletin board. For example, suppose you had created a hierarchical widget using a bulletin board: a data entry form with several type in boxes and an OK button. From your point of view, you've created a completely different kind, or class, of widget: the "data entry form" class. To SUIT, though, it's just a bulletin board. Properties set at the "bulletin board" class (such as foreground color) would affect bulletin boards AND your data entry form. To allow you to set different class level properties on these two kinds of widgets, you need a way of telling SUIT that the bulletin board you are creating is not really a bulletin board for the purposes of class level property lookups, but is some other class instead. The SUIT call that does this is:

```
form=SUIT_createBulletinBoardWithClass("form1","data entry form");
```

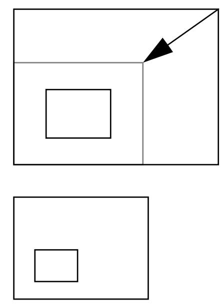
Where "data entry form" is the new class name you want to use for property lookups.

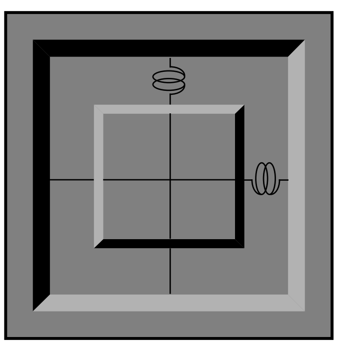
Springiness

Suppose you resize our example bulletin board to be smaller. Should the button and label get proportionally smaller or should they stay the same size as they were before the resize? The answer lies in the *springiness* of the label and the button, a property that allows widgets to specify how to resize if their parent changes in size. The metaphor we use here is one of *springs* and *stiff bars*: imagine that each widget is connected to the four walls of its parent by either springs or stiff bars. **The rule to remember is:** As the parent's walls move, the widget will stay the same distance from a wall if it is connected by a stiff bar. In the same fashion, widgets describe whether they are allowed to resize by having either a stiff bar or a spring in each of the two possible directions, horizontal and vertical.

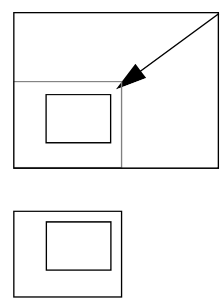


This is the default value for springiness. With no stiff bars, the child resizes proportionally as its parent resizes.





Here, springiness is set so that the child will always remain the same distance from the lower left hand corner of the parent, but the distance to the upper right hand corner is allowed to float. Because there are bars on the inside of the widget, both horizontally and vertically, the widget will stay the same size, regardless of how its parent resizes.



Though it may not be obvious at first, it turns out that there must be *some* springiness vertically and horizontally. If a widget was connected to its parents by a stiff bar on both its left and right sides, **and** had a stiff horizontal bar inside, resizing the parent would require following conflicting orders: the bars on the outside tell the child's left and right sides to stay the same distance from the parent's sides, but the stiff bar inside tells the child widget not to get wider. You might say that the child widget is "overconstrained." SUIT's springiness widget (above) does not allow you to select these illegal springiness states.

Changing Childrens' Viewports

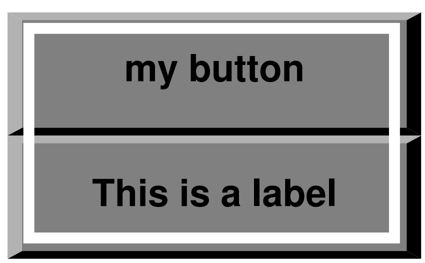
There are two ways of arranging children inside a parent widget:

- Interactively arrange the children using SUIT-move
- Explicitly arrange the children in C code by setting viewport properties.

Of course, you can't just use SUIT-move to move a bulletin board's children, the SUIT-move moves the bulletin board as a whole. In order to "get inside" a hierarchical widget, you need to open it up with SUIT-o. When opened, a red border will appear around the bulletin board so that the children can be accessed with SUIT commands like SUIT-click SUIT-e, etc. You can even add and remove children from a bulletin board by interactively dragging the objects into and out of the bulletin board.

The second way of laying out the bulletin board's children is by changing their viewports with calls to **`SUIT_setViewport()`**. This method is more precise, but somewhat more time consuming. The following code places a button and a label inside the parent widget:

```
SUIT_changeObjectSize(board, 400, 200);

SUIT_setViewport (label, VIEWPORT,
                    SUIT_defViewport(0, 0, 400, 100));
SUIT_setViewport(button, VIEWPORT,
                    SUIT_defViewport(0,100, 400, 200));.
```



Viewport of button (0,100, 400, 200)

Viewport of label (0, 0, 400, 100)

When you type SUIT-o,a border appears around the widget, letting you know that the widget has been "opened" and that the widgets inside can now be moved interactively.

NOTE: It is important that these calls come AFTER the call to **`SUIT_addChildToObject()`**. Because viewports are relative to the parent, if you set the viewport of the button before adding it to the bulletin board, it will compute its location relative to the root object, not the bulletin board.

Setting Viewports Using GP Coordinates

`SUIT_mapToParent()` is a special mapping function that allows you to use GP coordinates to position a child widgets inside its parent. The function takes an object and four floating point numbers that represent the corners of a GP_rectangle and returns the unique SUIT_viewport inside the parent that corresponds to the given GP_coordinates. For example, to place the above label inside the bulletin board as shown, you would use this code:

```
SUIT_addChildToObject (board, label);
SUIT_setViewport(label, VIEWPORT,
                    SUIT_mapToParent(label, 0,0, 0,0, 1.0, 0.5));
```

This mapping function makes it possible to place and size widgets using the same coordinate system used by the GP graphics calls to draw lines and such. You can see more examples of this mapping call in the example programs that come with the SUIT distribution.

Bulletin Boards

Bulletin boards are a class of widget that can hold other widgets. Bulletin boards can even contain other bulletin boards, allowing you to create very complex widgets. When clicked on, a bulletin board will pass the event down to its children and when moved, the bulletin board will make sure that its children are moved as well. Composing widgets with bulletin boards is easy: for example, to create a bulletin board that contains a button, you use the following code:

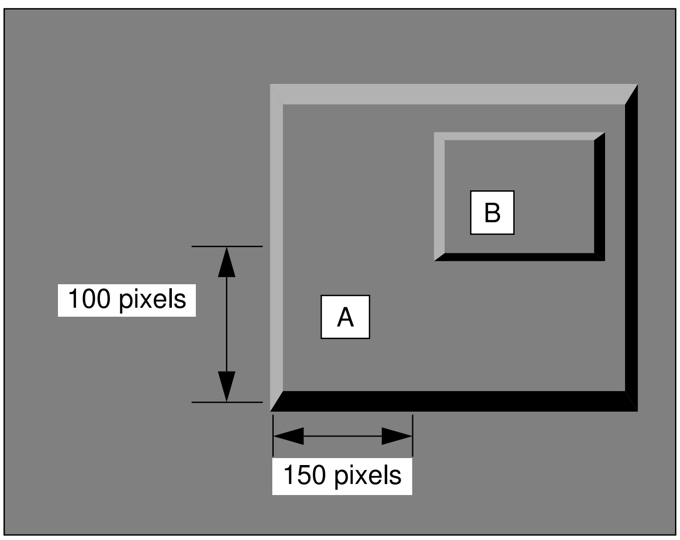
```
board = SUIT_createBulletinBoard("sample");
button = SUIT_createButton("my button", myCallbackFcn);
SUIT_addChildToObject (board, button);
```

Examples of bulletin board creation can be found in the collection of example files that comes with the SUIT installation. The composition calls that you see here do not specify *where* the label and the button will appear in the bulletin board, all they do is make sure that the button and label are treated as children of the bulletin board. To place the label and button in a particular position inside the bulletin board, you need to set the VIEWPORT property of the children.

Viewports

Viewports for hierarchical widgets are based relative to the lower left hand corner of the *parent widget*, not the lower left hand corner of the screen.

A is a hierarchical widget with one child, a button B.



Widget B's viewport is measured from the lower left hand corner of widget A, not the lower left hand corner of the application window.

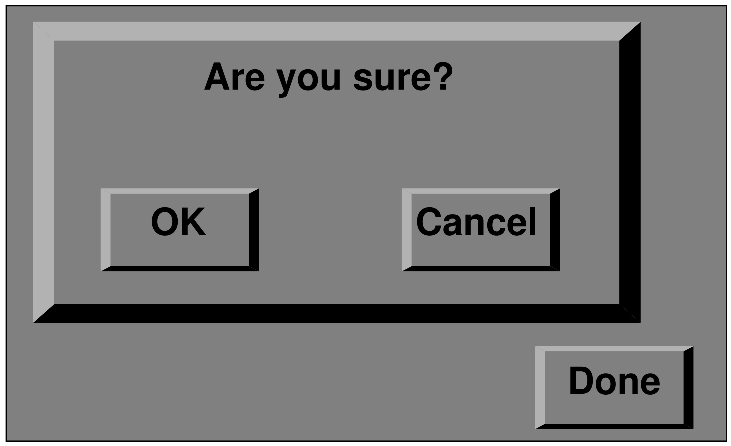
If B is 50x50 pixels, then B's viewport is
lower left = (150, 100)
upper right = (200, 150).

Widget A's viewport is measured from the lower left hand corner of the application window.

Hierarchy

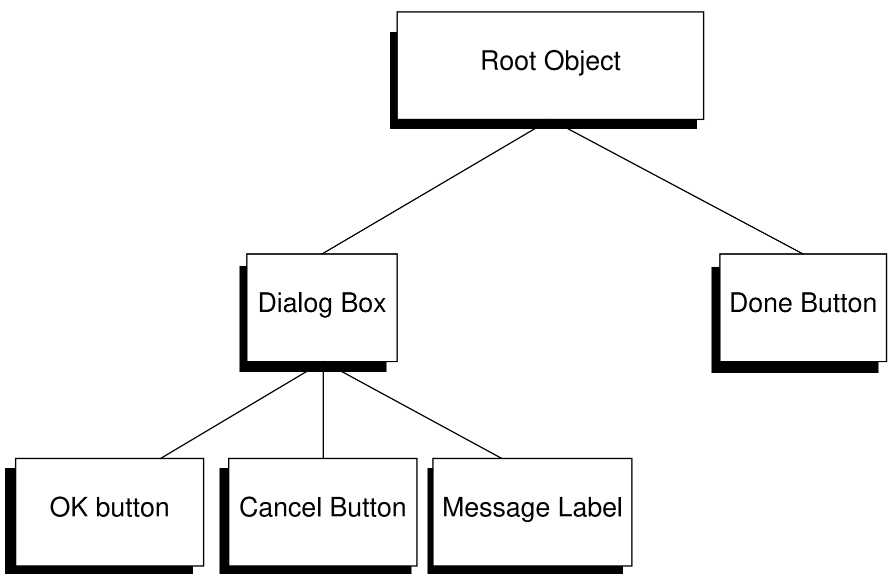
Introduction: Parents and Children

SUIT objects can be composed of other SUIT objects for purposes of creating more complicated widgets such as dialog boxes:



In the above example, we have a dialog box widget that contains a label widget and two button widgets. We say that the label and two buttons are the *children* of the dialog box and that the dialog box is the *parent* of the label and two buttons.

All objects are ultimately the children of an invisible *root object.* This relationship we can draw as a tree:



In this section, we will discuss how hierarchical widgets are different than the simple "flat" widgets that you might be used to, and what you need to do to compose a hierarchical widget of your own. We will start by looking at the basic building block of hierarchial widgets: the bulletin board widget.

3.) SUIT_checkAndProcessInput (timeToWait)

1.) REDISPLAY WIDGETS AS NEEDED: This step performs a redisplay of all widgets that have been flagged as requiring redisplay. Things that cause this flag to be set for a widget are:

- Changing the value of a property for that widget (See "When Do I Paint?", below). SUIT is careful not to flag the widget as needing a repaint if the result of setting a property does not in fact change the value.
  For example, if the CURRENT_VALUE of a bounded value is 3.141, calling
  `SUIT_setDouble (obj, CURRENT_VALUE, 1.0)`
  would flag the widget as needing a repaint, but
  `SUIT_setDouble (obj, CURRENT_VALUE, 3.141)`
  would not.
- Bringing the widget to the front of the stack of widgets
- Moving or resizing a widget
- Animating a widget

2.) GET EVENT: This step determines the type of the event that has occurred. Events are queued in a first-in, first- out manner. The timeToWait parameter determines the length of time in "ticks" (1/60 second) this function will wait for an event to enter the queue. If, after that period of time, no event has entered the queue, **`SUIT_checkAndProcessInput()`** exits. The one exception to this is if the **`timeToWait`** parameter is set to INDEFINITE, in which case, the function sleeps, waiting for an event. When an event finally does occur, the function exits.

3.) IS IT A SUIT EVENT? If the event happened while the SHIFT and CONTROL keys were pressed, the event is interpreted as a SUIT command and is processed as such (move, resize, open, etc.).

4.) HANDLE TRAPPER FUNCTION: If no trapper function has been registered, proceed to step 5, otherwise call the trapper function. Trapper functions designed to catch all input events before they are handed off to whatever widget was hit. Typical applications of trapper functions include adding hot keys to a menu and allowing the return key to activate a dialog box "OK" button, even if the cursor is not over the OK button. Trappers are discussed in detail on page 70. If the trapper returns NULL this means that the trapper has consumed and handled the event; we abort the loop and begin again. Otherwise, the trapper will return with a SUIT_object, though not necessarily the SUIT_object that was hit. This SUIT_object returned is handed over to step 5.

5.) HIT WIDGET: The hit procedure for the widget is called.

When Do I Paint?

1.) Painting a widget happens only when the widget has been flagged as requiring a repaint. Calling **`SUIT_redisplayRequired(obj)`** does not actually repaint the object, it merely flags the object as needing a repaint. The repainting happens at the beginning of each iteration of the SUIT main loop. If there is more than one widget that requires repainting (as there often is), the widgets are drawn from back to front in the Z-ordering of widgets.

2.) If you use any of the SUIT_set functions to change the value of a widget's property, SUIT will flag that widget as needing a repaint. Because of this "courtesy" side-effect, **it is vital that you not call any of the SUIT_setProperty functions from inside a widget's painting procedure.** Doing so will cause a widget to repaint multiple (perhaps endless) times because after each invocation of the paint procedure the widget will be flagged as needing redisplay, which will cause yet another call of the paint procedure.

The SUIT Main Loop

Every SUIT program will need to have the following calls in order to initialize SUIT and to get the SUIT main loop running. The lines are numbered and explained in the sections below:

A typical **main()** would look like:

```
    void main (int argc, char *argv[]) {
(1)     SUIT_init (argv[0]);
(2)     SUIT_beginDisplay();
            while (TRUE) {
(3)         SUIT_checkAndProcessInput(INDEFINITE)
        }
    }
```

Note that instead of the **while()** loop and a call to **SUIT_checkAndProcessInput()**, you can use a single call to **SUIT_beginStandardApplication()**, which is just a sugar coating for the above while loop. Below is explained the things that go on at each stage of the loop.

1.) SUIT_init (argv[0])

1.) Initializes all global variables that SUIT needs, including the name of the application.

2.) Registers all of SUIT's supported types (see page 35 for a discussion of types).

3.) Initializes all of the SUIT global properties (see page 110 for a list of global properties).

4.) Peeks at the .sui file to establish the correct screen size and window size.

5.) Begins the underlying raster graphics package, SRGP.

6.) Initializes the cursors, color table and fonts that SUIT uses.

7.) Initializes all the built-in widget classes (see page 108 for a list of SUIT widget classes).

2.) SUIT_beginDisplay()

1.) Reads the .sui file. Create any interactively created objects (those not created from the source code).

2.) Clears the screen.

3.) Draws all widgets for the first time.

Appendix

UVA Logo

Class: “uva logo”

Display Styles:

uva big v rotunda

`SUIT_object SUIT_createUVALogo (char *name)`

Description: This widget displays several different logos for the University of Virginia. It is included in the toolkit because we are insanely proud of our institution.

Property Name	Notes
LINE_WIDTH *integer*	Sets the line thickness for the rotunda display style.

INPUT_SEQUENCE
text

This buffer stores the editor commands before the editor processes them. After the editor recognizes a command, this property is reinitialized to he null string. If you register an interest in this property, you can examine the keyboard command before the editor responds to it. See also the SUIT function called **`SUIT_sendToEditor()`**.

MARK_INDEX
integer

This number denotes the beginning position of the highlighted region in the text. To change the position of the highlighted region, use the **`SUIT_highlightBlockInTextEditor()`** call.

MARK_END_INDEX
integer

This number denotes the ending position of the highlighted region in the text. To change the position of the highlighted region, use the **`SUIT_highlightBlockInTextEditor()`** call.

NUMBER_OF_LINES
integer

This represents the total number of text lines being edited. This number will always be at least 1.

READ_ONLY
boolean

When TRUE, commands that change the buffer are disabled and the widget becomes a text browser rather than an editor.

SPACING_GAP
integer

Number of pixels between lines of text measured from the lowest descender to the highest ascenders of the next line.

TAB_LENGTH
integer

The length of a tab, measured in characters. For example, setting tab length to 5 will provide tabs every five spaces across the width of the editor. Note that tabs will only line up with a fixed width font such as courier.Tabs are preserved in the text as tabs, not converted to spaces.

Type In Box

Class:"type in box"

Display Styles:

standard

SUIT_object SUIT_createTypeInBox (char *name, void callback (SUIT_object))

Description: Creates a one line text entry field.

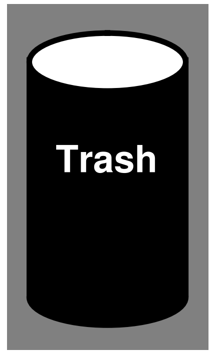
See Also: The keybindings are exactly the same as those for the Text Editor Widget.

Property Name	Notes
ALTERED *boolean*	FALSE until the text is edited.
ANY_KEYSTROKE_TRIGGERS *boolean*	When TRUE, the callback is called at every keystroke. When FALSE, the callback is only invoked when the user presses RETURN.
CURRENT_VALUE *text*	The text currently being edited. This string is allowed to contain newlines, tabs, but cannot use the GP_text special characters; all characters in the text editor will be the same font.
CURSOR_COLOR *GP_color*	The color of the cursor, which defaults to black.
CURSOR_INDEX *integer*	This temporary property denotes which character the cursor is in front of. For example, when CURSOR_INDEX is 0, this means that the cursor is in front of the first character.
CURSOR_STYLE *SUIT_enum*	This denotes the style of the cursor, either "vertical bar" or "i-beam."
CUT_BUFFER *text*	Contains the last text cut from the CURRENT_VALUE string. This buffer is set whenever the WIPE_BLOCK_KEY or KILL_LINE_KEY key sequence is used (CUT_BUFFER is set to be the killed text) or when the user highlights the text by dragging over the text editor with the mouse (CUT_BUFFER is set to be the text in the highlighted region).
HIGHLIGHT_BLOCK *boolean*	When true, the text that appears between the MARK_INDEX and the MARK_END_INDEX appears in reverse video.

Trash Can

Class: "trash can"

Display Styles:



standard

`SUIT_object SUIT_createTrashCan (char *name)`

Description: This widget is nothing more than a fancy placemat.

Property Name	Notes
HELP_MESSAGE *text*	This is the message that appears in a SUIT_inform() style dialog box when the user clicks on the trash can.

Text Editor / Type in Box Keybindings

The notation used here is similar to that used in Emacs: “C-b” is interpreted as “Control-b” and “M-<“is interpreted as “press the escape key, then press the **<** key“. The values given here are default values.

BACKWARD_CHAR_KEY Default value is C-b
text

BEGINNING_OF_LINE_KEY Default value is C-a
text

BEGINNING_OF_TEXT_KEY Default value is M-<
text

DELETE_CHAR_KEY Default value is C-d
text

DELETE_ENTIRE_LINE_KEY Default value is C-u
text

DONE_EDITING_KEY Default value is C-x. This command invokes the callback.
text

END_OF_LINE_KEY Default value is C-e
text

END_OF_TEXT_KEY Default value is M->
text

FORWARD_CHAR_KEY Default value is C-f
text

KILL_LINE_KEY Default value is C-k
text

NEWLINE_KEY Default value is C-m
text

NEXT_LINE_KEY Default value is C-n
text

OPEN_LINE_KEY Default value is C-o. This opens a line at the current cursor position.
text

PREVIOUS_LINE_KEY Default value is C-p
text

REPAINT_KEY Default value is C-l (that’s a lower case letter, not the number 1)
text

SCROLL_DOWN_KEY Default value is M-v
text

SCROLL_UP_KEY Default value is **C-v**
text

SET_MARK_KEY Default value is C-‘ which is equivalent to a C-<space> key sequence.
text

TAB_KEY Default value is C-i
text

WIPE_BLOCK_KEY Default value is C-w
text

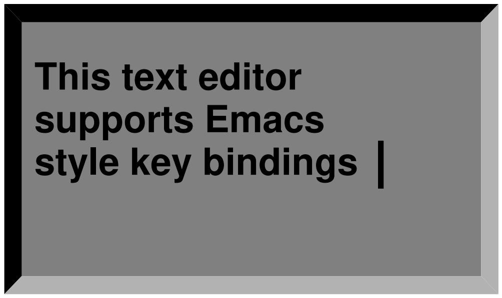
YANK_KEY Default value is C-y
text

Property Name	Notes
ALTERED *boolean*	FALSE until the text is edited.
ANY_KEYSTROKE_TRIGGERS *boolean*	When TRUE, the callback is called at every keystroke. When FALSE, the callback is only invoked at the DONE_EDITING_KEY key sequence
CURRENT_VALUE *text*	The text currently being edited. This string is allowed to contain newlines, tabs, but cannot use the GP_text special characters; all characters in the text editor will be the same font.
CURSOR_COLOR *GP_color*	The color of the cursor, which defaults to black.
CURSOR_INDEX *integer*	This temporary property denotes which character the cursor is in front of. For example, when CURSOR_INDEX is 0, this means that the cursor is in front of the first character.
CURSOR_STYLE *SUIT_enum*	This denotes the style of the cursor, either "vertical bar" or "i-beam."
CUT_BUFFER *text*	Contains the last text cut from the CURRENT_VALUE string. This buffer is set whenever the WIPE_BLOCK_KEY or KILL_LINE_KEY key sequence is used (CUT_BUFFER is set to be the killed text) or when the user highlights the text by dragging over the text editor with the mouse (CUT_BUFFER is set to be the text in the highlighted region).
HIGHLIGHT_BLOCK *boolean*	When true, the text that appears between the MARK_INDEX and the MARK_END_INDEX appears in reverse video (light on dark).
INPUT_SEQUENCE *text*	This buffer stores the editor commands before the editor processes them. After the editor recognizes a command, this property is reinitialized to the null string. If you register an interest in this property, you can examine the keyboard command before the editor responds to it. See also the SUIT function called **`SUIT_sendToEditor()`**.
MARK_INDEX *integer*	This number denotes the beginning position of the highlighted region in the text. To change the position of the highlighted region, use the **`SUIT_highlightBlockInTextEditor()`** call.
MARK_END_INDEX *integer*	This number denotes the ending position of the highlighted region in the text. To change the position of the highlighted region, use the **`SUIT_highlightBlockInTextEditor()`** call.
NUMBER_OF_LINES *integer*	This represents the total number of text lines being edited.
READ_ONLY *boolean*	When TRUE, commands that change the buffer are disabled and the widget becomes a text browser rather than an editor.
SPACING_GAP *int*	Number of pixels between lines of text measured from the lowest descender to the highest ascenders of the next line.
TAB_LENGTH *int*	The length of a tab, measured in characters. For example, setting tab length to 5 will provide tabs every five spaces across the width of the editor. Note that tabs will only line up with a fixed width font such as courier.Tabs are preserved in the text as tabs, not converted to spaces.

Text Editor

Class: “text editor”

Displays:



standard

`SUIT_object SUIT_createTextEditor(char *name, void callback (SUIT_object))`

Description: Creates an emacs-line text editor. The text that appears in the editor is a single string, which may contain imbedded newlines.

See Also: `SUIT_textOfFile()` on page 142.

```
SUIT_object SUIT_createTextEditorWithScrollBar(char *name,
                        void callback(SUIT_object))
```

Description: Creates a bulletin board which contains both an emacs-line text editor and an associated scrollbar. The text that appears in the editor is a single string, which may contain imbedded newlines. The object returned is the text editor, not the bulletin board.

`void SUIT_sendToEditor(SUIT_object o, char *command)`

Description: This sends a stream of characters to the text editor as if they were typed into the text editor directly. The command string may contain any keybinding (see page 144) defined for the text editor. Strings not recognized as commands are entered into the CURRENT_VALUE buffer exactly as they appear in the given command string.

`void SUIT_highlightBlockInTextEditor(SUIT_object o, int beginPos, int endPos)`

Description: This function highlights the section of the text between the beginPos and the endPos. First position in the text editor is position 0.

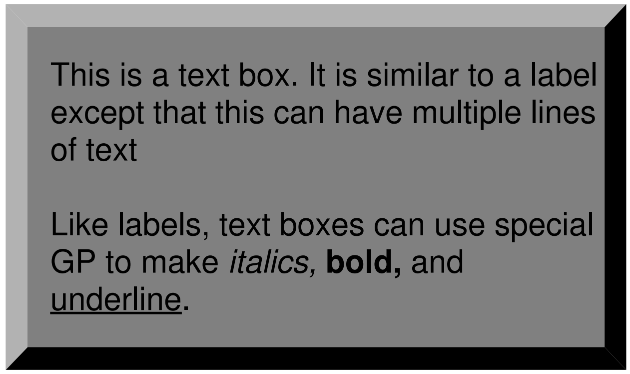
`char *SUIT_textOfFile (char *filename)`

Description: Provided as a convenience. Opens and returns, as a single string, the contents of a file. Can be used to set the **CURRENT_VALUE** property of a Text Editor widget.

Text Box

Class: “text box”

Display Styles:



standard

`SUIT_object SUIT_createTextBox(char *name, char *textToDisplay)`

Description: This is essentially a multiline label that performs wrapping of the text (no hyphenation, sorry) automatically, depending on the size of the viewport. The text box can use GP’s special text attribute notation to create text boxes with underlines, bold, italics and special characters like ligatures and accents. For more information on this notation, see page 104.

Example:

```
char *Notes = "This is a text box. It is similar to a label except
that this can have multiple lines of text.\n Like labels, text boxes
can use special GP to make @italic(italics), @bold(bold), and @under-
line(underline).";

SUIT_createTextBox("my notes", Notes);
```

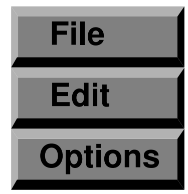
Property Name	Notes
LABEL *text*	The text that appears on the face of the Text Box. This string is allowed to contain imbedded newlines.
LINE_SPACING *double*	This number determines the space between lines of text, measured in text heights. 2.0 is double spaced.
NUMBER_OF_LINES *integer*	This number represents the number of lines of text currently being displayed. This number changes if text is added, or if the widget resizes and lines wrap differently.
STARTING_LINE *integer*	Usually, this number is zero, meaning that the text being displayed starts at the first line of the total amount of text held by this text box (the **LABEL** property). By setting **STARTING_LINE** to some number greater than zero, the text can be made to scroll off the top of the text box.

Stacker

Class: “stacker”

Display Styles:

horizontal stacking



vertical stacking

`SUIT_object SUIT_createStacker (char *name);`

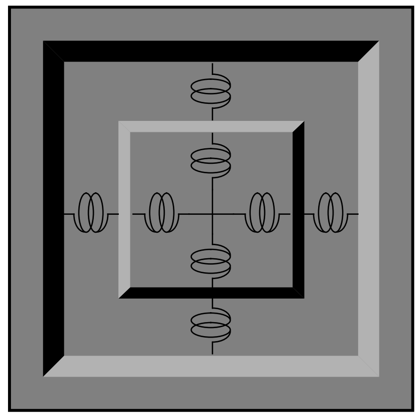
Description: This widget, like the bulletin board, is meant to hold other widgets. Stackers arrange their children in horizontal or vertical stacks, depending on the display style. The hieght of a stacker in the "horizontal stacking" display style is the height of the tallest child widget. Similarly, the width of a vertical stacker is that of its widest child. You can add children to stacker widgets as you would a bulletin board with **`SUIT_addChildToObject()`** or interactively, by opening up the stacker (SUIT-o) and SUIT-dragging the new children into the stacker. With vertical stackers,. new children are added to the stacker above the object you drop the new child on, and with horizontal stackers, new children are added to the right. Children maintain their size and shape as they are added to the stacker; the stacker changes size to accomodate them.

Property Name	Notes

Spring Panel

Class: “spring panel”

Display Styles:



standard

`SUIT_object SUIT_createSpringPanel(char *name)`

Description: This widget allows the user to control the SUIT_springiness data type. SUIT_springiness is used to control the way a widget behaves when the widget’s parent resizes (e.g. resizing the application window or a bulletin board widget).

See also: There is a discussion of springiness on page 156 of the reference manual.

Scrollable List

Class: "scrollable list"

Display Styles:

standard d

```
SUIT_object SUIT_createScrollableList(char *name,
                                      void(*callback)(SUIT_object))
```

Description: Allows the user to select one string from a scrolling list of strings.

Parameters: `name`: The name of the scrollable list widget.

`callback:` A function to call when a string is selected.

Property Name	Notes
CURRENT_VALUE *text*	The currently selected string. Initially, this is the empty string.
CURRENT_ROW *integer*	The integer number of the currently selected string. The first string is string 0. If nothing is highlighted, the **CURRENT_ROW** is -1.
HAS_BACKGROUND *boolean*	When FALSE, the background can be seen between the text list and the scrollbar.
HAS_BORDER *boolean*	When FALSE, the scrollbar appears to be separate from the text list.
LABEL *text*	If not the empty string "", the scrollable list will display this **LABEL** string at the top of the list. For example, you could set this property to be the string "colors" for the list of items: "red", "green", and "blue".
LIST *SUIT_textList*	This is the entire list of items in the scrollable list.

Radio Button Utility Functions

---

void SUIT_addButtonToRadioButtons (SUIT_object radio, char *buttonName)

Description: Allows the programmer to add a button choice to a set of radio buttons. The button choice is added to the bottom of the list of choices already present in the set of radio buttons (if any).

Parameters: **radio:** A set of radio buttons created in a call to **SUIT_createRadioButtons().**

name: The name of the button to add. The name must be unique among the names of all SUIT objects in that SUIT will use this name to create a button widget for you.

Example:

```
/* Using the radio buttons */

void ChooseRadioButton(SUIT_object obj) {
printf ("button pressed is %s\n",
                    SUIT_getEnumString(obj,CURRENT_VALUE));
}

void MakeRadioButtons(void) {
     SUIT_object radio = SUIT_createRadioButtons("my buttons",
            ChooseRadioButton);
     SUIT_addButtonToRadioButtons (radio, "Underdog");
     SUIT_addButtonToRadioButtons (radio, "Bullwinkle");
     SUIT_addButtonToRadioButtons (radio, "Ren and Stimpy");
}
```

---

void SUIT_pressThisRadioButton (SUIT_object radio_buttons, char *buttonName)

Description: Allows the programmer to select one of the choices on the set of radio buttons. Note that this does NOT call the radio button callback, it merely changes the CURRENT_VALUE and changes the display to match.

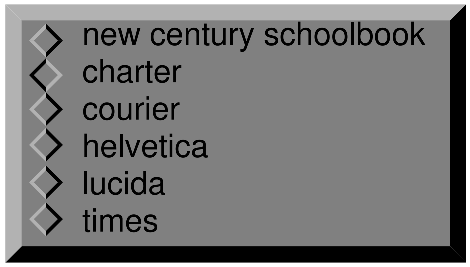
Example:

```
/* select the Fred radio button choice under program control */

SUIT_object radio;

radio = SUIT_createRadioButtons("Flintstones", NULL);
SUIT_addButtonToRadioButtons(radio, "Fred");
SUIT_addButtonToRadioButtons(radio, "Barney");
SUIT_addButtonToRadioButtons(radio, "Wilma");

SUIT_pressThisRadioButton(radio, "Fred");
```

Radio Buttons

Class: "radio buttons"

Display Styles:



`SUIT_object SUIT_createRadioButtons (char *name,`
`void (*callback)(SUIT_object))`

Description: Allows the user to select one (and only one) item from a list of many choices in a mutually exclusive fashion.

Parameters: **`callback:`** A callback to the radio button that SUIT will execute when any of the radio buttons are pressed.

See Also: Radio Button Utility Functions on page 137.

Examples: See the example programs radio1.c and radio2.c

Property Name	Notes
CURRENT_VALUE *SUIT_enum*	This is the item that is currently selected. The string representation of the current SUIT enumeration is taken to be the label for each of the radio buttons.

Menu Functions

SUIT_object SUIT_createVerticalMenu (char *name)

Description: This creates a placard style menu. (Like a pulldown menu that is always down.)

SUIT_object SUIT_createMenuBar (char *name)

Description: This creates a menu bar that can hold other menu buttons. This is just a horizontal stacker.

Example: See the example file menu.c

```
SUIT_object SUIT_addToMenu(SUIT_object obj, char *buttonName,
                           void callback (SUIT_object))
```

Description: This adds a menu item to a menu.

Parameters: **buttonName**:This is the name of the button. Because this is a widget name, this string must be unique among all widget names in your application.

callback This is the function that you want called when this choice of the menu is used. The SUIT_object passed in as a parameter is the menu that was used.

```
SUIT_object SUIT_addToMenuWithHotKey (SUIT_object obj, char *button Name, void
                    callback (SUIT_object), char *hotkey)
```

Description: This adds a menu item to a menu and adds a hot key designation for the label. You will need to register a trapper function to make the hotkey active (See page 70).

Pulldown Menu

Class: “pulldown menu”

Display Styles:

standard

`SUIT_object SUIT_createPullDownMenu (char *name)`

Description: This creates a menu with a button that can be pressed, revealing choices underneath.

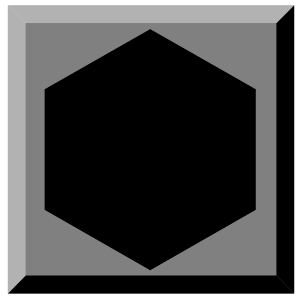
Example: See the example file menu.c

Property Name	Notes
LABEL *text*	This is the label that appers on the front of the pulldown menu button.

Polygon

Class: “polygon”

Display Styles:



standard

`SUIT_object SUIT_createPolygon(char *name)`

Description: This is the famous polygon widget that appears in the SUIT tutorial.

Property Name	Notes
FILLED *boolean*	When TRUE, the polygon will be filled solid in the **FOREGROUND_COLOR**. When FALSE, only the outline is drawn.
NUMBER_OF_SIDES *integer*	The number of sides for the polygon. Setting this property to be a number less than 3 will draw a triangle.

Place Mat

Class: “place mat”

standard

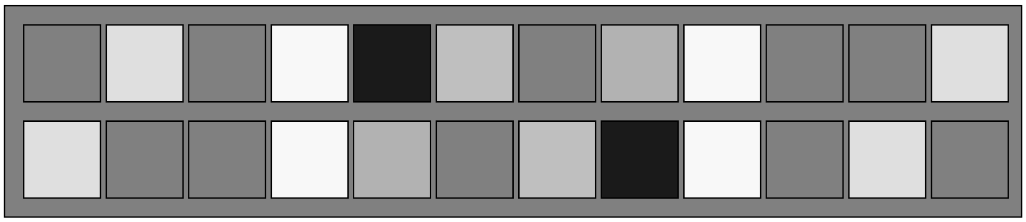
`SUIT_object SUIT_createPlaceMat (char *name)`

Description: This creates a do nothing square on the screen. It has neither a paint procedure nor a hit procedure, making it useful only for decoration. Commonly placed behind other widgets to create a particular visual effect.

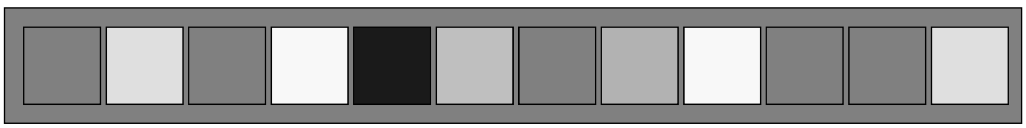
Pattern Chips

Class: "pattern chips"

Display Styles:



parquet



standard

The picture here is schematic. The real pattern chips show 40 different patterns.

`SUIT_object SUIT_createPatternChips(char *name, void callback (SUIT_object))`

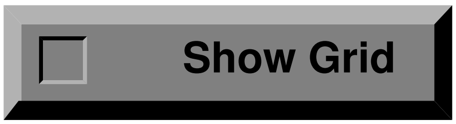
Description: Allows the user to select one of the standard GP patterns.

Property Name	Notes
CHIP_BORDER *double*	The width of the border around each chip.
CURRENT_VALUE *integer*	Pattern currently selected.
HIGHLIGHT_COLOR *GP_color*	The color of the highlight around the selected pattern chip.
PREVIOUS_VALUE *integer*	Pattern previously selected.

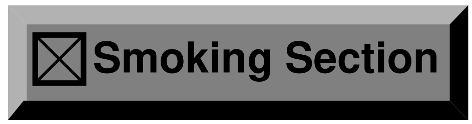
On Off Switch

Class “on/off switch”

Display Styles:



motif square



check box

motif diamond

dot in circle

`SUIT_object SUIT_createOnOffSwitch (char *name, void callback (SUIT_object))`

Description: Allows the user to select a boolean state.

Property Name	Notes
CURRENT_VALUE *boolean*	Set to yes if switch is on, no if the switch is off.
DISABLED *boolean*	If TRUE, the text for the button is drawn in the **DISABLED COLOR** (rather than **FOREGROUND COLOR**) and the callback is not called when the button is pressed. This is useful for times in an application when certain functions are not valid - for example, a drawing editor might disable the “delete current object” button whenever there was no currently selected object.
DISABLED_COLOR *GP_color*	The color of the text when the button is disabled (see object level property **DISABLED**).
LABEL *text*	The text to display on the face of the button.

Label

Class: “label”

Display Styles:

standard

`SUIT_object SUIT_createLabel (char *name)`

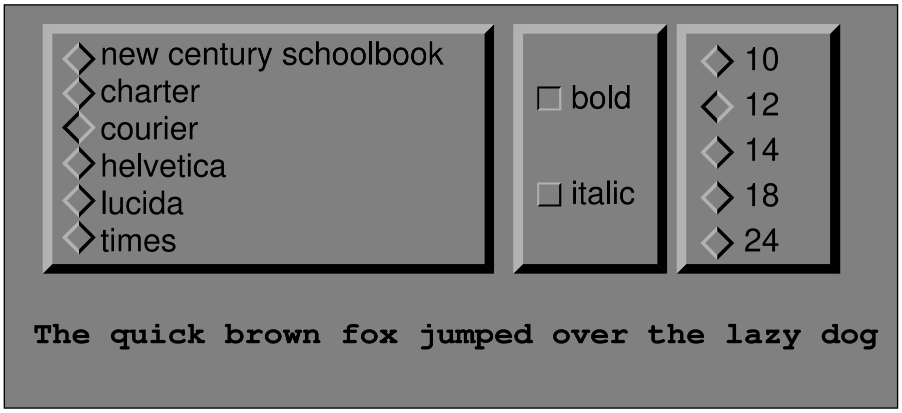
Description: This call creates a widget that displays a text string. It has no callback. The default label is its name, though you can change that by setting the **LABEL** property. By default, labels are set to shrink to fit and to have no border, but these properties may be changed through the **SHRINK_TO_FIT** and **HAS_BORDER** properties, respectively.

Property Name	Notes
HAS_BORDER *boolean*	By default set to FALSE, meaning that it would have no border.
LABEL *text*	The string that appears on the face of the label.

Font Panel

Class: "font panel"

Display Styles:



standard

`SUIT_object SUIT_createFontPanel (char *name)`

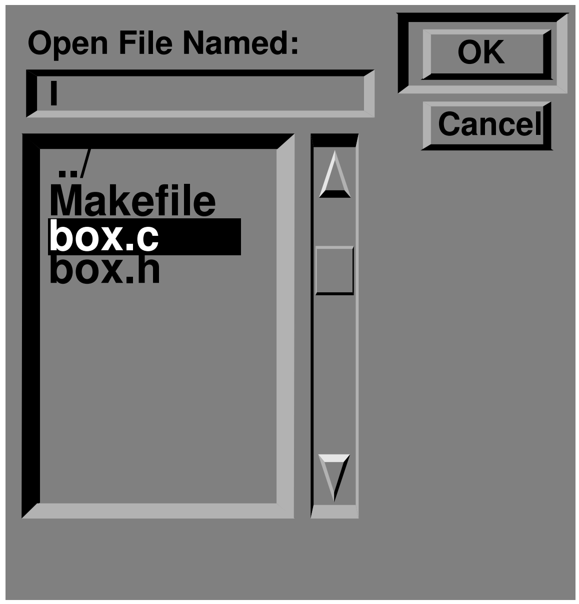
Description: This widget allows the user to select one of the standard GP fonts. The CURRENT_VALUE property of this widget is the font that is currently displayed on the panel.

Property Name	Notes
CURRENT_VALUE *GP_font*	This is the font that is currently selected on the font panel.

File Browser

Class: “file browser”

Display Styles:




```
SUIT_object SUIT_createFileBrowser (char *name, char *startDirectory,
                                    char *label, char *typeInBoxLabel,
                                    SUIT_callbackFunctionPointer callback))
```

Description: Creates a widget that displays the files in a given directory, starting with the given **`startDirectory`** . This currently only works on UNIX platforms.

NOTE: To specify the "current directory", use a NULL srting or "", not the sting "."

Property Name	Notes
CURRENT_DIRECTORY *string*	The name of the directory (without the final filename appended) that the browser is looking at
CURRENT_FILE *string*	The name of the file (without the pathname prepended) that the browser is looking at.
CURRENT_VALUE *string*	The name of the file or directory that appears in the type-in box. This is the combination of CURRENT_DIRECTORY and CURRENT_FILE property values.
FILE_FILTER *string*	A wildcard specification (e.g. *.c or *.sui) that limits the kins of files that the browser will list. Example: if FILE_FILTER is set to *.c, the browser will only list files that end in ".c".

```
Reply SUIT_getString(char *message, char *defaultString,
                        char answer[], int answerLength)
```

Description: Brings forth a dialog box with a type in box, an OK button and a CANCEL button. The string that the user typed comes back in the parameter called **answer[]**, which must be an allocated array of charcters of length **answerLength**. The **message** parameter is the prompt for the user and the **defaultString** is a character constant that appears in the type in box when the box first appears.

Returns: **REPLY_OK**
REPLY_CANCEL

Example:

```
char answer[50];

if (SUIT_getString("What is your name?",
                   "Bob the Amazing", answer, 50) == REPLY_OK){
     printf ("hello, %s\n", answer);
}
```

```
SUIT_object SUIT_createFileBrowserDialogBox (char *name, char *startDir,
                        char *label, char *typeInBoxLabel,
                        void callback(SUIT_object) )
```

Description: Brings forth a dialog box with a file browser widget, an OK button and a CANCEL button in it. As with the general **SUIT_createOKCancelDialogBox()**, to cause the dialog box to be visible to the user, use **SUIT_activateDialogBox()**. In general, it is usually easier to call **SUIT_askForFileName()**, which handles all the creation/activation/destruction operations, and just produces a string value for a file name as a result. Use **SUIT_createFileBrowserDialogBox()** call when you want more control over the dialog box (position, color, font, etc.).

NOTE: To specify the "current directory", use a NULL srting or "", not the sting "."

See Also: **SUIT_creaateFileBrowser()** on page 127.

```
char *SUIT_askForFileName (char *startDirectory, char *label,
                              char *typeInBoxLabel)
```

Description: Brings forth a dialog box with a file browser widget, an OK button and a CANCEL button in it in it. The value returned is the name of the file that the user selected, or NULL if the user pressed CANCEL.

NOTE: To specify the "current directory", use a NULL srting or "", not the sting "."

See Also: **SUIT_creaateFileBrowser()** on page 127.

Reply SUIT_askOKCancel(char *question)

Description: Brings forth a dialog box that asks a question and offers two responses: OK and CANCEL.

Returns: **REPLY_OK**
REPLY_CANCEL

Example:

```
if (SUIT_askOKCancel ("About to transmit large file.") == REPLY_OK)
     printf("User selected OK");
else
     printf("User selected CANCEL");
```

Reply SUIT_askYesNo(char *question)

Description: Brings forth a dialog box that asks a question and offers two responses: YES and NO.

Returns: **REPLY_YES**
REPLY_NO

Example:

```
if (SUIT_askYesNo ("Are you sure?") == REPLY_YES)
     printf ("User selected YES");
else
     printf ("User selected NO");
```

Reply SUIT_askYesNoCancel (char *question)

Description: Brings forth a dialog box that asks a question with three responses: YES, NO and CANCEL

Returns: **REPLY_YES**
REPLY_NO
REPLY_CANCEL

Example:

```
switch (SUIT_askYesNoCancel("Save changes before exiting?")) {
 case REPLY_YES:
     SaveTheChanges();
     break;
 case REPLY_NO:
     ExitWithoutSaving();
     break;
 case REPLY_CANCEL:
     /* Do nothing, user returns to application */
     break;
}
```

SUIT_object SUIT_createOKCancelDialogBox(char* name, SUIT_object innards, SUIT_validationFunction dataOK)

Description: Creates a dialog box that contains a single SUIT widget and offers two responses: OK and CANCEL. The widget can be a bulletin board that itself contains many widgets. This function does not actually place the dialog box on the screen, that is done with the function called **SUIT_activateDialogBox()**.

Parameters: **name** This is the name for this widget.
innards This is the SUIT_object that the dialog box is to wrap around.
dataOK This is a function that returns a boolean that indicates whether the data specified in the dialog box is valid or not and is called when the user selects the dialog box's OK button. The validation function takes a single SUIT_object as a parameter which will be the the object that the dialog box is currently wrapping. If your validation function returns TRUE, this means that the data from the wrapped widget is valid and **SUIT_activateDialogBox()** returns REPLY_OK. If the validation function returns FALSE, this means that there is something wrong with the data that the user specified, in which case, **SUIT_activateDialogBox()** does not return at all; the dialog box stays up, waiting for the user to continue interacting with the widget in the dialog box. It is not uncommon for the validation function to use **SUIT_inform()** to notify the user that the data was invalid, though this is not required.

Returns: A SUIT_object that represents the dialog box. Use it as the argument to **SUIT_activateDialogBox()**.

See Also: **SUIT_activateDialogBox()** page 124.
Example program for dialog boxes included in the SUIT distribution.

Reply SUIT_activateDialogBox (SUIT_object dbox)

Description: This function causes the dialog box created with **SUIT_createOKCancelDialogBox()** to be placed on the screen. The dialog box waits for input and exits with either **REPLY_OK** or **REPLY_CANCEL**.

See Also: **SUIT_createOKCancelDialogBox()** page 124.
Example program for dialog boxes included in the SUIT distribution.

Reply SUIT_askWithCancel(char *question, char *button1Name, char *button2Name)

Description: Asks a question and places three buttons on the screen. The first two buttons that appear are named by the two buttonName parameters. The last button is a cancel button.

Returns: **REPLY_BUTTON1**
REPLY_BUTTON2
REPLY_CANCEL

Example:
```
switch (SUIT_askWithCancel("Save changes before exiting?",
                              "Save", "Don't Save")) {
 case REPLY_BUTTON1:
     SaveTheChanges();
     break;
 case REPLY_BUTTON2:
     ExitWithoutSaving();
     break;
 case REPLY_CANCEL:
     /* Do nothing, user returns to application */
     break;
}
```

Dialog Boxes

Class: “dialog box”

All SUIT dialog boxes are “moded” meaning that once activated, they must be answered (by pressing an OK or Cancel button, for example) before you can interact with the rest of your application. For code examples using dialog boxes, see the DialogBoxes program in the SUIT examples directory.

IMPORTANT: Dialog Boxes return values of type Reply. (See page 35). Code that depends on a particular numeric value being returned from the dialog boxes may not work as expected.

Example:

WRONG:

```
if (SUIT_askYesNo(“are you there?”) )
....
```

RIGHT:

```
if (SUIT_askYesNo(“are you there?”) == REPLY_YES)
....
```

`void SUIT_inform (char *info)`

Description: Creates and activates a dialog box that contains a text string and a single button: OK. Like all dialog boxes, the **`SUIT_inform()`** box must be answered before the user can interact with the rest of the application.

Example:

```
SUIT_inform ("Mail Sent");
```

`Reply SUIT_ask (char *question, char *button1Name, char *button2Name)`

Description: Asks a question and places two buttons on the screen. The buttons that appear are named by the two buttonName parameters.

Returns:

`REPLY_BUTTON1`
`REPLY_BUTTON2`

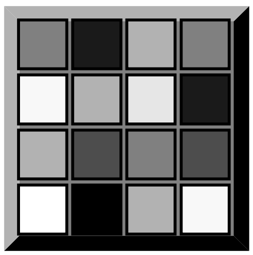
Example:

```
if (SUIT_ask("Which units?", "Meters", "Feet") == REPLY_BUTTON1)
     printf ("user seleced Meters");
else
     printf ("user selected feet");
```

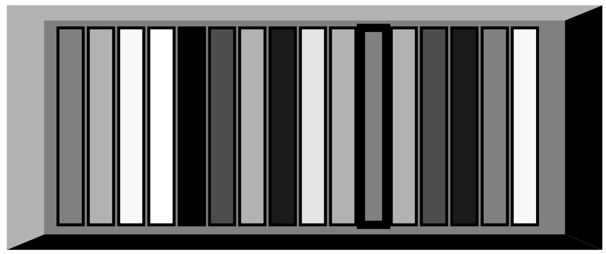
Color Chips

Class: “color chips”

Display Styles:



parquet



standard

`SUIT_object SUIT_createColorChips (char *name, void callback (SUIT_object))`

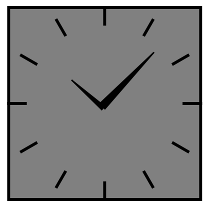
Description: Allows the user to select a color from the available palette of colors.

Property Name	Notes
CHIP_BORDER *double*	The current color is shown by drawing a highlighted border around its chip. The **CHIP BORDER** property controls the width of that border, expressed as a fraction of the width of the entire set of color chips. Typical range for this property is (0.05-0.1).
CURRENT_VALUE *GP_color*	The color currently selected.
HIGHLIGHT_COLOR *GP_color*	The color of the border used to highlight of the **CURRENT_VALUE** color.
PREVIOUS_VALUE *GP_color*	The color previously selected.

Clock

Class: “clock”

Display Styles:



analog

digital

SUIT_object SUIT_createClock (char *name)

Description: This creates a widget that displays the system time from the computer’s clock.

Property Name	Notes
HAS_RIM *boolean*	Denotes whether or not the analog clock has a circular rim around it. Has no effect on the digital display style of the clock.
HAS_SECOND_HAND *boolean*	Denotes whether or not the analog display of the clock displays a second hand. This property can be toggled with the mouse. In the digital display style, if **HAS_SECOND_HAND** is TRUE, then the seconds are updated every second, otherwise they remain at zero.
MILITARY_TIME *boolean*	When TRUE, the digital display shows the time using a 24 hour clock. Unused in the analog display.

Buttons (Abort and Done)

SUIT_object SUIT_createAbortButton (SUIT_callbackFunctionPtr callback)

Description: Creates a button whose label says "abort" and whose callback calls **SUIT_done(ABORT)**, which leaves the application without saving the .sui file. If your application requires that some cleanup be done before SUIT exits, you can pass it a pointer to a function that will get called before the program ends. If you have no such function, you can pass NULL.

Example:

```
void MyCleanupFunction(SUIT_object me){
     /* "me" is the Abort button object. */
     }

obj = SUIT_createAbortButton(MyCleanupFunction);
```

See Also: **SUIT_done()** page 50

SUIT_object SUIT_createDoneButton (SUIT_callbackFunctionPtr callback)

Description: Creates a button whose label says "done" and whose callback calls **SUIT_done(NORMAL_EXIT)**, which leaves the application after saving the.sui file. If your application requires that some cleanup be done before SUIT exits, you can pass it a pointer to a function that will get called before the program ends. If you have no such function, you can pass NULL.

Example:

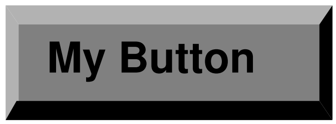
```
void MyCleanupFunction(SUIT_object me) {
     /* me is the Done button object. */
     }

obj = SUIT_createDoneButton(MyCleanupFunction);
```

See Also: **SUIT_done()** page 50

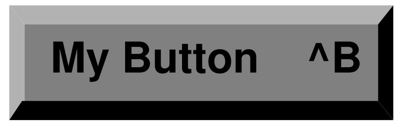
Button Widget

Class: “button”

Display Styles:



standard



button with hot key

Standard is a rectangular region.
Button with hot key shows the hot key (if any) that is associated with the button**.**

`SUIT_object SUIT_createButton (char *name, void callback (SUIT_object))`

Description: Creates a button that invokes the callback function when the button is pressed.

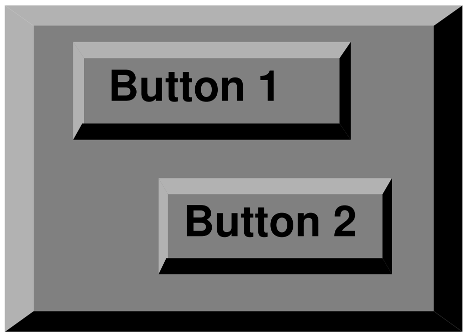
Parameters: **`name:`** By default, the text on the front of the button is the same as its name, but you can change that by setting the “label” property of the button to something else.

Property Name	Notes
DISABLED *boolean*	If **TRUE**, the text for the button is drawn in the **DISABLED COLOR** (rather than **FOREGROUND COLOR**) and the callback is not called when the button is pressed. This is useful for times in an application when certain functions are not valid - for example, a drawing editor might disable the “delete current object” button whenever there was no currently selected object.
DISABLED_COLOR *GP_color*	Sets the color of the button if the button is deemed to be an “inactive choice” in the application. See the **DISABLED** property at the object level.
HOTKEY *text*	The keyboard combination that can also be used to activate the button. Note that this is only a piece of text that appears alongside the LABEL. In order to actually attach the hotkey to the button requires registering an input trapper function.
LABEL *text*	The name that appears on the front of the button.

Bulletin Board

Class: “bulletin board”

Display Styles:



standard

For a complete understanding of this widget, read the section called "Hierarchy" on page 153.

SUIT_object SUIT_createBulletinBoard (char *name)

Description: Creates a container to hold other widgets.

See Also: A full explanation of bulletin boards begins with the section on Hierarchy on page 153. Functions that manipulate hierarchical widgets are described on page 72.

Example: To create a bulletin board that contains 2 buttons:

```
SUIT_object board, button;

board = SUIT_createBulletinBaord (“my board”);
button1 = SUIT_createButton (“my button”, callback1);
button2= SUIT_createButton (“another button”, callback2);

SUIT_addChildToObject (board, button1);
SUIT_addChildToObject (board, button2);
```

SUIT_object SUIT_createBulletinBoardWithClass (char *name, char *className)

Description: Creates a container to hold other widgets, but registers a different class name for the bulletin board so that it does not inherit the class level properties that govern all bulletin boards.

See Also: A full explanation of bulletin boards begins with the section on Hierarchy on page 153.

Example: If, in the above example, you replaced

```
SUIT_createBulletinBoard(“my board”);
```

with

```
SUIT_createBulletinBaordWithClass(“my board” , “control panel”);
```

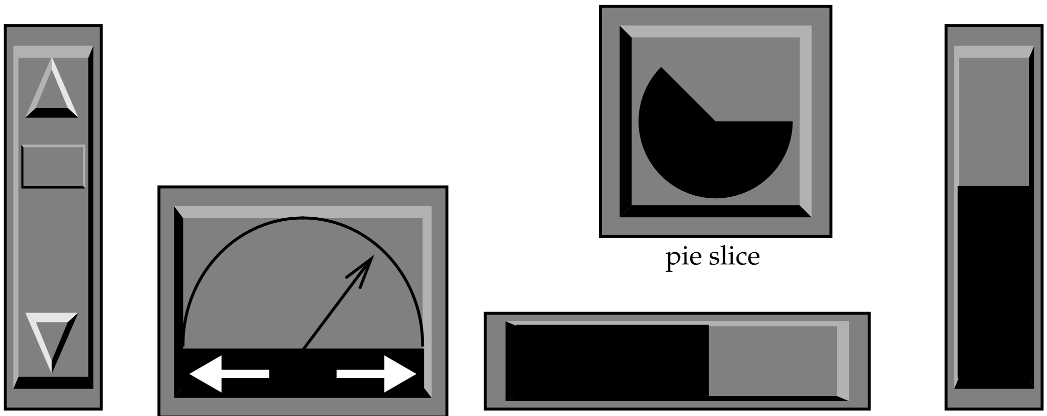
The widget would look class level properties up under the “control panel” class, not the “bulletin board” class, allowing control panels and bulletin boards to possess different properties.

HAS_ARROW *boolean*	If TRUE, the speedometer needle ends in an arrow. Ignored in all other display styles.
HAS_TICK_MARKS *boolean*	If TRUE and if **GRANULARITY** is not zero, tick marks will be drawn on the sides of the thermometer display styles for the bounded value: one mark for each **GRANULARITY** step.
INCREASE_CLOCKWISE *boolean*	If TRUE, the pie slice display style increases in a clockwise fashion.
MAXIMUM_VALUE *double*	The highest possible value.
MINIMUM_VALUE *double*	The lowest possible value.
NEEDLE_COLOR *GP_color*	Color of speedometer needle and its arrowhead.

Bounded Value

Class: “bounded value”

Display Styles:



scroll bar speedometer pie slice horizontal thermometer vertical thermometer

`SUIT_object SUIT_createBoundedValue (char *name, void callback (SUIT_object))`

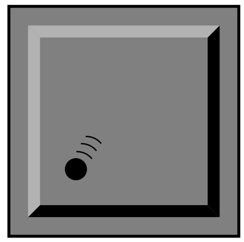
Description: Allows the user to select an integer or double between the minimum and maximum values specified. By setting the “granularity” property to 1.0, the current value of the widget can be constrained to integral values (i.e. doubles with no fractional part).

Property Name	Notes
ARROWHEAD_ANGLE *double*	The angle in degrees that the arrow head makes with the needle in the speedometer display style. A typical arrow would range from 10 to 50 degrees. 0 degrees would give no arrow at all.
ARROWHEAD_LENGTH *double*	Length of the arrowhead, expressed in world coordinates. Applies only to the speedometer display style. A typical arrowhead would range from 0.1 to 0.5 in length.
BUTTON_BACKGROUND_COLOR *GP_color*	Color of the bar on the bottom of the speedometer where the left/right arrows are shown.This only applies to the speedometer display style.
BUTTON_FOREGROUND_COLOR *GP_color*	Color of the left/right arrows shown below the speedometer. This only applies to the speedometer display style.
CURRENT_VALUE *double*	As shown by the widget, this is a floating point number between the minimum and maximum values.
GRANULARITY *double*	In the speedometer and scroll bar displays, this is the amount that the buttons will increment or decrement the current value of the bounded value. In all displays, the current value must be a multiple of the granularity - for example, setting a **GRANULARITY** of 1.0 effectively makes the bounded value an integer-only widget.

Bouncing Ball

Class: “bouncing ball”

Display Styles:



standard

`SUIT_object SUIT_createBouncingBall(char *name)`

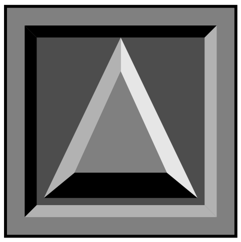
Description: This creates a ball that bounces around inside the box. Purely for demonstrating animation.

Property Name	Notes
BALL_SIZE *double*	The diameter of the ball expressed in world coordinates.
BALL_X *double*	The X position of the ball, in world coordinates.
BALL_Y *double*	The X position of the ball, in world coordinates.
FILLED *boolean*	When TRUE, the ball is filled in with the FOREGROUND_COLOR.
PIXELS_PER_SECOND *int*	The number of pixels per second (the speed) that the ball moves

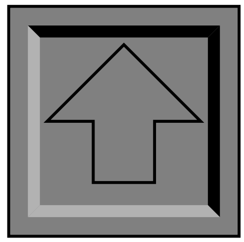
Arrow Button

Class: "arrow button"

Display Styles:



motif



simple arrow

`SUIT_object SUIT_createArrowButton(char *name, void callback (SUIT_object))`

Description: Just like regular buttons, arrow buttons call functions when they are pressed. These buttons are provided as a convenience.

Property Name	Notes
DARKEN_BACKGROUND *boolean*	Determines whether the background of a motif arrow is drawn darker than the BACKGROUND_COLOR for this widget.
DIRECTION *SUIT_enum*	Sets direction of the arrow: up, down, left, or right.
INTERMEDIATE_FEEDBACK *boolean*	Determines whether the widget is currently displaying intermediate feedback -- a visual cue to let you know that the button is depressed but has not been released yet. Most users need not pay attention to this property.
SHADOW_THICKNESS *integer*	Sets the depth, in pixels, of the shadow for the motif arrow. Most users need not pay attention to this property.

Common Properties

These are properties that are found at the object level for many widgets.

Property Name	Notes
ACTIVE_DISPLAY *SUIT_enum*	Each screen object implements an abstract idea, such as a clock. However, most abstract ideas can be displayed on the screen in many different ways. These are called display styles, or just "displays." A clock, for example, has two different displays: digital and analog. The **ACTIVE_DISPLAY** property tells SUIT which display style to use. By convention, if a widget only has one display style, that style is called "standard." You can change the display style for a widget without using the property editor: just type SUIT-C with the mouse over that widget to "cycle" to the next display.
CALLBACK_FUNCTION *SUIT_functionPointer*	The function to be called when the SUIT object is hit. This is a property that gets bound at runtime, but has no ASCII text representation. You should not attempt to change this temporary property interactively.
NUMBER_OF_CHILDREN *integer*	This property applies to bulletin board and pulldown menu widgets, and indicates how many different widgets are contained in the bulletin board or menu. All children of a widget must stay inside that widget on the screen. Widgets with children are sometimes called hierarchical widgets; they and all their children can be manipulated as a unit. hierarchical widgets can even have other hierarchical widgets as their children. This property should never be directly set by the user.
VIEWPORT *SUIT_viewport*	The physical location of the SUIT object on the screen. Changing this property is usually done interactively with SUIT-drag, but it is also possible to change a viewport by typing the four integers into the property editor directly. The four numbers correspond to lower-left x coordinate, lower-left y coordinate, upper-right x coordinate, upper-right y coordinate, also known as x1, y1, x2, y2. For more information, see the section on Windows and Viewports in the SUIT Reference Manual. Also see: **WINDOW**.

When you click on a springiness property, you interact with pictures of the springs and rods. Of course, in each of the directions (horizontal, vertical), there must be some springiness - if a widget was connected to its parents by a stiff rod on both its left and right sides, and had a stiff horizontal rod inside, it would be "overconstrained" if its parent got wider or narrower - that's why you must always have at least one spring in each direction (vertical and horizontal). For more information, see the discussions of springiness in the heirarchy section of the reference manual.

SUIT_SYSTEM_FONT
GP_font

This is the font used by SUIT for the property editor and the SUIT menu.

VISIBLE
boolean

Denotes whether the widget is visible in the application. Widgets that have their **VISIBLE** property set to FALSE are not painted on the screen, and do not respond to mouse events. If you set **VISIBLE** to FALSE accidently and then exit the property editor, the only way to set the property back to TRUE is through program control or by hand editing the .sui file. Also see: **VISIBLE_WITHIN_PROPERTY_EDITOR**.

VISIBLE_WITHIN_PROPERTY_EDITOR
boolean

Denotes whether the widget is visible while being edited in the property editor. All the widgets except for the one being edited have their **VISIBLE_WITHIN_PROPERTY_EDITOR** set to FALSE. You should not need to worry about this property - it is used internally by SUIT.

WINDOW
SUIT_window

It is often inconvenient for widgets to draw in screen (pixel) coordinates, so most widgets use another coordinate system and they let the GP graphics package perform the calculations to figure out which pixels the graphics should end up on. The **WINDOW** property denotes the floating point coordinate system that a widget uses to display its graphics. It's an arbitrary rectangle used as a convenience in the painting routine. For example, a widget that implemented football fields could draw in a coordinate system which was 100 yards by 40 yards. Most widgets just use from 0.0 to 1.0 in each direction. For more information, see the section on Windows and Viewports in the SUIT Reference Manual. Also see: **VIEWPORT**.

FONT
GP_font

Default font that widgets will use if they do not specify a font at their class or object levels. The format of the string you see is: typeface, style, point size. For example 10 point bold italic Times would be written: times, bold italic, 10.

FOREGROUND_COLOR
GP_color

The default foreground color. Widgets paint their graphics in the foreground color. For example, the color of the text on a button is drawn in **FOREGROUND_COLOR**, but the button itself is considered the **BACKGROUND_COLOR**. Also see: **BACKGROUND_COLOR** and **BORDER_COLOR**.

HAS_BACKGROUND
boolean

When TRUE, SUIT will fill a widget's viewport by drawing a rectangle in the background color before calling that widget's paint procedure. When FALSE, SUIT will not do this. Most widgets leave this TRUE, except for those which are animated, or are performing optimized painting. Also see: **BACKGROUND_COLOR** and **ANIMATED**.

HAS_BORDER
boolean

When TRUE, the object is drawn with a border. Also see: **BORDER_RAISED**, **BORDER_WIDTH**, **BORDER_TYPE**, and **DRAW_BORDER_ON_INSIDE**.

MARGIN
integer

Denotes the number of pixels between the text and the widget border for those widgets that have text in them. This property applies to the top, bottom, left and right margins.

SCREEN_HEIGHT
integer

Vertical resolution of the screen in pixels. When SUIT is running on a computer screen using a window manager, this refers to the height of the window.

SCREEN_WIDTH
integer

Horizontal resolution of the monitor in pixels. When SUIT is running on a computer screen using a window manager, this refers to the width of the window.

SHOW_TEMPORARY_PROPERTIES
boolean

Each SUIT property is either temporary or permanent. Temporary properties are not saved to the .sui file when the program exits, therefore any changes made to a temporary property will not be in effect the next time the program is invoked. The SUIT property editor normally displays only permanent properties, because most temporary properties are used for internal bookkeeping purposes and aren't very interesting. When **SHOW_TEMPORARY_PROPERTIES** is TRUE, the SUIT property editor shows all properties, and indicates which are temporary by using a different color to display them, or by showing "(temporary)" after their name on a monochrome display.

SHRINK_TO_FIT
boolean

When TRUE, labels and buttons will resize themselves to be the size of the **LABEL** plus whatever **MARGIN** is set. Attempting to resize a widget that has **SHRINK_TO_FIT** set to TRUE will have no effect.

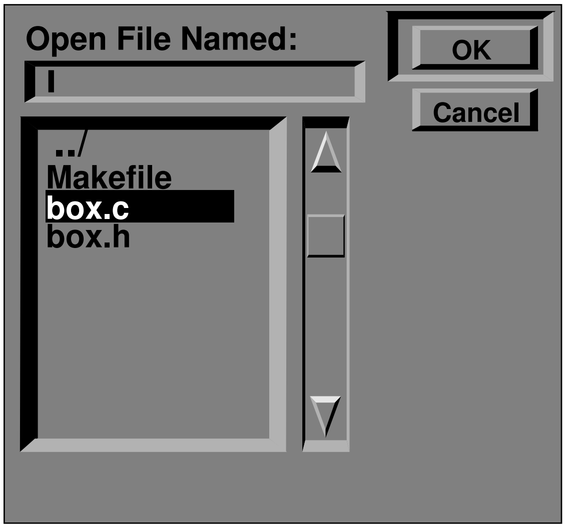
SPRINGINESS
SUIT_springiness

If a **SUIT** application is running inside a window in a window managed environment, and the user resizes the entire window, should each button get proportionally larger? The **SPRINGINESS** property allows widgets to specify what they do if their parent (for most widgets, this just means the window or screen) changes in size. Conceptually, imagine that each widget is connected to the four walls of its parent by either springs or stiff rods - as the parent's walls move, the widget will stay the same distance from a wall if it is connected by a stiff rod. In the same fashion, widgets describe whether they get bigger by having either a stiff rod or a spring in each of the two possible directions, horizontal and vertical.

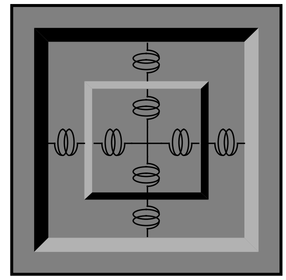
Global Properties

These are the properties that all objects can inherit from the global level. Because GLOBAL level properties usually hold important, system wide information, GLOBAL properties cannot be destroyed interactively with the SUIT property editor.

Property Name	Notes
ANIMATED *boolean*	True when an object continually performs some animated function. For example, when **ANIMATED** is TRUE for the bouncing ball widget, the ball constantly moves around in its box.
BACKGROUND_COLOR *GP_color*	Default background color. For beveled display styles and GP shapes, this should be set to the same color as the border color. Also see: **FOREGROUND_COLOR** and **BORDER_COLOR**.
BORDER_COLOR *GP_color*	Default border color. For beveled display styles and GP shapes, this should be set to the same color as the background color. Also see: **FOREGROUND_COLOR** and **BORDER_COLOR**.
BORDER_RAISED *boolean*	Used in the motif border style. Setting this to FALSE makes a widget appear to be depressed into the screen, away from the viewer. Setting this to TRUE makes a widget appear to be raised out of the screen, toward the viewer. Also see: **BORDER_TYPE**, **BORDER_WIDTH**, **DRAW_BORDER_ON_INSIDE**, and **HAS_BORDER.**
BORDER_TYPE *SUIT_enum*	Three options are available: “simple”, “motif” and “fancy motif.” A simple border is just a rectangle drawn in the border color. Motif borders have a 3-D look about them. Fancy motif borders have the 3-D look with much more decoration added. They are usually used for SUIT Dialog Boxes which grab the input focus and thus the user’s attention (hence the fanciness). Also see: **BORDER_RAISED**, **BORDER_WIDTH**, **DRAW_BORDER_ON_INSIDE**, and **HAS_BORDER**.
BORDER_WIDTH *integer*	The width of the border given in pixels. Also see: **BORDER_RAISED, BORDER_TYPE, DRAW_BORDER_ON_INSIDE**, and **HAS_BORDER**.
CAN_BE_OPENED *boolean*	Set to TRUE when the widget being examined is a pulldown menu or a bulletin board.
CLIP_TO_VIEWPORT *boolean*	When TRUE, a widget cannot draw graphics outside its own viewport. When FALSE, a widget can draw anywhere on the screen. This property is almost always TRUE. Also see: **VIEWPORT**.
DEFAULT_OBJECT_HEIGHT *integer*	Sets the height of widgets created interactively and widgets that have no **VIEWPORT** property specified(pixels).
DEFAULT_OBJECT_WIDTH *integer*	Sets the width of widgets created interactively and widgets that have no **VIEWPORT** property specified(pixels)
DRAW_BORDER_ON_INSIDE *boolean*	When **FALSE**, a widget’s border is drawn to the outside of the widget’s stated viewport. Also see: **BORDER_RAISED, BORDER_WIDTH, BORDER_TYPE**, and **HAS_BORDER**.

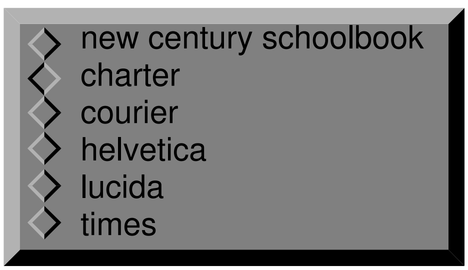


"File Browser" on page 127

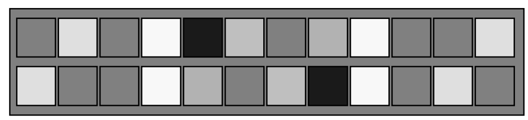


"Spring Panel" on page 139

"Trash Can" on page 145

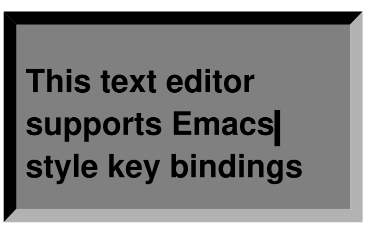


"Radio Buttons" on page 136

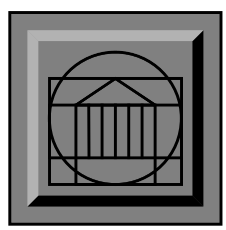


"Pattern Chips" on page 131

"Type In Box" on page 146

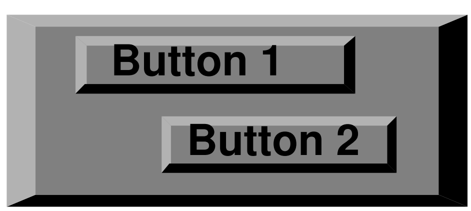


"Text Editor" on page 142



"UVA Logo" on page 148

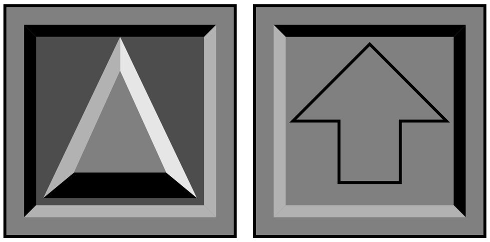
"Scrollable List" on page 138



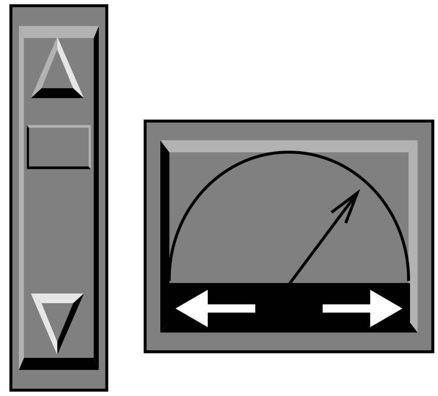
"Bulletin Board" on page 118

"Pulldown Menu" on page 134

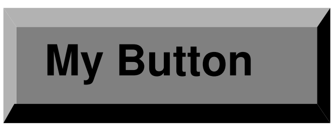
The SUIT Widgets at a Glance



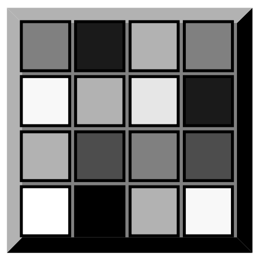
"Arrow Button" on page 114



"Bounded Value" on page 116

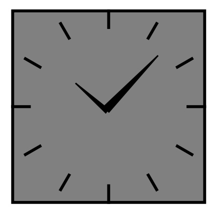


"Button Widget" on page 119

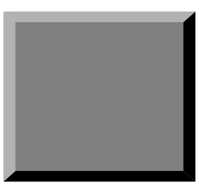


"Color Chips" on page 122

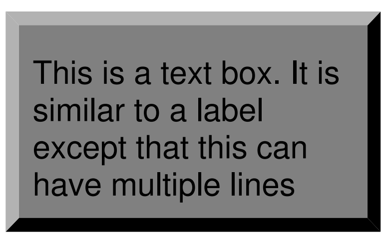
"Label" on page 129



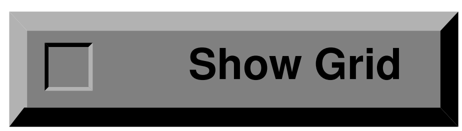
"Clock" on page 121



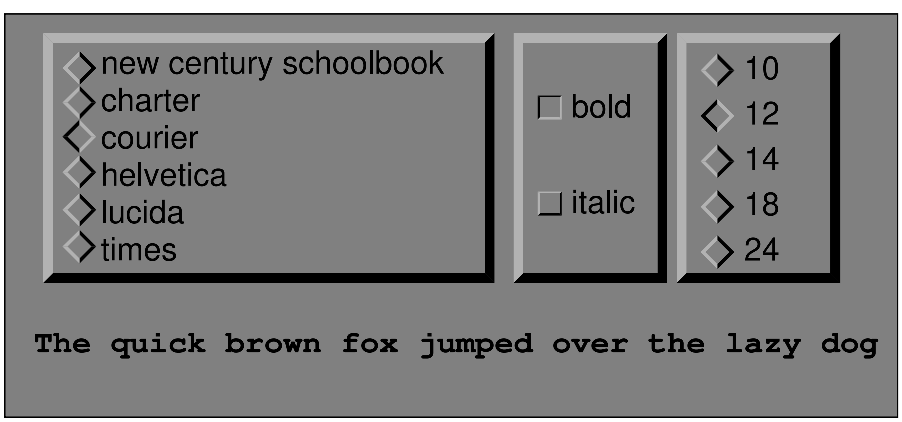
"Place Mat" on page 132



"Text Box" on page 141



"On Off Switch" on page 130



"Font Panel" on page 128

The SUIT Widget Library

Time Functions

`GP_time GP_getCurrentTime()`

Description: This returns the current system time.

See Also: The definition of GP_time on page 34

`time_t GP_timeDifference (GP_time t1, GP_time t2)`

Description: This function computes the difference of the two GP_times and returns the result in milliseconds.

See Also: The definition of GP_time on page 34

`void GP_convertTime (GP_time t, int *hour, int *min, int *second, int *milli)`

Description: This routine converts a GP_time into hours, minutes, seconds and milliseconds. On some systems, it may not be possible to measure time to millisecond accuracy.

See Also: The definition of GP_time on page 34

Advanced GP_text Functions

These functions allow the user to extend the normal set of special characters that are provided.

```
void GP_registerSpecialCharacter (char *character,
                            GP_rectangle (*callback) (GP_point, char*))
```

Description: This creates a new character for the GP_text special characters detailed on page 104.

```
void GP_registerAccent (char *character,
                            GP_rectangle (*callback) (GP_point, char*, char*))
```

Description: This creates a new style for the GP_text special characters detailed on page 104. The callback supplied is a pointer to a function that takes a GP_point (the location of the text) a string (the command that follows the "@" sign) and another string (the string to print).

Example: /* This program registers new styles that let you print red text by saying @red(my text) */

```
GP_rectangle ColorMe (GP_point pt, char *color, char *str) {
    GP_setColor (GP_defColor (color, BLACK_ON_MONO));
    return GP_text (pt, str);
}


void main (int argc, char *argv[]) {
    SUIT_init(argv[0]);
    GP_registerStyle ("red", ColorMe);
    GP_registerStyle ("blue", ColorMe);
    GP_registerStyle ("green", ColorMe);
    GP_registerStyle ("yellow", ColorMe);
    SUIT_beginStandardApplication();
}
```

GP Special Characters and Type Attributes

These codes can be used in the calls to GP_text() and the type in boxes in the SUIT property editor.

Special Characters

Command	Produces
`@(?)`	¿
`@(!)`	¡
`@(ss)`	ß
`@(ae)`	æ
`@(AE)`	Æ
`@(cents)`	¢
`@(pounds)`	£
`@(yen)`	¥
`@(copyright)`	©
`@(restricted)`	®
`@(subsection)`	§
`@(paragraph)`	¶
`@(<<)`	«
`@(>>)`	»
`@(divison)`	÷
`@(times)`	×
`@(+-)`	±

Styles

Style	Command	Short-cut	What this looks like
Italic	`@italic(`*text*`)`	`@i(`*text*`)`	*Example Text*
Bold	`@bold(`*text*`)`	`@b(`*text*`)`	**Example Text**
Underline	`@underline(`*text*`)`	`@u(`*text*`)`	Example Text
Superscript	`@superscript(`*text*`)`	`@+(`*text*`)`	Example Text
Subscript	`@subscript(`*text*`)`	`@-(`*text*`)`	Example Text
Centered	`@center`	@c(text)	Example Text

Accents

Command	Produces
`@'(a)`	á
``@`(a)``	à
`@^(a)`	â
`@:(a)`	ä
`@,(a)`	ą
`@~(a)`	ã

void GP_polyMarkerCoord(int vertexCount, double xlist[], double ylist[])

Description: This draws a series of markers (number = numMarker) at the coordinates specified by the arrays of coordinates. This is particularly useful for placing markers along the vertices of a polyline.

void GP_polyPoint(int vertexCount, GP_point vertices[])

Description: This places points, one pixel big at the coordinates specified by the array of GP_points.

void GP_polyPointCoord(int vertexCount, double xlist[], double ylist[])

Description: This places points, one pixel big at the coordinates specified by the arrays of x, y values.

void GP_rectangleCoord(double x1, double y1, double x2, double y2)

Description: This draws an unfilled rectangle given the coordinates for the corners of the rectangle.

void GP_rectanglePt(GP_point bottom_left, GP_point top_right)

Description: This draws an unfilled rectangle given the two points for the corners of the rectangle.

GP_rectangle GP_text(GP_point pt, char *string)

Description: Places text at the specified GP_point. GP_text supports a small “language” that can be used to apply special attributes to the text. To facilitate the creation of new special characters, this function returns the bounding rectangle of the text just drawn.

Example: **GP_text(pt, “This is @underline(really) fun!”);**

On the screen:

This is <u>really</u> fun

See Also: "GP Special Characters and Type Attributes" on page 104.

void GP_lineCoord(double x1, double y1, double x2, double y2)

Description: This draws a line from the first x,y coordinate pair to the second.

void GP_marker(GP_point)

Description: This draws a marker at the specified GP_point. The shape of the marker is determined by the marker style.

See Also: **GP_setMarkerStyle()** and **GP_setMarkerSize()** on page 91.

void GP_markerCoord(double x, double y)

Description: This draws a marker at the specified coordinates.

See Also: **GP_setMarkerStyle()** and **GP_setMarkerSize()** on page 91.

void GP_pointCoord(double x, double y)

Description: This places a point one pixel big at the specified coordinates.

void GP_polygon(int vertexCount, GP_point vertices[])

Description: This draws an unfilled polygon given an array of GP_points which describe the vertices of the polygon. See the illustration on page 101 for details.

void GP_polygonCoord(int vertexCount, double xlist[], double ylist[])

Description: This draws an unfilled polygon given a two arrays of coordinates (one for x one for y) which describe the vertices of the polygon. See the illustration on page 101 for details.

void GP_polyLine(int vertexCount, GP_point vertices[])

Description: This draws a polyline (a string of connected line segments) given an array of GP_points that describe the endpoints of the lines.

void GP_polyLineCoord(int vertexCount, double xlist[], double ylist[])

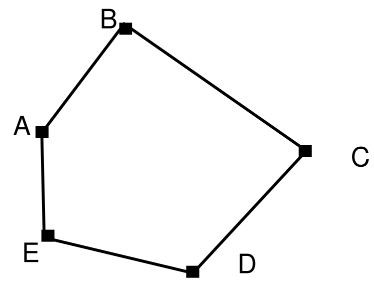
Description: This draws a polyline (a string of connected line segments) given two arrays of coordinates that describe the endpoints of the lines.

void GP_polyMarker(int numMarkers, GP_point vertices[])

Description: This draws a series of markers (number = numMarker) at the coordinates specified by the array of GP_points. This is particularly useful for placing markers along the vertices of a polyline.

`void GP_fillPolygon(int vertexCount, GP_point vertices[])`

Description: This function draws a filled polygon given an array of GP points that describe the vertices of the polygon. The array should include all unique vertices, that is to say, the first coordinate should NOT be the same as the last. This function will close the polygon for you.



vertexCount = 5 (not 6)

vertices are A, B, C, D, E (do not repeat vertex A)

`void GP_fillPolygonCoord(int vertexCount, double xlist[], double ylist[])`

Description: This function draws a filled polygon given the number of vertices to draw and two arrays of GP_points that define those vertices.

Parameters: **`xlist`** and **`ylist`** These are lists of x coordinates and y coordinates respectively.

`void GP_justifyText(char *str, GP_justification just)`

Description: This routine will justify text in the current viewport.

Parameters: **`just:`** the kind of justification wanted. Allowed values are:

JUSTIFY_BOTTOM_LEFT	**JUSTIFY_BOTTOM_CENTER**	**JUSTIFY_BOTTOM_RIGHT**
JUSTIFY_CENTER_LEFT	**JUSTIFY_CENTER**	**JUSTIFY_CENTER_RIGHT**
JUSTIFY_TOP_LEFT	**JUSTIFY_TOP_CENTER**	**JUSTIFY_TOP_RIGHT**

`void GP_justifyTextInRectangle (char *text, GP_justification just, GP_rectangle r)`

Description: This routine will justify text inside the given rectangle.

Parameters: **`just:`** the kind of justification wanted. Allowed values are:

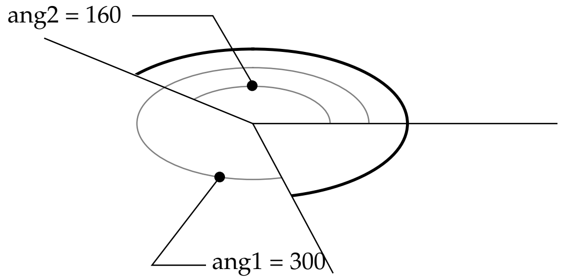
JUSTIFY_BOTTOM_LEFT	**JUSTIFY_BOTTOM_CENTER**	**JUSTIFY_BOTTOM_RIGHT**
JUSTIFY_CENTER_LEFT	**JUSTIFY_CENTER**	**JUSTIFY_CENTER_RIGHT**
JUSTIFY_TOP_LEFT	**JUSTIFY_TOP_CENTER**	**JUSTIFY_TOP_RIGHT**

`void GP_line(GP_point pt1, GP_point pt2)`

Description: This routine will draw a line from p1 to p2.

`void GP_ellipseArc(GP_rectangle rect, double ang1, double ang2)`

Description: This function draws an arc. The arc may be circular or elliptical. The angles are given in degrees and must lie between 0 and 360, where 0 degrees is defined by the positive part of the X axis. The arc sweeps around counter-clockwise from ang1to ang2.



`void GP_ellipseCoord(double x1, double y1, double x2, double y2)`

Description: This function draws an unfilled ellipse given the four coordinates of the bounding rectangle.

`void GP_fillEllipse(GP_rectangle rect)`

Description: The rectangle passed in to this function is the bounding rectangle of the filled ellipse.

`void GP_fillEllipseArc (GP_rectangle rect, double ang1, double ang2)`

Description: This is the same function as **`GP_ellipseArc()`**, except that the arc is filled.

`void GP_fillEllipseCoord(double x1, double y1, double x2, double y2)`

Description: This function draws a filled ellipse given the four coordinates of the bounding rectangle.

`void GP_fillRectangleCoord(double x1, double y1, double x2, double y2)`

Description: This function draws a filled rectangle given the four coordinates of the rectangle.

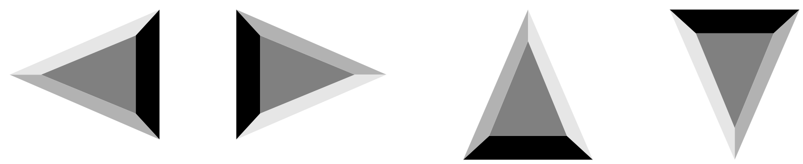
`void GP_fillRectanglePt(GP_point bottom_left, GP_point top_right)`

Description: This function draws a filled rectangle given the two points for the corners of the rectangle.

`void GP_fillRectangle(GP_rectangle rect)`

Description: This function draws a filled rectangle given a rectangle data object to draw.

The beveled triangle calls create triangles that point in various directions. Notice that the parameters are in a particular order, where p1 points in the direction of the arrow and p2, p3 are in a clockwise direction.



```
void GP_beveledTriangleEast(GP_point p2, GP_point p3, GP_point p1,
                              GP_color color, boolean raised, int thickness)
```

```
void GP_beveledTriangleNorth (GP_point p3, GP_point p2, GP_point p1,
                        GP_color color, boolean raised, int thickness)
```

```
void GP_beveledTriangleSouth (GP_point p1, GP_point p2, GP_point p3,
                        GP_color color, boolean raised, int thickness)
```

```
void GP_beveledTriangleWest (GP_point p1, GP_point p2, GP_point p3,
                        GP_color color, boolean raised, int thickness)
```

`void GP_drawRectangle(GP_rectangle rect)`

Description: This function draws an unfilled rectangle.

`void GP_ellipse(GP_rectangle rect)`

Description: This draws an unfilled ellipse, where the rectangle is the bounding rectangle of the ellipse.

Drawing Primitives

The following functions place drawing primitives on the current canvas, in the current color, in the current line style, with the current bitmap (if applicable) using the current drawing mode. In short, all the functions in this section are affected by the current graphics state. (See page 89 for a discussion of the current graphics state).These functions and their parameters are, for the most part, self-explanatory.

`void GP_beep(void)`

Description: This causes the system to emit a short beep of fixed duration.

`void GP_beveledBorder (rectangle box, GP_color color, boolean raised, int width)`

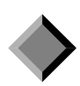
Description: This draws a beveled hollow rectangle.

`void GP_beveledBox (rectangle box, GP_color color, boolean raised, int width)`

Description: This draws a beveled filled rectangle.

`void GP_beveledDiamond (rectangle box, GP_color color, boolean raised, int width)`

Description: This draws a beveled filled diamond. Examples of these can be seen in the SUIT radio buttons.



Color, Pattern, Cursor, And Font Table Functions

These functions allow control of various tables which the graphics package maintains for these attributes. The tables contain the options for each attribute. New choices may be added to the colors and patterns.

```
void GP_describeColor (GP_color c, unsigned short *red, unsigned short *green,
                        unsigned short *blue)
```

Description: This function takes a GP_color as a parameter and Returns through three integer pointers, the red, green and blue components of that GP_color in the range (0-65536).

```
void GP_replaceColor (GP_color oldColor, GP_color newColor)
```

Description: This function replaces one color in the color table with another. Anything painted in the old color will change to the new color.

```
void GP_loadColorTable(int start, int count,unsigned short red[]
                                            unsigned short green[],
                                            unsigned short blue[])
```

Description: This loads the color table at the given starting index with the colors given by the arrays of unsigned shorts which represent the red, green and blue components (range 0-65536) of the colors you are loading into the table.

```
void GP_loadCommonColor(int entry, char *colorName)
```

Description: Loads a color into the given color table slot by name.

See Also: Names for the colors on page 84.

Canvas Control Functions

These functions are primarily useful to the support programs. They are used to set up the initial drawing area, and can be used to perform more complex graphical feats.

`canvasID GP_createCanvas(int width, int height)`

Description: This function, given the height and width of a canvas, returns the canvasID of a newly created canvas and sets that canvas as the active canvas.

`void GP_deleteCanvas(canvasID whichCanvas)`

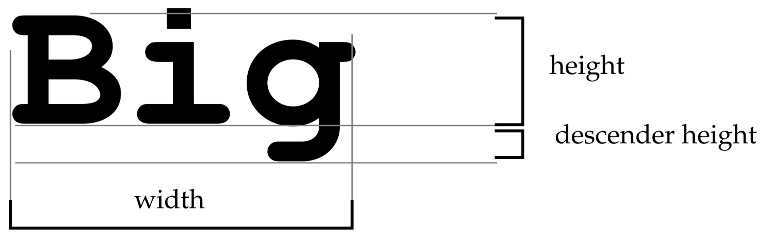
Description: Deletes a canvas that was created with **`GP_createCanvas()`**.

`void GP_useCanvas(canvasID whichCanvas)`

Description: Sets the given canvas to be the current canvas. Subsequent GP calls will be drawn on this canvas.

void GP_inquireTextExtent(char *string, double *width, double *height, double *decenderHeight)

Description: Finds the physical size (in world coordinates) of the string using the current font. The parameters passed in are pointers to doubles, which, after the call, contain the width of the string, the distance from the baseline to the highest ascender and the distance from the baseline to the lowest descender. Note: this is different from the traditional usage of these terms.



void GP_inquireTextExtentWithoutMapping (char *string, int *width, int *height, int *descenderHeight)

Description: This function is like **SUIT_inquireTextExtent()**, except that the dimensions of the text remain in pixels.

int GP_numColorsAllocated (void)

Description: This routine tells you how many colors GP has currently allocated.

canvasID GP_inquireActiveCanvas(void)

Description: This function returns the canvasID of the canvas that is currently in use.

rectangle GP_inquireCanvasExtent(canvasID whichCanvas)

Description: This function returns a rectangle whose dimensions are those of the given canvas.

int GP_inquireCanvasDepth(void)

Description: This function returns the number of bit planes of the current canvas. Use this function to determine if the system currently running SUIT is color or monochrome. (Monochrome systems will have a canvas depth of 1. Color systems will have a canvas depth greater than 1). Equivalently, you can ask if the system is monochrome with:

```
if (BILEVEL_DISPLAY) { ... }
```

void GP_inquireCanvasSize(canvasID whichCanvas, int *width, int *height)

Description: This function is like GP_inquireCanvasExtent in that it gives the size of the canvas in question, but this function produces the dimensions in the form of two integers. Remember to pass in pointers to integers as the parameters to this function.

```
void GP_inquireColorTable(int start, int count, unsigned short red[],
                                                unsigned short green[],
                                                unsigned short blue[])
```

Description: Given a starting index into the color table and a number of colors to get, this function places into the arrays passed in as parameters the red, green and blue components of the colors in those slots in the color table. Be sure to allocate enough room in these arrays.

Example:

```
/* allocate memory for the arrays */
unsigned short red[10];
unsigned short green[10];
unsigend short blue[10];

GP_inquireColorTable(0, 10, red, green, blue);
```

Getting Graphics Attributes

A number of functions are included here which allow programs to gain information about the current status of the graphics package.

`int GP_getColorIndex(char *name)`

Description: This routine accepts a color name and returns the color's index in the color table

---

`char *GP_getColorName(int index)`

Description: Accepts an index into GP's color table and returns that color's name as an ASCII string. This string is a pointer into GP's internal data structures, and therefore, you must not alter it in any way. For more information, see "Manipulating Strings in SUIT" on page 76.

`GP_color GP_getDepthColor (GP_color clr)`

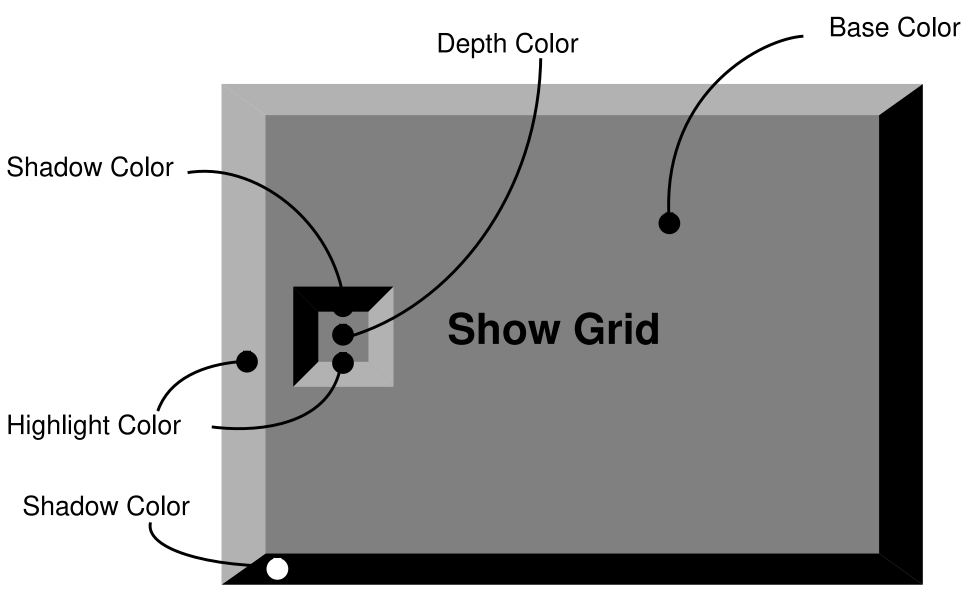
Description: Returns the appropriate motif depth color, given a base color. The depth color is the color one sees in a button during a mouse down event, when border raised is false.

`GP_color GP_getHighlightColor (GP_color clr)`

Description: Returns the appropriate motif highlight color, given a base color.

`GP_color GP_getShadowColor (GP_color clr)`

Description: Returns the appropriate motif shadow color, given a base color.



void GP_setPenBitmapPattern(int value)

Description: This specifies the bitmap pattern to be used on subsequent calls that use either **BITMAP_-PATTERN_TRANSPARENT** or **BITMAP_PATTERN_OPAQUE** fill styles.

See Also: **fill styles** page 31

void GP_setPenPixmapPattern(int value)

Description: This specifies the bitmap pattern to be used on subsequent calls that use drawing fill style **PIXMAP_PATTERN**.

See Also: **fill styles** page 31

void GP_setPenStyle(int fill_style)

Description: The pen style controls the way lines will appear on the screen. Drawing an entity in SOLID fill style replaces existing pixels with a solid flood in the current color. **PIXMAP_PATTERN** replaces pixels with the pattern for the currently defined pixmap pattern. **BITMAP_PATTERN_TRANSPARENT** uses the currently defined bitmap pattern, and considers the "background" color pixels in the pattern to be transparent; pixels that were on the screen will show through. **BITMAP_PATTERN_OPAQUE** replaces the existing region with the existing bitmap. "Background" colors in the bitmap are considered opaque and will replace existing screen pixels.

Legal Values:
SOLID
PIXMAP_PATTERN
BITMAP_PATTERN_TRANSPARENT
BITMAP_PATTERN_OPAQUE

void GP_setWindow (GP_rectangle newWorld)

Description: Sets the world coordinate window for drawing. SUIT calls this routine automatically before each widget's paint procedure, using the value of the widget's window property as the parameter to the function.

void GP_setWriteMode(int write_mode)

Description: Denotes the manner in which pixels are drawn to the screen.

Legal Values:
WRITE_REPLACE **WRITE_XOR**
WRITE_OR **WRITE_AND**

void GP_setViewport (rectangle newView)

Description: This sets the viewport before drawing graphics. SUIT calls this routine automatically before each widget's paint procedure, using the value of the widget's viewport property as the parameter to the function.

`void GP_setInputMode(int device, int mode)`

Description: This function sets a given device into a given mode for collecting data.

Legal Values: Devices:

NO_DEVICE	
KEYBOARD	
LOCATOR	This is the mouse

Modes:

INACTIVE	turns the device off
SAMPLE	device returns the state when asked
EVENT	device reports state when state changes

`void GP_setLineStyle(int line_style)`

Description: This function sets the line style that will be used on the next call to any of the primitive drawing functions.

Legal Values: **CONTINUOUS**
DASHED
DOTTED
DOT_DASHED

`void GP_setLineWidth(int newValue)`

Description: This call sets the line width for subsequent graphics calls.

Rarely Used

`void GP_setLocatorMeasure(point position)`

Description: This function moves the mouse pointer to the specified point. Use this call with restraint, as “warping” the cursor under program control is considered a bad user interface technique.

`void GP_setMarkerStyle(int marker_style)`

Description: This sets the style of marker to be used on the next call to any of the GP_marker() calls.

Legal Values: **MARKER_CIRCLE**
MARKER_SQUARE
MARKER_X

See Also:

GP_setMarkerSize()	page 91
GP_marker()	page 102

`void GP_setMarkerSize(int marker_size)`

Description: This sets the size of the markers (more exactly, the size is that of the bounding box of the marker in pixels) to be used on the next call to any of the **GP_marker()** calls.

See Also:

GP_setMarkerStyle()	page 91
GP_marker()	page 102

void GP_setColor(GP_color clr)

Description: This sets the current color to the specified GP_color.

void GP_setCursor(int cursorType)

Description: This function sets the current cursor shape.

Legal Values:

STANDARD_CURSOR	the regular left pointing arrow
PIRATE_CURSOR	skull and crossbones: seen when destroying widgets interactively
WATCH_CURSOR	a wristwatch for when the user is made to wait
PROMPT_CURSOR	a left pointing finger
RIGHT_ARROW_CURSOR	a right pointing arrow

void GP_setFillBitmapPattern(int value)

Description: This sets the current bitmap pattern. This only has an effect if the current fill style is **BITMAP_PATTERN_TRANSPARENT** or **BITMAP_PATTERN_OPAQUE.**

See Also: **GP_setFillStyle()** page 90

void GP_setFillPixmapPattern(int value)

Description: This sets the current pixmap pattern. This only has an effect if the current fill style is **PIXMAP_PATTERN**.

See Also: **GP_setFillStyle()** page 90

void GP_setFillStyle(int draw_style)

Description: The fill style controls the way filled polygons, rectangles and ellipses will appear on the screen. Drawing an entity in SOLID fill style replaces existing pixels with a solid flood in the current color. PIXMAP_PATTERN replaces pixels with the pattern for the currently defined pixmap pattern. BITMAP_PATTERN_TRANSPARENT uses the currently defined bitmap pattern, and considers the “background” color pixels in the pattern to be transparent; pixels that were on the screen will show through. BITMAP_PATTERN_OPAQUE replaces the existing region with the existing bitmap. “Background” colors in the bitmap are considered opaque and will replace existing screen pixels.

Legal Values:
SOLID
PIXMAP_PATTERN
BITMAP_PATTERN_TRANSPARENT
BITMAP_PATTERN_OPAQUE

int GP_setFont(GP_font newfont)

Description: This sets the value of the current font.

See Also: **GP_defFont()** on page 85.

Setting and Getting the Graphics State

The graphics state is defined as being made up of the following attributes:

- GP_font
- world coordinate window
- screen coordinate viewport
- write mode
- clipping rectangle
- line style
- line width
- marker size
- marker style
- foreground and background colors
- plane mask
- fill style
- pen style
- pixmap and bitmap ID's

The next two functions store and retrieve the graphics attributes to and from a stack. After pushing the current set of attributes on the stack, you can set your own drawing attributes (e.g. foreground color), draw the required entities, and pop the attributes off the stack, leaving the state exactly the way you found it. For example, you can write a subroutine that pushes the graphics state, sets the color to red, draws a line, and pops the graphics state, without having to worry about all subsequent entities coming out red.

SUIT automatically pushes the graphics state before executing a paint procedure and pops the graphics state afterward, so it is rare that a user should ever need to call these functions.

Rarely Used

`void GP_popGraphicsState(void)`

Description: This function restores the global graphics state to the value that was most recently pushed on to the stack with **`GP_pushGraphicsState()`**. See the discussion above.

Rarely Used

`void GP_pushGraphicsState(void)`

Description: This function saves the current graphics state attributes (see above) on a stack so that the state can be temporarily altered, then restored to its previous value with **`GP_popGraphicsState()`**. See the discussion above.

Example:

```
/* save the current graphics state */
        GP_pushGraphicsState();
/* change the state -- set the current color to some color */
        GP_setColor (GP_defColor("red", TRUE));
/* draw a line in that color */
        GP_line(0.0, 0.0, 0.1, 0.1);
/* restore the graphics state to the way we found it */
        GP_popGraphicsState();
```

Rarely Used

`void GP_setClipRectangle(rectangle rect)`

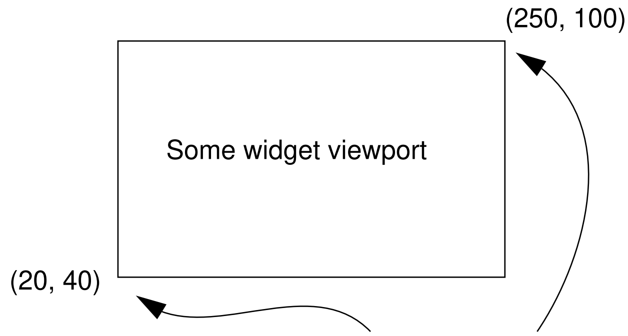
Description: Sets the clipping rectangle to be the given rectangle. There is usually no need to call this function, as SUIT calls this function for you before executing a widget's paint procedure to ensure that under "normal" circumstances, a widget will not draw outside of its own viewport.

See Also: The property called **CLIP_TO_VIEWPORT**page 110

Mapping and Unmapping Widths and Heights

See Also: "Mapping and Unmapping Rectangles and Points" on page 86

These functions are to be used when you need to compute the ***differences*** of GP coordinates. The following illustration shows the differences between mapping widths and heights and mapping X's and Y's. Essentially, these functions convert **DISTANCES** from one coordinate system to another.



Assume the Window of this widget is (1.0, 1.0) to (2.0, 2.0)

```
GP_mapX(1.0) = 20                   /* just converts the coordinates */
GP_mapY(1.0) = 40                   /* just converts the coordinates */
GP_mapWidth(1.0) = 230              /* GP x DISTANCE of 1.0 is 230 */
GP_mapHeight(1.0) = 60              /* GP y DISTANCE of 1.0 is 60 */
GP_unMapHeight(30) = 0.5            /* DISTANCE of 30 Y pixels = 0.5 in GP coords */
```

Rarely Used

`int GP_mapHeight (double height)`

Description: This function returns the height in pixels of the height in world coordinates given.

Rarely Used

`int GP_mapWidth (double width)`

Description: This function returns the width in pixels of the width in world coordinates given.

Rarely Used

`double GP_unMapHeight (int height)`

Description: This function returns the height in world coordinates of the pixel height given.

Rarely Used

`double GP_unMapWidth (int width)`

Description: This function returns the width in world coordinates of the pixel width given.

`GP_point GP_unMapPoint (point pt)`

Description: This function takes a point in the screen coordinate system and returns the corresponding GP_point in world coordinates.

`GP_rectangle GP_unMapRectangle(rectangle rect)`

Description: This function takes a rectangle in the screen coordinate system and returns the corresponding GP_rectangle in world coordinates.

`double GP_unMapX (int x)`

Description: This function converts an x coordinate in screen coordinates to the corresponding x coordinate in world coordinates for the current viewport.

`double GP_unMapY (int y)`

Description: This function converts a y coordinate in screen coordinates to the corresponding y coordinate in world coordinates for the current viewport.

Mapping and Unmapping Rectangles and Points

These functions are useful in converting from the floating point notation which GP uses to the integer coordinate system on the screen, which SUIT uses. GP_map functions convert from floating point to screen coordinates, and GP_unMap convert from screen to floating point coordinates.This separation between the coordinates on the screen and the World coordinates allows SUIT to resize the widgets without loss of resolution. The viewport being mapped to is the viewport of the current widget.

See Also: The following functions are more helpful for higher level mapping functions.

`SUIT_mapScreenToViewport()`	page 52
`SUIT_mapViewportToScreen()`	page 52
`SUIT_mapRelativeLocationToScreen()`	page 52
`SUIT_mapScreenToRelativeLocation()`	page 52
`SUIT_mapPointToObject()`	page 52
`GP_setViewport()`	page 92.
`GP_setWindow()`	page 92.

For more information, see "Windows, Viewports, and Rectangles" on page 17.

	World Coordinates	Screen Coordinates
Coordinate Type	floating point	integer
Rectangle Type	GP_rectangle or SUIT_window	rectangle or SUIT_viewport
Point Type	GP_point	point

Mapping →

Unmapping ←

`point GP_mapPoint(GP_point pt)`

Description: This function takes a GP_point in world coordinates and returns the corresponding point in the current viewport.

`rectangle GP_mapRectangle(GP_rectangle r)`

Description: This function takes a GP_rectangle in the world coordinate system and returns the corresponding rectangle in the current viewport.

`int GP_mapX (double x)`

Description: This function converts an x coordinate in world coordinates to the corresponding x coordinate in screen coordinates for the current viewport.

`int GP_mapY (double y)`

Description: This function converts a y coordinate in world coordinates to the corresponding y coordinate in screen coordinates for the current viewport.

GP_color GP_defColorRGB (unsigned short red, unsigned short green, unsigned short blue, boolean blackOnMonocrome);

Description: This returns a GP_color. The information supplied here is the red, green and blue components of the color to be defined in the range (0-65536). This function does not allocate colors in the color table, to do so requires a call to **`GP_loadCommonColor()`.**

See Also: Definition of the GP_color type on page 33.

GP_font GP_defFont(char *family, char *style, double pointSize)

Description: To define a GP_font from family, style and point size information.

Parameters: **`family:`** The font family. Allowed values here include: "times". "courier", "helvetica", "lucida", and "new century schoolbook".

`style:` The font style. Allowed values here are: bold, italic, bold italic and the null string for normal (or roman).

`pointSize:` The size of the text in points (1/72 inch).

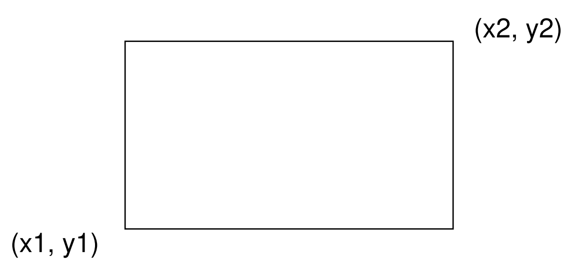
See Also: **`GP_setFont()`** page 90

GP_point GP_defPoint(double x, double y)

Description: To define a GP_point given its X and Y coordinates.

GP_rectangle GP_defRectangle(double x1, double y1, double x2, double y2)

Description: To define a GP_rectangle given the coordinates of the lower left hand corner (x1, y1) and the coordinates of the upper right hand corner (x2, y2).



The corners of a GP rectangle

Type Definition Functions

`GP_color GP_defColor(char *name, boolean black)`

Description: This returns a GP_color. Notice that only the only information supplied here is the name and whether the color appears as black on a monochrome screen. This does not allocate colors on the color table, to do so requires a call to **`GP_loadCommonColor()`** or a call to **`SUIT_setColor()`** with a new color which creates the color and loads the color table as well.

See Also: Definition of the GP_color type on page 33.

Legal Values: In addition to the color "**`peru`**", the following color names can be used:

aliceblue	**green**	**navajowhite**
antiquewhite	**greenyellow**	**navy**
aquamarine	**grey**	**navyblue**
azure	**honeydew**	**oldlace**
beige	**hotpink**	**olivedrab**
bisque	**indianred**	**orange**
black	**ivory**	**orangered**
blanchedalmond	**khaki**	**orchid**
blue	**lavender**	**palegoldenrod**
blueviolet	**lavenderblush**	**palegreen**
brown	**lawngreen**	**paleturquoise**
burlywood	**lemonchiffon**	**palevioletred**
cadetblue	**lightblue**	**papayawhip**
chartreuse	**lightcoral**	**peachpuff**
chocolate	**lightcyan**	**pink**
coral	**lightgoldenrod**	**plum**
cornflowerblue	**lightgoldenrodyellow**	**powderblue**
cornsilk	**lightgray**	**purple**
cyan	**lightgrey**	**red**
darkgoldenrod	**lightpink**	**rosybrown**
darkgreen	**lightsalmon**	**royalblue**
darkkhaki	**lightseagreen**	**saddlebrown**
darkolivegreen	**lightskyblue**	**salmon**
darkorange	**lightslateblue**	**sandybrown**
darkorchid	**lightslategray**	**seagreen**
darksalmon	**lightslategrey**	**seashell**
darkseagreen	**lightsteelblue**	**sienna**
darkslateblue	**lightyellow**	**skyblue**
darkslategray	**limegreen**	**slateblue**
darkslategrey	**linen**	**slategray**
darkturquoise	**magenta (of course)**	**slategrey**
darkviolet	**maroon**	**snow**
deeppink	**mediumaquamarine**	**springgreen**
deepskyblue	**mediumblue**	**steelblue**
dimgray	**mediumorchid**	**tan**
dimgrey	**mediumpurple**	**thistle**
dodgerblue	**mediumseagreen**	**tomato**
firebrick	**mediumslateblue**	**turquoise**
floralwhite	**mediumspringgreen**	**violet**
forestgreen	**mediumturquoise**	**violetred**
gainsboro	**mediumvioletred**	**wheat**
ghostwhite	**midnightblue**	**white**
gold	**mintcream**	**whitesmoke**
goldenrod	**mistyrose**	**yellow**
gray	**moccasin**	**yellowgreen**

GP Function Calls

Rarely Used

```
void SUIT_registerClass(char *className,
                    SUIT_object (*CreateProc)(char *objName), char* helpText)
```

Description: This function is used for creating SUIT_objects interactively with SUIT-N or by exporting. As every SUIT_object belongs to a class, to create a new widget interactively requires that you have first registered that class with SUIT. Registering a class with SUIT tells SUIT the name of the class and how to create a widget of that class.

Parameters: **`className:`** The name of the new class of be created (e.g. "bounded value", "radio buttons").

`SUIT_object (*CreateProc) (char *objName):` This is the widget creation procedure. It takes as a parameter, the name of the new object to be created. Remember: all SUIT widget names must be unique. The CreateProc is responsible for calling **`SUIT_createObject()`** and making all the necessary **`SUIT_addDisplayToObject()`** calls. This CreateProc will return the SUIT_object created.

`helpText`: A string that describes the class. This text appears in a dialog box when the user drags a widget over the info icon in the interactive property editor.

See Also: *Frequently Asked Questions: How Do I Create My Own Widgets?* on page 21.

void SUIT_selectObject(SUIT_object object)

Description: This routine selects the given widget, as if the user had typed SUIT-s over the widget.

void SUIT_deselectObject(SUIT_object object)

Description: This routine selects the given widget, as if the user had typed SUIT-s over the selected widget.

SUIT_object SUIT_name(char *name)

Description: This function will return a SUIT_object given only its name. All object names in SUIT are unique. This can be used to access any SUIT_object whose name you know, without having to pass it as a parameter.

Remarks: **name:** The name of the SUIT object.

Example:

```
void MakeAButton() {

/* create a button with the SUIT name "mybutton" */
mywidget = SUIT_createButton ("mybutton", buttonCallback);
}

void UseTheButton() {
/* note that this function takes no parameters. */
/* we can still access the button,by asking for it by name. Here we
will change the label of the button.*/

SUIT_setText(SUIT_name("mybutton"), LABEL, "new label");

}
```

void SUIT_registerHelp (char *className, char *propName, char *helpText)

Description: This routine adds a help string to a particular property of a widget of a particular class. These help strings can be viewed using the property editor's Info facility. This function does not make a copy of the help text and therefore should not be altered once it is registered, though usually, this text comes in the form of a string constant.

Example:

```
SUIT_registerHelp("button", BORDER_WIDTH,
                         "The width of the border in pixels");
```

See Also: **SUIT_getHelp()** on page 79.

SUIT_object SUIT_copyObject(SUIT_object objectToCopy, char* nameForNewWidget)

Description: This will create an exact copy of the given SUIT_object. The new widget's properties are exactly the same, except for the widget's name (which is given as a parameter) and the viewport, which is the same as the source widget's viewport, offset by 5 pixels.

---

void SUIT_destroyObject(SUIT_object obj)

Description: This will cause the SUIT_object to be marked as "to be destroyed." The actual destruction of the widget happens at the very end of the SUIT main loop when SUIT has finished painting and is ready to check for more user input.

Remarks: Calling this function can have dire effects (dangling references) if other portions of your program call **`SUIT_name(`**`<some string>`**`)`** with the name of a destroyed object or use pointers to destroyed SUIT_objects. Use with caution.

---

char *SUIT_getHelp(char *className, char *propName)

Description: This routine returns the help string (if any) that was registered for the property.

See Also: **`SUIT_registerHelp()`** on page 80.

---

void SUIT_iterateOverObjects(void (*callback) (SUIT_object))

Description: This routine calls the given function once for each object in the interface (parents and their children, but not their employees).

Example:

```
/* to print out the names of all objects when button is hit */

void dumpObjectName(SUIT_object o){
     printf("name: %s\n", OBJECT_NAME(o) );
}

void myDoIterate(SUIT_object o) {
     SUIT_iterateOverObjects(dumpObjectName);
}

void main (int argc, char *argv[]) {
     SUIT_init(argv[0]);
     SUIT_createButton("do the iteration", myDoIterate);
     SUIT_beginStandardApplication ();
}
```

---

void SUIT_makeObjectTemporary(SUIT_object object)

Description: This routine flags the object as one whose properties are NOT written to the ".sui" file.

See Also: **`OBJECT_PERMANENT()`** on page 77.

---

void SUIT_makeObjectPermanent(SUIT_object object)

Description: This routine flags the object as one whose properties ARE written to the ".sui" file. By default, all objects are permanent, so you should only need to call this function if you've made the object temporary.

See Also: **`OBJECT_PERMANENT()`** on page 77.

---

void SUIT_cycleObject(SUIT_object object)

Description: This routine sets the ACTIVE_DISPLAY property to the next available value, as if the user had typed SUIT-c over the widget.

SUIT_viewport OBJECT_VIEWPORT(SUIT_object obj)

Description: This function returns the current viewport of a given SUIT object. Note that this is a macro for **SUIT_getViewport(obj, VIEWPORT)**.

See Also: **SUIT_getObjectSize()** on page 51.

SUIT_window OBJECT_WINDOW(SUIT_object obj)

Description: This function returns the current window of a given SUIT object. Note that this is a macro for **SUIT_getWindow(obj, WINDOW)**.

Rarely Used

```
SUIT_viewport SUIT_adjustForSpringiness (SUIT_viewport parents_old_vp,
                                         SUIT_viewport parents_new_vp,
                                         SUIT_viewport childs_old_viewport,
                                         SUIT_springiness spr)
```

Description: This is a rarely called function as SUIT will manage the resizing of children automatically. The function returns the properly adjusted viewport of a child widget taking into account the value of the parent's old and new viewports, the old viewport of the child object in question and the value of the springiness property possessed by the child.

SUIT_object SUIT_dummyObjectInClass(char *className)

Description: Returns a dummy object of the given class. Used in setting properties at the class level when there is no object of that class.

Example:

```
SUIT_deluxeSetBoolean(SUIT_dummyObjectInClass("bounded value"),
                      HAS_BORDER, FALSE, CLASS);
```

Rarely Used

Pointer SUIT_convertType (Pointer value, char *fromType, char *toType)

Description: This function converts the value of one property into another using the fact that SUIT knows how to convert all of its registered types into and out of ASCII. The pointer returned is a pointer to safely allocated memory.

Parameters:
- **value**: a pointer to the value being converted.
- **fromType** the character string for the type of the property being converted.
- **toType** the character string for the type being converted to.

Allowed values for types are: "boolean", "double", "DynArray", "GP_color", "GP_font", "int", "SUIT_object", "SUIT_functionPointer", "SUIT_springiness", "SUIT_enum", "SUIT_textList", "text", "viewport", and "window".

Pointer SUIT_copyData (Pointer ptr, int len)

Description: This function will create data that is local and safely malloc-ed. (Really a **malloc()** followed by a **memcpy()**) This is useful for creating objects that can be the return values of functions.

Parameters:
- **ptr:** A Pointer to the data to be copied.
- **len:** The sizeof() the data to be copied.

Miscellaneous Functions

`char* OBJECT_CLASS(SUIT_object obj)`

Description: Returns the class of a SUIT_object in the form of a text string (e.g. “bounded value”, “radio button”, “label”).

WARNING: This macro DOES NOT ALLOCATE MEMORY FOR THIS STRING. You are being handed a pointer to a string that SUIT is keeping for internal use. You must therefore NEVER assign a value to this string or use **`strcat()`** to append characters to this string. To change the value of this string, you must first use **`strdup()`** or **`SUIT_copyString()`** to make a safe copy.

`char* OBJECT_NAME(SUIT_object obj)`

Description: Returns a pointer to the name (an ASCII string) of a given SUIT object.

WARNING: This macro DOES NOT ALLOCATE MEMORY FOR THIS STRING. You are being handed a pointer to a string that SUIT is keeping for internal use. You must therefore NEVER assign a value to this string or use **`strcat()`** to append characters to this string. To change the value of this string, you must first use **`strdup()`** or **`SUIT_copyString()`** to make a safe copy.

`boolean OBJECT_OPEN(SUIT_object obj)`

Description: Returns the boolean flag in the SUIT_object structure that denotes whether the object is open (in a hierarchical sense). SUIT paints open objects with an “open border” around the inside of the widget. This flag can be read and written to.

Example:

```
/* test the open flag */
if (! OBJECT_OPEN(obj) )
      printf ("object is not open\n");

/* set the flag */
OBJECT_OPEN(obj) = TRUE;
```

`boolean OBJECT_SELECTED(SUIT_object obj)`

Description: Returns the boolean flag in the SUIT_object structure that denotes whether the object is selected. This flag can be read and written to.

Example:

```
/* test the selected flag */
if (! OBJECT_SELECTED(obj) )
      printf ("object is not selected\n");

/* set the flag */
OBJECT_SELECTED(obj) = TRUE;
```

`boolean OBJECT_PERMANENT(SUIT_object obj)`

Description: Returns the boolean flag in the SUIT_object structure that denotes whether the object is permanent. Non-permanent objects are not saved to the ".sui" file. This flag can be read and written to.

Example:

```
/* test the permanence flag */
if (! OBJECT_PERMANENT(obj) )
      printf ("object is not written to sui file\n");

/* set the flag */
OBJECT_PERMANENT(obj) = TRUE;
```

Manipulating Strings in SUIT

A WARNING ABOUT STRINGS IN SUIT:

Strings in SUIT appear in many guises: property names, object names, and text properties just to name a few. In all cases where a SUIT function passes back one of these types or a (char *), you must realize that YOU ARE NOT GETTING A COPY OF THE STRING, YOU ARE GETTING A POINTER TO THE STRING. If you intend to alter the string in some way, you must first make a copy of the string with **`SUIT_copyString()`** or **`strdup()`**.

`int SUIT_caseInsensitiveCompare(char *a, char *b)`

Description: This function returns 0 if the strings are the same, a negative number if **`a`** is lexicographically less than **`b`** and a positive number if **`a`** is lexicographically greater than b. This comparison is not case sensitive, but otherwise is like the C **`strcmp()`** function.

See Also: **`SUIT_stringsMatch()`** page 76

`boolean SUIT_caseInsensitiveMatch(char *a, char *b)`

Description: This function returns TRUE if the strings are the same, FALSE if they are not. This comparison is not case sensitive.

See Also: **`SUIT_stringsMatch()`** page 76

`int SUIT_stringContains(char *s1, char *s2)`

Description: This function will return a positive number if s1 is a substring of s2, indicating the character position (zero indexed) where the second string begins inside the first string. The function returns a -1 if the second string is not a substring of the first. The comparison is case *insensitive*

Examples:

```
SUIT_stringContains("doghouse","house") ;    returns   3
SUIT_stringContains("cat","cat");            returns   0
SUIT_stringContains("cat","dog");            returns   -1
SUIT_stringContains("DoGhOuSe","HoUsE");     returns   3
```

`boolean SUIT_stringsMatch(char *s1, char *s2)`

Description: This function will return TRUE if the strings are the same, FALSE if they are not. This comparison is case sensitive.

Parameters: **`s1`** and **`s2:`** The two strings that are to be compared.

`char* SUIT_copyString(char *str)`

Description: This function will create strings that are local and safely malloced. This is the same as the common C function **`strdup()`** found on many platforms.

Remarks: **`str:`** The string that is to be copied.

Dragging Functions

Note: When you call these routines the mouse should be down.

`point SUIT_dragText(char *text)`

Description: This routine XORs the given text around the screen. When you let up the mouse button, it returns the point where the mouse went up. The text's lower lefthand corner is drawn at the cursor.

`point SUIT_dragTextWithOffset(char *text, int x, int y)`

Description: Like the above but allows an offset from the cursor. The x and y values passed in represent the x and y offset of the text with respect to the cursor (so positive x and y values will put the text to above and to the right of the cursor).

`point SUIT_drag(void (*graphicsCallback)(point))`

Description: This allows arbitrary graphics to be dragged around by the user. The function passed in as a parameter makes SRGP calls that draw the graphics. The graphics specified will be drawn in XOR mode with dasehd lines. The function that draws the graphics looks like:

```
void DrawThatFunkyThang (point p)
     {
     /* make SRGP graphics calls here */
     }
```

The point passed in as a parameter is in pixel cordinates and represents the current location of the cursor.

Example: See the example programs drag1.c and drag2.c

Interest Functions

`void SUIT_registerInterest(SUIT_object obj, SUIT_objectInterestCallback fcn)`

Description: This function registers with SUIT an interest callback function. This function will get called any time any property of the given object changes. This can be useful for imposing constraints on a property of an object.

An interest callback function looks like:

```
void (*SUIT_objectInterestCallback)(SUIT_object obj,
char* propertyName, char* propertyType,
Pointer newValue, Pointer oldValue)
```

Where the parameters are:

`obj`: the suit object that caused the interest callback to be invoked

`propertyName`:the name of the property that changed

`propertyType`:the type of the property that changed, as an ASCII string. Allowed values here are: "boolean", "double", "DynArray", "GP_color", "GP_font", "int", "SUIT_object", "SUIT_functionPointer", "SUIT_springiness", "SUIT_enum", "SUIT_textList", "text", "viewport", and "window" and the string name for any other type that you have registered with SUIT (see "Registering User Defined Types" on page 40).

`oldValue` a pointer to the value of the property before the property was changed. See below.

`newValue` a pointer to the value of the property after the property was changed. See Below.

NOTICE: By the time the interest callback is called, the property value has already changed, so **`newValue`** points to the value the property currently has. The interest callback is also allowed to set properties, triggering yet another (recursive) call to the interest callback.

Rarely Used

`void SUIT_registerInterestInClass (char *classname, SUIT_objectInterestCallback callback)`

Description: This function registers an interest in all widgets of a particular class. If an OBJECT level property changes for a widget of the given class, the callback function is called with that object. If a CLASS or GLOBAL level property changes, then the callback function is called for ALL objects of the given class. This is useful for imposing constraints on all objects of a given class.

Remarks: Bear in mind that when calling this function, the callback is likely to get called quite often. If the callback performs some time-consuming task, your application will slow down considerably.

Rarely Used

`void SUIT_registerInterestInGlobal (SUIT_objectInterestCallback callback)`

Description: This function registers an interest in the root object. The callback is called whenever a GLOBAL level property changes.

SUIT_object SUIT_getParent(SUIT_object obj)

Description: This function returns the parent of the given object or NULL if the object has no parent (i.e. is the root of the heirarchy tree) .

SUIT_object SUIT_getSibling(SUIT_object obj, char* relativeSiblingName)

Description: Returns the sibling of the given object given the sibling's relative name.

Example: Given an object "foobar" with children "foobar: label" and "foobar: slider", they can refer to each other with:

```
SUIT_getSibling(obj, "slider")          /* obj here is the label */
SUIT_getSibling(obj, "label")           /* obj here is the slider */
```

boolean SUIT_isAncestor (SUIT_object parent, SUIT_object child)

Description: Returns TRUE if **parent** is an anscestor (parent , grandparent , etc.) of **child**.

int SUIT_numberOfEmployees(SUIT_object, char *displayName)

Description: This function returns the number of employees a given object has for a particular display style.

int SUIT_numberOfChildren(SUIT_object obj)

Description: Returns the number of children a given SUIT_object possesses.

void SUIT_paintChildren(SUIT_object parent)

Description: This calls the paint procedures for each of the children of the parent object. This function is often used as the paint procedure for hierarchical widgets.

void SUIT_paintEmployees (SUIT_object o)

Description: This calls the paint procedures for each of the employees of the parent object, using the parent object's current display style. This function is often used as part of the paint procedure for hierarchical widgets that have employees.

char *SUIT_relativeName(SUIT_object obj, char* childName)

Description: This generates a string that is composed of the name of the given object concatenated to the childName passed in. This is useful for creating unique names of objects that are hierarchical.

void SUIT_removeChild(SUIT_object child)

Description: Removes the child object from the parent's list of children. The child becomes the child of root. Note that this does not destroy the object, it merely changes the object's parent.

void SUIT_removeEmployee(char *displayName, SUIT_object emp)

Description: Removes the employee object from its parent. The employee becomes the child of no object.

SUIT_viewport SUIT_mapToParent (SUIT_object obj,
double x1, double y1, double x2, double y2)

Description: Maps a floating point set of coordinates into the corresponding viewport of the given object.

See Also: A detailed description of this function is given in the section on heirarchy, page 153.

Hierarchy Functions

For a complete discussion of hierarchy, see the section called "Hierarchy" on page 153.

NOTE: Functions that manipulate events for hierarchical widgets are listed in the Events section, page 69.

```
void SUIT_addEmployeeToDisplay(SUIT_object obj, char *displayName,
                               SUIT_object employee)
```

Description: This function will link an employee object to a parent object in the given display style. The employees are added to the parent and each is assigned a unique index (starting with index 0) by which the employee can be retrieved with a call to **`SUIT_getEmployee()`**. Note that there is no need to call **`SUIT_removeEmployee()`** before calling this function. Adding an employee to a display automatically removes the employee from whatever display it was associated with already (if any).

See Also:

`SUIT_removeEmployee()`	page 73
`SUIT_getEmployee()`	page 72

```
void SUIT_addChildToObject(SUIT_object obj, SUIT_object child)
```

Description: This function adds a child object to a parent object. The children are added to the parent and each is assigned an index (starting with index 0) by which the child can be retrieved with a call to **`SUIT_getChild()`**. Note that there is no need to call **`SUIT_removeChild()`** before calling this function. Adding a child object to a parent object automatically removes the child from whatever parent it was associated with already (if any). The new child is added at the bottom of the z-ordering of children.

See Also:

`SUIT_removeChild()`	page 73
`SUIT_getChild()`	page 72

```
SUIT_object SUIT_getChild(SUIT_object parent, int whichChild)
```

Description: Given the parent object and the child object's index, this function will return the child object. As children are added to a parent with **`SUIT_addChildToObject()`**, new employees are assigned ascending indices that start with index 0.

Rarely Used

```
DynArray SUIT_getChildren (SUIT_object obj)
```

Description: This returns a DynArray of all the children of the given object.

```
SUIT_object SUIT_getEmployee(SUIT_object, char *displayName, int whichEmp)
```

Description: Given the parent object, the employee object's index, and the display to which that employee belongs, this function will return the employee object. As employees are added to a display with **`SUIT_addEmployeeToDisplay()`**, they are assigned indices that start with index 0.

Rarely Used

```
DynArray SUIT_getEmployees (SUIT_object obj)
```

Description: This returns a DynArray of all the employees of the given object.

Widget Creation Functions

`SUIT_object SUIT_createObject(char *name, char *className)`

Description: This routine creates a SUIT object and returns a pointer to it. This is used exclusively for creating new widget types, not for using any of the standard widgets in the SUIT library. Once you create a new SUIT widget, you **must** add a display style to it, which will register the hit and paint procs for the new widget. For more information on adding displays, see **`SUIT_addDisplayToObject()`**.

Parameters: **`name:`** The name of the newly created object.

`className:`The name of the class that the newly created object belongs to. Every object belongs to one and only one class.

Example:

```
/* skeleton code for creating a widget */

SUIT_object CreateANewWidget (char *name){
      SUIT_object retval;
      retval = SUIT_createObject (name, "newwidgetclass");

      /* Required: add display styles here */
      SUIT_addDisplayToObject (retval, "standard", HitFred, PaintFred)
      return (retval);
      }
```

See Also: *Frequently Asked Questions: How Do I Create My Own Widgets?* on page 21.

`void SUIT_addDisplayToObject(SUIT_object obj,`
`char *displayName,`
`void (*hitproc)(SUIT_object obj, SUIT_event ev),`
`void (*paintproc)(SUIT_object obj))`

Description: This routine adds an alternate display to an existing object. Call this routine after you've called SUIT_createObject. The different display styles of a widget determine how the widget will appear (determined by the paintproc) and how it will behave when it receives a mouse event (determined by the hit proc).

Parameters: **`obj:`** The SUIT_object to which the new display is to be added.

`displayName`: The character string for the name of the display.

`void (*hitproc)(SUIT_object obj, SUIT_event ev):` The hit procedure for this display style. Like any hit procedure, the parameters for this function are the SUIT_object being hit and the SUIT_event description of the hit. Hit procedures must return void.

`void (*paintproc)(SUIT_object obj):` The paint procedure for the object. Like all paint procedures, the parameter for this function is the SUIT_object being painted. Paint procedures must return void.

Example:

```
/* adding displays */

/* This is the code for two of the four display styles for the standard
bounded value widget. Notice that the display styles are named in a way
that denotes their appearance and that the hit and paint procs are
different for the two different styles. */

o = SUIT_createObject (name, "bounded value");
SUIT_addDisplayToObject(o,"speedometer",
      HitAnalogueDisplay, DrawAnalogueDisplay);
SUIT_addDisplayToObject(o, "vertical thermometer",
          HitVerticalThermometer, DrawVerticalThermometer);
```

See Also: *Frequently Asked Questions: How Do I Create My Own Widgets?* on page 21.

Rarely Used

`void SUIT_registerTrapper(SUIT_trapperPtr trapper)`

Description: This function registers a function with SUIT that gets called before any events are passed to objects that are hit; a feature that can be used to implement “hot keys” or “keyboard accelerators” for menus.

Parameters: **`trapper`**: This is a pointer to a function (that the programmer writes) that examines all user input before the SUIT main loop processes it. The parameters to this function are: the SUIT_object that the cursor was over when the event occurred (NULL if no object) and a pointer to the SUIT_event itself. The function either returns a SUIT_object or NULL. These return values are explained below.

In the case where the trapper function returns a SUIT_object, SUIT continues processing the event in the usual manner using the object returned as the object that was hit by the event. Notice that the trapper function can return any SUIT_object it pleases so, for example, a trapper function can be told that an input event happened over button A, but the trapper function can return button B as a result, causing SUIT to react as if the event actually happened on button B. Notice also that because the trapper function is given a pointer to the SUIT_event, the trapper function is allowed to alter the SUIT_event as needed before returning (for instance, to change the coordinates of the locator field).

In the case where the trapper returns NULL, this means that the event has been “consumed” or “handled” and so the SUIT main loop is considered “done”. This is the return value one uses normally for processing hot keys.

Trappers are additive. Calling **`SUIT_registerTrapper()`** several times in a row will cause the previously registered trapper function(s) to be pushed down on a stack. When an input event happens, the trapper functions are called in reverse order: the last trapper registered is called first, followed by the next to last trapper registered and so on. When all trapper functions in the stack have been called, SUIT processes the event as usual (assuming that the event wasn’t consumed by one of the trapper functions in the stack along the way). The last trapper registered can be removed (popped) from the stack by calling **`SUIT_unregisterTrapper()`**. This is useful, for example, for bringing up several dialog boxes sequentially, each of which wants to grab the keyboard event of the “enter key” being pressed, each needing a different trapper function.

See Also: The description of the type **`SUIT_trapperPtr`** described on page 38.

`SUIT_unregisterTrapper()` on page 70.

Rarely Used

`void SUIT_unregisterTrapper()`

Description: This function unregisters the last function with SUIT that was registered with **`SUIT_registerTrapper()`**.

See Also: **`SUIT_registerTrapper()`** on page 70.

Event Functions

`SUIT_event SUIT_adjustEventForObject(SUIT_event starter,`
`SUIT_object parent,`
`SUIT_object child)`

Description: This function re-computes all the appropriate fields of a SUIT_event so that it can be processed by a widget's child. The event returned is ready to be handed to the child event via **`SUIT_hitObject()`**.

`void SUIT_hitObject(SUIT_object obj, SUIT_event ev)`

Description: This function invokes the hit procedure of the given object. This hit procedure is called with the event ev.

`void SUIT_paintObject(SUIT_object obj)`

Description: This function pushes the graphics state on a stack, calls the paint procedure that was registered with the object when the object was created and then pops the graphics state off the stack. The pushing and popping of graphics state is done so that the widget can set colors and line styles without disturbing the painting routines of any other widget. For more information, see page 89.

See Also: **`SUIT_addDisplayToObject()`** on page 71.

`void SUIT_passEventToChild(SUIT_object parent, SUIT_event ev)`

Description: This function will search the hierarchy structure of the given object, looking for the child object that the given event happened over. Once the correct child is found, this function calls that child's hit procedure. It is up to the child object to handle the event from there.

NOTE: Do not call **`SUIT_adjustEventForObject()`** before calling this function. **`SUIT_passEventToChild()`** calls the adjustment function for you.

`void SUIT_passEventToEmployee(SUIT_object parent, SUIT_event ev)`

Description: This function will search the hierarchy structure of the given object, looking for the employee object that the given event happened over. Once the correct employee is found, this function calls that employee's hit procedure. It is up to the employee object to handle the event from there.

`void SUIT_reportMouseMotion(SUIT_object obj, SUIT_mouseMotion motion)`

Description: This function registers with SUIT the kind of mouse motion that is to be reported for the given object. **`WHILE_MOUSE_DOWN`** (report mouse motion while the mouse button is down over the widget), and **`UNTIL_MOUSE_UP`** (report mouse motion until the mouse button comes up, regardless of whether the mouse is over the widget or not)

See Also: **`SUIT_mouseMotion`** on page 35.

Rarely Used

`void SUIT_suspendMarkingRedisplay(SUIT_object obj)`

Description: This function instructs SUIT not to mark objects as requiring redisplay when the value of a property changes (which is SUIT's default behavior). This is useful for changing the value of a property for a parent object for book keeping purposes while keeping the number of redisplays down to a minimum.

See Also: **`SUIT_resumeMarkingRedisplay()`** on page 68.

Rarely Used

`void SUIT_resumeMarkingRedisplay(SUIT_object obj)`

Description: This function instructs SUIT to resume the marking of objects as requiring redisplay when the value of a property changes (this is SUIT's default behavior). This is useful for changing the value of a property for a parent object for book keeping purposes while keeping the number of redisplays down to a minimum.

See Also: **`SUIT_suspendMarkingRedisplay`** on page 67.

`void SUIT_backgroundAndBorderObject(SUIT_object obj)`

Description: This routine draws a rectangle in the object's BACKGROUND_COLOR the size of the object's viewport and then calls **`SUIT_borderObject()`** to draw a border around the rectangular extents of the given object. Unlike **`SUIT_redisplayRequired()`**, this function draws when called.

`void SUIT_borderObject(SUIT_object obj)`

Description: This routine draws a border around the rectangular extents of the given object using the BORDER_COLOR, BORDER_WIDTH and BORDER_STYLE properties for that object. Unlike **`SUIT_redisplayRequired()`**, this function draws when called.

Rarely Used

`void SUIT_clearScreen(void)`

Description: This routine will clear the application window by repainting over all widgets in the GLOBAL background color.

`void SUIT_eraseObject(SUIT_object obj)`

Description: This function paints over a given object in the OBJECT's background color. Note that the object is not destroyed, merely painted over.

Rarely Used

`SUIT_viewport SUIT_getScreenViewport(void)`

Description: This routine will find out the size (in pixels) of the size of the application window and return that size in the form of a SUIT_viewport. In a DOS environment, this will be the size of the whole physical screen.

Display Functions

Notice that this section contains some functions that DO NOT perform their actions (such as re-draw widgets) immediately upon being called; they merely *flag objects as requiring repainting*, so that all the painting can be done in one place in the SUIT main inner loop. Read the function descriptions carefully: those that indicate that they "mark an object for redisplay" are delay their actions; those that "perform a redisplay" do so immediately. See also the section on the main SUIT inner loop on page 151.

`void SUIT_redisplayRequired(SUIT_object obj)`

Description: This function marks a SUIT_object as needing redisplay. The redisplay is not immediate, but happens the next time objects are redisplayed in the SUIT main loop. NOTE: this function is called automatically, every time a property of an object is changed, therefore, there is little need to ever call this function.

`void SUIT_redisplayNotRequired(SUIT_object object)`

Description: This function marks a SUIT_object as NOT needing redisplay.

Rarely Used

`void SUIT_allObjectsRequireRedisplay(char *className)`

Description: This routine marks all objects in the specified class (or in the entire application) as needing redisplay.

Remarks: Like **`SUIT_redisplayRequired()`**, the redisplay here is not immediate, but happens the next time objects are redisplayed in the main SUIT loop. If passed NULL, this routine will mark all objects as needing to be redisplayed.

Parameters: **`className:`** the class of objects that you want to redisplay.

`void SUIT_performRedisplay(void)`

Description: This function performs a redisplay of all widgets in the application that require it. This is only necessary if you wish an object to be redisplayed before the usual time which is at the return from **`SUIT_checkAndProcessInput()`**.

See Also:

`SUIT_allObjectsRequireRedisplay()`	page 67
`SUIT_redisplayRequired()`	page 67

`void SUIT_redisplayRequiredInRegion(SUIT_viewport rect)`

Description: This routine will correctly restore a rectangular region of the screen.

Parameters: **`SUIT_viewport rect:`** The rectangular region of the screen that is to be redrawn, given in integer pixel coordinates.

See Also: For more information on creating viewports, see page 39.

`void SUIT_redrawObjectsAbove(SUIT_object obj)`

Description: This function redraws all objects that overlap and are “above” the given object.

Miscellaneous Property Functions

```
void SUIT_eraseProperty(SUIT_object obj, char *propertyName,
                        SUIT_level whichLevel)
```

Description: This function removes a property from a SUIT_object at the level specified.

Parameters: **obj:** The object from which the property is going to be removed

propertyName: The name of the property to remove.

level: The level from which to remove the property.

Example:

```
/* Remove the "current value" property from fred at the OBJECT level */
SUIT_eraseProperty (fred, CURRENT_VALUE, OBJECT);
```

```
boolean SUIT_propertyExists (SUIT_object obj, char *propertyName,
                        char* propertyType, SUIT_level level)
```

Description: Returns TRUE if the property of the given name AND type exists at the given level.

```
SUIT_object SUIT_getOneObjectFromClass(char *classname)
```

Description: This function returns a suit object from the given class. This is useful because setting a property at the class level with one of the deluxe set calls requires that you pass in an object of that class.

Example:

```
/* Set a property at the class level "bounded value" */
obj = SUIT_getOneObjectFromClass("bounded value");
SUIT_deluxeSetDouble(obj, CURRENT_VALUE, 4.0, CLASS, PERMANENT);
```

```
void SUIT_lockProperty(SUIT_object o, char *propertyName, SUIT_level level)
```

Description: This function prevents a user from changing the value of a property from the property editor or from exporting a property via the property editor's "export" facility. A locked property can still be changed under program control.

```
void SUIT_unlockProperty(SUIT_object o, char *propertyName, SUIT_level level)
```

Description: This function unlocks a property that has been locked with a **SUIT_lockProperty()** call. Once the property is unlocked, the user may change its value by using the property editor and may export it via the "export" button.

```
void SUIT_makePropertyPermanent(SUIT_object o, char *propertyName,
                        SUIT_level level)
```

Description: This function makes a property PERMANENT, meaning that the property will be saved at the end of program execution, to be restored the next time the program is run.

```
void SUIT_makePropertyTemporary(SUIT_object o, char *propertyName,
                        SUIT_level level)
```

Description: This function makes a property TEMPORARY, meaning that the property will NOT be saved at the end of program execution.

```
boolean SUIT_propertyIsLocked(SUIT_object o, char *propertyName,
                        SUIT_level level)
```

Description: This function returns true if the given property has been locked.

SUIT_getProperty

Rarely Used

```
Pointer SUIT_deluxeGetProperty(SUIT_object obj,
                               char *propertyName,
                               char *propertyType,
                               SUIT_level whichLevel)
```

Description: This is a very specific routine similar to the other SUIT_get calls: it checks only one level for the property, and if it doesn't already exist, it creates it as a PERMANENT property, as specified. It returns a Pointer (a built-in generic pointer type) to the value of the property in question. This is a pointer into SUIT's internal data structures: be careful not to change the value of the property.

Parameters: **`obj:`** The SUIT_object whose property you wish to get. To look for properties only at the GLOBAL level, pass in NULL.

`level:` Denotes at which of the three levels the property is to be set. Legal values here are OBJECT, CLASS, and GLOBAL.

`propertyName:` This is a string that denotes the name of the property that you wish to get.

`propertyType:` This is a string that denotes the type of the property that you wish to set. Allowed values for this string are: "boolean", "double", "DynArray", "GP_color", "GP_font", "int", "SUIT_object", "SUIT_functionPointer", "SUIT_springiness", "SUIT_enum", "SUIT_textList", "text", "viewport", and "window". To use other types requires that you first call **`SUIT_registerType().`** For details, see page 40.

Rarely Used

```
Pointer SUIT_getProperty(SUIT_object obj, char *propertyName,
                         char *propertyType)
```

Description: As with SUIT_setProperty, there is a generic "SUIT_getProperty" that is used only when you define your own types. Usually, one of the type-specific "SUIT_get" calls is used to get the value of a property.

Remarks: SUIT will look up the value of the property in a list of properties SUIT maintains for each object. For every object, there are three levels at which a property might be found; SUIT searches for the property first at the OBJECT level followed by the CLASS and GLOBAL levels, in that order. If SUIT_getProperty is passed NULL for the object, the SUIT runtime system will botch an assertion. To search for properties at the GLOBAL level, use **`SUIT_deluxeSetProperty()`** and pass in NULL as the object. If after exhausting the search at all levels, the property being requested is not found, this means that the property does not exist and therefore SUIT creates the property, placing it by default at the CLASS level and assigning it a default value dependent on its type.

Parameters: **`obj:`** This is the object whose property you want to get. If SUIT_getProperty is passed NULL for the object, it only searches the GLOBAL level for the attribute.

`propertyName:` This is a string that denotes the name of the property that you wish to get.

`propertyType:` This is a string that denotes the type of the property that you wish to set. Allowed values for this string are "boolean", "double", "DynArray", "GP_color", "GP_font", "int", "SUIT_object", "SUIT_functionPointer", "SUIT_springiness", "SUIT_enum", "SUIT_textList", "text", "viewport", and "window".

GP_font **SUIT_deluxeGetFont(SUIT_object obj, char *name, SUIT_level whichLevel)**

Pointer **SUIT_deluxeGetFunctionPointer(SUIT_object obj, char *name, SUIT_level whichLevel)**

int **SUIT_deluxeGetInteger(SUIT_object obj, char *name, SUIT_level whichLevel)**

SUIT_springiness **SUIT_deluxeGetSpringiness(SUIT_object obj, char *name, SUIT_level whichLevel)**

SUIT_object **SUIT_deluxeGetObject(SUIT_object obj, char *name, SUIT_level whichLevel)**

char * **SUIT_deluxeGetText(SUIT_object obj, char *name, SUIT_level whichLevel)**

WARNING: This function DOES NOT MAKE A COPY OF THE STRING. The char pointer returned is a pointer to the copy of the string that SUIT is maintaining. Be very careful with this string, so as not to overwrite it or change it. To make a copy of the string for your own use, you can use the C function **strdup()** (string duplicate) or **SUIT_copyString()**.

Example:

```
/* OK */
char *s;
s = SUIT_getText(myobj, LABEL);
/* SUIT_setText will make a copy when doing the set */
SUIT_setText (obj_1, LABEL, SUIT_deluxeGetText(typeInBox_1,
CURRENT_VALUE, OBJECT));

/* NOT OK */
strcat (SUIT_deluxeGetText(obj, LABEL, OBJECT),"This will trash SUIT's
string");
```

SUIT_textList **SUIT_deluxeGetTextList (SUIT_object obj, char *nameSUIT_level whichLevel)**

Description: This is a function used mostly in conjunction with the scrollable list widget. The textList returned is a COPY of the text list, not the original text list maintained by the object.

SUIT_viewport **SUIT_deluxeGetViewport(SUIT_object obj, char *name, SUIT_level whichLevel)**

SUIT_window **SUIT_deluxeGetWindow(SUIT_object obj, char *name, SUIT_level whichLevel)**

The “SUIT_deluxeGet” Functions

These functions are analogous to the SUIT_get functions, except that these allow you to specify the level at which the property is to be found. If the property is not found at the specified level, a PERMANENT property of the given name and type will be created at the given SUIT_level.

Parameters: **`obj:`** The SUIT_object whose property you wish to get.

`name:` The name of the property that you want to get.

`level:` Denotes at which of the three levels the property is to be found. Legal values here are OBJECT, CLASS, and GLOBAL. If the property is not found at that level, SUIT will create the property at that level.

IMPORTANT NOTES

SUIT_deluxeGetText returns a ***pointer*** to the string; it does ***not*** allocate memory for the string returned. For this, you will need to call **`SUIT_copyString()`**. See page 76.

`SUIT_deluxeGetEnum()` returns a SUIT_enum, which is almost never what you want. To return the current value of a property of type SUIT_enum, use **`SUIT_deluxeEnumString()`**.

```
boolean            SUIT_deluxeGetBoolean(SUIT_object obj, char *name,
                            SUIT_level whichLevel)
```

```
GP_color           SUIT_deluxeGetColor(SUIT_object obj, char *name,
                            SUIT_level whichLevel)
```

```
double             SUIT_deluxeGetDouble(SUIT_object obj, char *name,
                            SUIT_level whichLevel)
```

```
DynArray           SUIT_deluxeGetDynArray(SUIT_object obj, char *name,
                            SUIT_level whichLevel)
```

Rarely Used

```
SUIT_enum                SUIT_deluxeGetEnum(SUIT_object obj, char *name,
                                  SUIT_level whichLevel)
```

```
char *SUIT_deluxeGetEnumString (SUIT_object o, char *propName,
                            SUIT_level level)
```

Description: This function returns the string corresponding to the currently selected choice in the SUIT_enum. It is analogous to the **`SUIT_deluxeGetEnum()`** call, except that this function returns the string rather than the SUIT_enum itself. Like the other **`"SUIT_deluxeGet"`** calls, this function allows you to specify the level at which to search for the property.

Example: **`SUIT_deluxeGetEnumString (obj, BORDER_STYLE, CLASS);`**

SUIT_textList   SUIT_getTextList (SUIT_object obj, char *name)

Description: This is a function used mostly in conjunction with the scrollable list widget. The textList returned is a COPY of the text list, not the original text list maintained by the object.

SUIT_viewport   SUIT_getViewport(SUIT_object obj, char *name)

Description: This returns a property of type SUIT_viewport.

Example: `SUIT_viewport vp =SUIT_getViewport(obj, VIEWPORT);`

SUIT_window   SUIT_getWindow(SUIT_object obj, char *name)

Description: This returns a property of type SUIT_windowt.

Example: `SUIT_window win =SUIT_getViewport(obj, WINODW);`

GP_font **SUIT_getFont(SUIT_object obj, char *name)**

Description: This gets a GP_font property from a widget.

Example:
```
GP_font myFont = SUIT_getFont (obj, FONT);
```

Pointer **SUIT_getFunctionPointer(SUIT_object obj, char *name)**

Description: This function gets the value of a function pointer property. The pointer returned is really just a (void *), so be careful to pass the correct number and type of parameters if you are going to use it as a function pointer.

Example:
```
func = SUIT_getFunctionPointer (obj, CALLBACK_FUNCTION);

/* now call func */
func();
```

int **SUIT_getInteger(SUIT_object obj, char *name)**

Description: This function gets an integer property.

Example:
```
SUIT_getInteger(obj, BORDER_WIDTH);
```

Rarely Used

SUIT_springiness **SUIT_getSpringiness(SUIT_object obj, char *name)**

Description: Gets a SUIT_springiness value from a property.

Example:
```
spring = SUIT_getSpringiness (obj, SPRINGINESS);
```

See Also: The discussions of Springiness on pages 156, 38 and 111.

SUIT_object **SUIT_getObject(SUIT_object obj, char *name)**

Description: This function returns the value of property of type SUIT_object that was set with **SUIT_setObject()**.

Example:
```
SUIT_getObject (myobj, SOME_SUIT_OBJECT);
```

char * **SUIT_getText(SUIT_object obj, char *name)**

WARNING: This function DOES NOT MAKE A COPY OF THE STRING. The char pointer returned is a pointer to where the string is internally stored in SUIT's data structures. Be very careful with this string, so as not to overwrite it or change it. To make a copy of the string for your own use, you can use the C function **strdup()** (string duplicate) or the equivalent SUIT utility function **SUIT_copyString()**.

Example:
```
/* OK */

char *s;

s = SUIT_getText(myobj, LABEL);
/* SUIT_setText will make a copy when doing the set */
SUIT_setText (obj_1, LABEL, SUIT_getText(typeInBox_1, CURRENT_VALUE));

/* NOT OK */
strcat (SUIT_getText(obj, LABEL), "This will trash SUIT's string");
```

Getting Values of Properties

These macros are used to get the values of SUIT properties.

Parameters: **obj:** The SUIT_object whose property you wish to get.

name: The name of the property that you want to get.

IMPORTANT NOTES

`SUIT_getText()` returns a ***pointer*** to the string; it does ***not*** allocate memory for the string returned. For this, you will need to call **`SUIT_copyString()`**. See page 76.

`SUIT_getEnum()` returns a SUIT_enum, which is almost never what you want. To get the value of a SUIT_enum, use **`SUIT_getEnumString()`**.

`boolean SUIT_getBoolean(SUIT_object obj, char *name)`

Description: This function returns either TRUE or FALSE, values that can be used in conditionals.

Example:
```
if (SUIT_getBoolean (obj, HAS_BORDER)) { /* .... */}
```

`GP_color SUIT_getColor(SUIT_object obj, char *name)`

Description: This call returns a color from a GP_color property.

Example:
```
GP_color myColor = SUIT_getColor (obj, FORGROUND_COLOR);
```

`double SUIT_getDouble(SUIT_object obj, char *name)`

Description: This call returns a double.

Example:
```
double num = SUIT_getDouble (obj, CURRENT_VALUE);
```

`DynArray SUIT_getDynArray(SUIT_object obj, char *name)`

Description: This call returns a DynArray. For more information on DynArrays, see page 42.

Example:
```
DynArray arr = SUIT_getDynArray (obj, "my DynArray property");
```

Rarely Used

`SUIT_enum SUIT_getEnum(SUIT_object obj, char *name)`

Description: This returns a SUIT_enum from a property of type SUIT_enum. If you want to get the **value** of the SUIT_enum rather than the SUIT_enum itself, use **`SUIT_getEnumString()`**.

Example:
```
/* getting a SUIT_enum -- not usually what you want */
SUIT_enum myEnum = SUIT_getEnum (obj, ACTIVE_DISPLAY);
```

`char *SUIT_getEnumString (SUIT_object o, char *propName)`

Description: This function returns the string corresponding to the currently selected choice in the SUIT_enum. It is analogous to the **`SUIT_getEnum()`** call, except that this function returns the string rather than the SUIT_enum itself.

Example:
```
/* getting the value of a SUIT_enum and printing it out */
printf ("current display style is %s\n",
                SUIT_getEnumString(obj, ACTIVE_DISPLAY));
```

SUIT_setProperty

Rarely Used

```
void SUIT_setProperty(SUIT_object obj,
                      char *propertyName,
                      char *propertyType,
                      Pointer propertyPtr,
                      SUIT_level whichLevel,
                      int permanence)
```

Description: IMPORTANT: This function is used to set the value of a property of a SUIT object if the property is of a user defined type. If you are setting the value of a property of a built-in SUIT type (which is the usual case) you should use one of the type-specific calls available instead. see "Setting Values of Properties" on page 54. In practice, calling the function “SUIT_setProperty” is necessary only if you are intending to register your own data types with SUIT.

Parameters: **obj:** This is the object whose property you want to set.

propertyName: This string is the name of the property that you wish to set.

propertyType: This denotes the type of the property that you wish to set. Allowed values for this string are "boolean", "double", "DynArray", "GP_color", "GP_font", "int", "SUIT_object", "SUIT_functionPointer", "SUIT_springiness", "SUIT_enum", "SUIT_textList", "text", "viewport", and "window". To use other types requires that you first call **SUIT_registerType().** For details, see page 40.

propertyPtr: This is a generic pointer (of type Pointer) to the information being copied into the property.

whichLevel: The level at which the property is to be set. Allowed values are OBJECT, CLASS, and GLOBAL.

Examples:

```
/* setting properties */

/* This will set an integer property called CURRENT_COUNT at the OBJECT
level on a SUIT_object called MyCounter. Notice that a pointer to the
integer is passed in, not the integer itself.
 */

/* the use of #define’s helps avoid the possibility of typos... */
#define CURRENT_COUNT "CURRENT COUNT"

SUIT_object MyCounter;
int cur_count = 9;
SUIT_setProperty (MyCounter, CURRENT_COUNT, "int", &cur_count,
                  OBJECT, PERMANENT);
```

```
void SUIT_deluxeSetEnumString (SUIT_object o, char *propName,
                               char *enumString, SUIT_level level)
```

Description: This function sets the SUIT enum property given by string name rather than by giving the SUIT_enum. If the string is not a valid choice in the enumeration (e.g. attempting to set the choice to be "dog" in the enumeration "red, green, yellow") the set does not occur. Like the other **"SUIT_deluxeSet"** calls, this function allows you to specify the level at which to set the property.

Example: `SUIT_deluxeSetEnumString (obj, ACTIVE_DISPLAY, "standard", CLASS);`

```
void SUIT_deluxeSetFont(SUIT_object obj, char *name, GP_font val,
                        SUIT_level whichLevel)
```

```
void SUIT_deluxeSetFunctionPointer(SUIT_object obj, char *name, Pointer val,
                                   SUIT_level whichLevel)
```

```
void SUIT_deluxeSetInteger(SUIT_object obj, char *name, int val,
                           SUIT_level whichLevel)
```

```
void SUIT_deluxeSetSpringiness(SUIT_object obj, char *name,
                               SUIT_springiness val, SUIT_level whichLevel)
```

```
void SUIT_deluxeSetObject(SUIT_object obj, char *name, SUIT_object val,
                          SUIT_level whichLevel)
```

```
void SUIT_deluxeSetText(SUIT_object obj, char *name, char *val,
                        SUIT_level whichLevel)
```

```
void SUIT_deluxeSetTextList(SUIT_object obj, char *name,
                            SUIT_textList val, SUIT_level whichLevel)
```

```
void SUIT_deluxeSetViewport(SUIT_object object o, char *name,
                            SUIT_viewport val, SUIT_level whichLevel)
```

```
void SUIT_deluxeSetWindow(SUIT_object obj, char *name, SUIT_window val,
                          SUIT_level whichLevel)
```

The "SUIT_deluxeSet" Functions

These functions are analogous to the SUIT_set functions from the previous pages, but these "deluxe" functions specify at which SUIT level the property is to be set.

Description: These functions are used to set the values of SUIT properties.

Parameters: **`obj:`** The SUIT_object whose property you wish to set.

`name:` The name of the property that you want to set.

`val:` This is the new value that you want the property to have. The type of **`val`**, of course, depends on which type of property you are attempting to set.

`whichLevel:` Denotes at which of the three levels the property is to be set. Legal values here are OBJECT, CLASS, and GLOBAL.

IMPORTANT NOTES

If the property being set does not exist, SUIT will create a new property of that type at the specified level and, assign it the value passed to the SUIT_deluxeSet call.

`SUIT_deluxeSetEnum()` requires a SUIT_enum, which is almost never what you want. To set the value of a SUIT_enum, use **`SUIT_deluxeSetEnumString()`**.

```
void SUIT_deluxeSetBoolean(SUIT_object obj, char *name, BOOLEAN val,
                            SUIT_level whichLevel)
```

```
void SUIT_deluxeSetColor(SUIT_object obj, char *name, GP_color val,
                            SUIT_level whichLevel)
```

```
void SUIT_deluxeSetDouble(SUIT_object obj, char *name, double val,
                            SUIT_level whichLevel)
```

Rarely Used

```
void SUIT_deluxeSetDynArray(SUIT_object obj, char *name, DynArray val,
                                        SUIT_level whichLevel)
```

Rarely Used

```
void SUIT_deluxeSetEnum(SUIT_object obj, char *name, SUIT_enum val,
                                        SUIT_level whichLevel)
```

void SUIT_setObject(SUIT_object obj, char *name, SUIT_object val)

Remarks: This is a way of attaching one SUIT widget to another using the property mechanism. Usually not used because if SUIT widgets need to have "close affiliation" like this, one is usually made the child of the other.

Example:
```
anotherObject = SUIT_createLabel("a label");
SUIT_setObject (myobj, SOME_SUIT_OBJECT, anotherObject);
```

void SUIT_setText(SUIT_object obj, char *name, char *val)

Remarks: This function will make a copy of whatever text string you hand it. To prevent memory leaks, you should free whatever memory you can after using this function.

Example:
```
SUIT_setText(myObj, LABEL, "This is the label");
SUIT_setText(myObj, LABEL, sprintf(buf, "thing %d", num));
```

void SUIT_setTextList(SUIT_object obj, char *name, SUIT_textList list)

Remarks: This function is used mostly for the scrollable list widget.

Example:
```
char *myListOfNames[] = {"James", "Scotty", "Spock"};
SUIT_setTextList (myobj, LIST, SUIT_defTextList (myListOfNames, 3));
```

See Also: **SUIT_defTextList()** page 44

void SUIT_setViewport(SUIT_object obj, char *name, SUIT_viewport val)

Description: This function will set a property of type SUIT_viewport to some value. Below are some examples of how SUIT_viewports might appear as parameters to this function.

Example:
```
SUIT_setViewport (myobj, VIEWPORT, some_viewport_variable);
SUIT_setViewport (myobj, VIEWPORT, SUIT_defViewport(10, 10, 40, 50));
SUIT_setViewport (myobj, VIEWPORT,
                        SUIT_getViewport(anotherObj, VIEWPORT);
```

void SUIT_setWindow(SUIT_object obj, char *name, SUIT_window val)

Description: This function will set a property of type SUIT_window to some value. Below are some examples of how SUIT_windows might appear as parameters to this function.

Example:
```
SUIT_setWindow (myobj, WINDOW, some_window_variable);
SUIT_setWindow (myobj, WINDOW, SUIT_defWindow(0.5, 0.5, 2.3, 6.2));
SUIT_setWindow (myobj, WINDOW,
                        SUIT_getWindow(anotherObj, WINDOW);
```

void SUIT_setDynArray(SUIT_object obj, char *name, DynArray val)

Example: `SUIT_setDynArray(obj, CURRENT_VALUE, myArr);`

See Also: Discussion of DynArrays on page 42.

Rarely Used

void SUIT_setEnum (SUIT_object obj, char *name, SUIT_enum val)

WARNING: This is a highly specialized call. There are SUIT widgets that can use SUIT_enums directly, there are others that cannot. Usually, you will want to use the **`SUIT_setEnumString()`** call instead of this one.

See Also: For details concerning the definition of SUIT_enums, see page 41.

Example:

```
char* shapeList[4] = {"circle", "square", "line", "polygon"};

SUIT_setEnum(obj, CURRENT_VALUE, SUIT_defEnum("line", 4, shapeList));
```

void SUIT_setEnumString (SUIT_object o, char *propName, char *enumString)

Description: This function sets the SUIT enum property given by string name rather than by giving the SUIT_enum. If the string is not a valid choice in the enumeration (e.g. attempting to set the choice to be “dog” in the enumeration “red, green, yellow”) the set does not occur.

Example:

```
SUIT_setEnumString (obj, ACTIVE_DISPLAY, "standard");

SUIT_setEnumString (obj, EXAMPLE_COLORS, "red");
SUIT_setEnumString (obj, BORDER_STYLE, 'motif");
```

void SUIT_setFont(SUIT_object obj, char *name, GP_font val)

Example:

```
SUIT_setFont(myobj, FONT, GP_defFont("times", "bold", 12));

SUIT_setFont(myobj, FONT, SUIT_getFont(anotherObj, FONT));
```

void SUIT_setFunctionPointer(SUIT_object obj, char *name, Pointer val)

Description: Allows you to attach a function pointer to a SUIT_object.

Example:

```
/* myFcn is a pointer to a function that returns
void and takes no parameters */
SUIT_setFunctionPointer (myobj, CALLBACK_FUNCTION, myFcn);
```

void SUIT_setInteger(SUIT_object obj, char *name, int val)

Example: `SUIT_setInteger (myobj, BORDER_WIDTH, 4);`

void SUIT_setSpringiness(SUIT_object obj, char *name, SUIT_springiness val)

Example: `SUIT_setSpringiness (myobj, SPRINGINESS, BELOW_SPRINGINESS);`

See Also: The discussions of Springiness on pages 156, 38 and 111.

Setting Values of Properties

The following notes refer to all of the “SUIT_set” function calls:

Description: These functions are used to set the values of SUIT properties for all of the built-in SUIT types. If you wish to create a type of your own choosing, see "Registering User Defined Types" on page 40.

Parameters: **`obj:`** The SUIT_object whose property you wish to set.

`name:` The name of the property that you want to set. Property names are always strings of characters. The properties that SUIT defines have been **`#defined`** as constants with all uppercase and underscores between words. (The listing and explanation of all the built-in SUIT property names begins on page 110.) If you are going to create property names of your own, we strongly recommend that you also use **`#define`** to turn your strings into constants as well. This gain the advantage of compiler time checking of the names of your properties. If you spell a string incorrectly, SUIT will create a new property of that name; not at all what you want. #defines help prevent simple typos like this because the typo is caught at compile time.

`val:` This is the new value that you want the property to have. Note that this is the new value, not a pointer to that value, which is different than the **`SUIT_setProperty()`** call, which asks for a pointer to the new value. The type of **`val`**, of course, depends on which type of property you are attempting to set.

IMPORTANT NOTES:

If the property being set does not exist, SUIT will create a new property of that type at the OBJECT level and, assign it the value passed to the SUIT_set call.

All built-in SUIT properties should be expressed as **`#define`** constants: all uppercase with underscores between words. (e.g. FOREGROUND_COLOR)

All properties are set at the OBJECT level and tagged as being PERMANENT except for SUIT_objects and functionPointers, which are tagged as TEMPORARY (not written to the .sui file).

`SUIT_setEnum()` requires a SUIT_enum, which is almost never what you want.To set the value of a SUIT_enum, use **`SUIT_setEnumString()`**.

`void SUIT_setBoolean(SUIT_object obj, char *name, boolean val)`

Example:

```
SUIT_setBoolean(myobj, HAS_BORDER, FALSE);
SUIT_setBoolean(myobj, VISIBLE, TRUE);
```

`void SUIT_setColor(SUIT_object obj, char *name, GP_color val)`

Remarks: Below are three ways to produce a color:

Example:

```
SUIT_setColor(myobj, FOREGROUND_COLOR, GP_defColor("red", FALSE));
SUIT_setColor(myobj,FOREGROUND_COLOR,
                    GP_defColorRGB(320, 120, 320, TRUE));
SUIT_setColor(myobj,FOREGROUND_COLOR,
                    SUIT_getColor(anotherObj, FOREGROUND_COLOR));
```

See Also: For functions that define GP_colors, see page 33.

`void SUIT_setDouble(SUIT_object obj, char *name, double val)`

Example:

```
SUIT_setDouble(myobj, CURRENT_VALUE, 3.14159);
```

`rectangle SUIT_moveRectangle(rectangle oldLocation, point pt, boolean allowOffScreen)`

Description: This function is useful for moving a rectangle under program control. Upon calling this function, SUIT allows the user to press the mouse button down, at which point SUIT will manage the dragging of a dashed line rectangle that follows the mouse cursor. When the user releases the mouse button, the function exits and passes back as a return value, the location of the new rectangle.

NOTE: Notice that this can be used in conjunction with widget viewports, to allow you to interactively "drag" widgets around. Widgets that are not heirarchical (i.e. those that are children of the root object) can use their viewports as the **`oldLocation`** parameter. With widgets that are heirarchical, the viewport needs to be converted into absolute screen coordinates before being passed to **`SUIT_moveRectangle()`**. The general strategy is to call **`SUIT_mapViewportToScreen()`**) before the call to **`SUIT_moveRectnagle()`** and **`SUIT_mapScreenToViewport()`** afterwards.

Remarks: **`pt:`** This point represents the point from which the rectangle will be dragged around the screen. This quantity can be found in one of the fields of a SUIT_event (the field is called "locator.position").

`allowOffScreen`: This dictates whether or not the rectangle can be dragged off the screen.

Example:

```
/* fred's hit procedure: this moves the widget when the user holds the
   mouse button down. fred IS A NON-HEIRARCHICAL WIDGET. */

void HitFred (SUIT_object fred, SUIT_event e){
      point pos;
      SUIT_viewport oldvp, newvp;

      pos = e.locator.position;
      /* this will let the user move fred without using SUIT-keys */
      oldvp = OBJECT_VIEWPORT(fred);
      newvp = SUIT_moveRectangle (oldvp, pos, TRUE);
      SUIT_setViewport (fred, VIEWPORT, newvp);
}
```

`rectangle SUIT_resizeRectangle(rectangle originalvp)`

Description: This function is useful for resizing a viewport under program control. Upon calling this function, SUIT will wait for the user to select one of the resize handles at which point SUIT will manage the dragging of a rubberband dashed line rectangle that follows the mouse cursor. When the user releases the mouse button, the function exits and passes back as a return value the location of the newly resized SUIT_viewport.

Remarks: **`originalvp:`** This is the starting viewport.

Example:

```
/* fred's hit procedure: this resizes the widget when holds the mouse
button down. */

void HitFred (SUIT_object fred, SUIT_event e){
      SUIT_viewport oldvp, newvp;

      oldvp = OBJECT_VIEWPORT(fred);
      newvp = SUIT_resizeRectangle (oldvp);
      SUIT_setViewport (fred, VIEWPORT, newvp);
}
```

SUIT_object SUIT_mapPointToObject(point p)

Description: This function will take a point in screen coordinates (typically the coordinates of an event) and return the SUIT_object that the point falls in. The object returned is the most "deeply nested" object in the hierarchy that contains the point. If the point is over no objects, the function returns NULL. In the current version of SUIT, this function does not find employees, only children.

Example:

```
/* find the object struck with SUIT_event called "ev" */
object_struck = SUIT_mapPointerToObject (ev.relativePixelLocation);
```

SUIT_viewport SUIT_mapScreenToViewport (SUIT_object parent, rectangle scr);

Description: This function will convert a rectangle (specified in absolute screen coordinates) into the corresponding SUIT_viewport. Note that you must pass in a parent object -- this is the parent object that contains the returned SUIT_viewport. For more information on mapping screen coordinates to viewports, see page 20.

point SUIT_mapRelativeLocationToScreen (SUIT_object o, point p)

Description: This converts a point relative to an object's viewport to a point in absolute screen coordinates.

point SUIT_mapScreenToRelativeLocation (SUIT_object o, point p)

Description: This converts a point in absolute screen coordinates to point relative to the object's viewport.

rectangle SUIT_mapViewportToScreen (SUIT_object parent, SUIT_viewport vp)

Description: This function will take a SUIT_viewport and return the corresponding rectangle in absolute screen coordinates. For more information on mapping viewports to screen coordinates, see page 20.

boolean SUIT_viewportsEqual(SUIT_viewport vp1, SUIT_viewport vp2)

Description: This function returns TRUE if the viewports are the same, FALSE if they are not.

boolean SUIT_viewportsOverlap(SUIT_viewport vp1, SUIT_viewport vp2)

Description: This function returns TRUE if the viewports overlap, FALSE if they do not.

boolean SUIT_isAnyoneOverMe(SUIT_object me)

Description: This function returns TRUE if another SUIT object overlaps and is above the given object.

boolean SUIT_pointInObject(SUIT_object obj, int x, int y)

Description: This object returns TRUE if the object is visible and the x and y screen coordinates supplied fall inside the viewport of the given object, otherwise the function returns FALSE.

void SUIT_sendToBack(SUIT_object obj)

Description: Moves a SUIT object to the bottom of the stack in the Z ordering of objects on the screen. A SUIT_object moved to the back will appear behind every other widget on the screen.

Parameters: **SUIT_object obj:** the object to move to the back.

See Also: **SUIT_bringToFront()** on page 51.

Geometric Functions

void SUIT_bringToFront(SUIT_object obj)

Description: Moves a SUIT object to the top of the stack in the Z-ordering of objects on the screen. A SUIT_object moved to the front will appear in front of every other widget on the screen.

Parameters: **SUIT_object obj:** the object to move to the front.

See Also: **SUIT_sendToBack()** page 52.

void SUIT_centerInParent (SUIT_object obj, double centerx, double centery)

Description: This function places the given object such that its center lies at the given world coordinates inside its parent. If the object has no explicit parent, (i.e. it is the child of the ROOT object), the object is centered in the root window.

Example:

```
/* Using SUIT_centerInParent() */

/* place an object so that its center lies in the exact center of its
parent */
SUIT_centerInParent (obj, 0.5, 0.5);

/* place an object inside its parent so that the object's center is
centered horizontally and 1/4 of the way up from the bottom */
SUIT_centerInParent (obj, 0.5, 0.25);
```

void SUIT_centerObjectOnScreen(SUIT_object obj)

Description: This function will place the given object in the center of the application window.

void SUIT_changeHeightPreservingRatio(SUIT_object obj, int height)

Description: This function will change the height of a SUIT object to the desired height, while also changing the width to keep the proportions of the object intact.

void SUIT_changeObjectSize(SUIT_object obj, int width, int height)

Description: This function will change the width and height of a SUIT object by enlarging or reducing the object about the object's center (The center remains in the same place on the screen).

void SUIT_changeWidthPreservingRatio(SUIT_object obj, int width)

Description: This function will change the width of a SUIT object to the desired width, while also changing the height to keep the proportions of the object intact.

void SUIT_getObjectSize(SUIT_object obj, int *width, int *height)

Description: Note that this function takes as parameters, pointers to integers.

Example:

```
int w, h;

SUIT_getObjectSIze(obj, &w, &h);
if (w > h) ....
```

Exiting and Cleanup Functions

`void SUIT_done(SUIT_saveStatus saveStat, SUIT_exitStatus exitStat)`

Description: This routine terminates the GP graphics with the option of saving the sui file, closes down SUIT, then exits the application by calling **`exit(0)`**.

Parameters: **`saveStat:`** takes one of two values:

`DO_NOT_SAVE_SUI_FILE`	".sui" file not written. changes made to the interface are lost.
`SAVE_SUI_FILE`	New ".sui" file written to disk. Old file saved as backup.

`exitStat`: takes one of two values:

`DO_NOT_EXIT_APPLICATION`	SRGP and SUIT shut down. Application still runs
`EXIT_APPLICATION`	Calls **`exit(0).`**

Rarely Used

`void SUIT_writeSUIFile(char *filename)`

Description: To write the PERMANENT properties to a .sui file. Usually, this is only done by pressing the Done Button, but this can be used in any callback to create "checkpoint" .sui files.

Parameters: **`char *filename`**: The name of the file that is to be written to disk. Usually, this is the name of the application, with an ".sui" extension.

Rarely Used

`void SUIT_deluxeInit(int *argc, char *argv[])`

Description: This initialization procedure sets up lists, pointers, and variables for SUIT and starts up the graphics package. **`SUIT_deluxeInit()`** accepts argc and argv as parameters, extracts the command line arguments intended for the SUIT package, and returns the others. The application should call this first, and then parse its own command line parameters.

Parameters: **`argc`** and **`argv`**: These two parameters are, of course, the standard C parameters passed to **`main()`** on program invocation. Notice that in the following example, a pointer to argc is passed in, not argc.

Example:

```
/* sample SUIT application */

main (int argc, char *argv[]) {
      SUIT_deluxeInit (&argc, argv);
      /* Application processes its own command line parameters here */
      SUIT_beginApplication();
}
```

`void SUIT_init(char *programName)`

Description: This initialization procedure is a simplified version of **`SUIT_deluxeInit()`** to be used when you don't need to pass command line parameters to SUIT (which is most of the time). **`SUIT_init()`** instead accepts the program name as its only parameter. This parameter is used to construct the name of the .sui file and in a windowing environment like X windows, the parameter is also used as the name of the window. Like **`SUIT_deluxeInit()`**, this function also sets up lists, pointers, and variables for SUIT and starts up the graphics package.

Parameters: **`programName:`** This is a string for the program name, usually just **`argv[0]`** from **`main()`**.

Example:

```
/* sample SUIT application */

main (int argc, char *argv[]) {
      SUIT_init (argv[0]);
      SUIT_beginApplication();
}
```

Rarely Used

`void SUIT_initFromCode(char *programName)`

Description: This initialization procedure is exactly the same as **`SUIT_init()`**, except that it does not read the sui file, but instead, initializes all widgets from compiled code. It is intended that this funciton only be called when the application you are writing is ready to ship and you have gone thorugh the shipping process described on page 161.

Initialization and Setup Functions

`void SUIT_beginStandardApplication(void)`

Description: This is nothing more than a simple wrapper for the following code:

```
SUIT_beginDisplay();
while (TRUE) {
      SUIT_checkAndProcessInput(INDEFINITE);
}
```

`void SUIT_beginDisplay(void)`

Description: This routine should be called after all objects are created. It reads information from a hints file, (if one does not exist, one is created) and does the initial display of all created objects. This routine is used in conjunction with **`SUIT_checkAndProcessInput()`** to create the main loop.

Example:

```
/* dummy SUIT application */

main (int argc, char *argv[]) {
      SUIT_init (argv[0]);
      /* create some widgets here */
      SUIT_beginDisplay();
      while (TRUE) {
            SUIT_checkAndProcessInput(INDEFINITE);
      }
}
```

`void SUIT_checkAndProcessInput(int time)`

Description: This function is at the heart of the SUIT main loop; it checks for mouse and keyboard input and sends these events to the appropriate widgets by calling the registered hit procedure for the widget that was hit. If there are no events pending in the input queue, this function exits.

Parameters: **`time:`** Maximum amount of time SUIT will wait for an input event, measured in ticks (1/60 of a second). Legal values are INDEFINITE (wait for mouse or keyboard events), or any non-negative integer which represents the maximum amount of time to wait. Setting time to 0 will cause the function to poll the queue for an input event, and exit immediately if one is not present.

Warning: Using any other time other than INDEFINITE is extremely costly in terms of CPU cycles.

See Also: **`SUIT_beginDisplay()`** page 48.

Rarely Used

`void SUIT_limitedCheckAndProcessInput (int time, Dynarray activeObjects)`

Description: This function is exactly the same as **`SUIT_checkAndProcessInput`** except that SUIT is told that there is a limited number of objects that need to be scanned for input. The objects that are to be checked are held in a DynArray of SUIT_objects.

See Also: **`SUIT_checkAndProcessInput()`** page 48.

SUIT Function Calls

int SUIT_sizeOfTextList (SUIT_textList list)

Description: This returns the number of elements in the Text List.

Example:

```
int n;

char *animals[] = {"horse", pig", "wombat"};
SUIT_textList list = SUIT_defTextList (animals , 3);
n = SUIT_sizeOfTeztList(list);
/* n is now 3 */
```

void SUIT_sortTextList (SUIT_textList list)

Description: This sorts the Text List in ascending order using a case sensitive sorting routine.

SUIT_textLists

SUIT maintains a type called Text Lists for handling lists of strings. These functions are used mostly by the scrollable list widget. For examples of Text Lists in use, see the example program for the scrollable list.

`void SUIT_appendToTextList (SUIT_textList list, char *aString)`

Description: This function will append a string to an existing Text List.

`void SUIT_addToTextList (SUIT_textList list, int beforeIndex, char *aString)`

Description: This function will add a textual item to a list before the item listed. Text lists are zero indexed, meaning that the first item is number 0.

`SUIT_textList SUIT_copyTextList (SUIT_textList list)`

Description: This function will make a copy of the given text list.

`SUIT_textList SUIT_defTextList (char *list[], int numitems)`

Description: This function will create a Text List. The parameters are the number of items and the strings that make up the list.

Example:

```
char *animals[] = {"horse", pig", "wombat"};
SUIT_textList list = SUIT_defTextList (animals , 3);
```

`void SUIT_deleteFromTextList (SUIT_textList list, int index)`

Description: This removes a string from a text list, given the string's index into the list. Text lists are zero based, meaning that the first item has index 0.

`void SUIT_destroyTextList(SUIT_textList list)`

Description: This destroys a SUIT_textList.

`char *SUIT_itemInTextList (SUIT_textList list, int index)`

Description: This retrieves an item from the Text List, given the index. **Like all occasions in SUIT where a char * is passed back, you will need to remember to copy the string if you intend to modify it.** SUIT is actually passing back a pointer to the internal string that SUIT is maintaining for this Text List.

`int DynLow(DynArray obj)`

Description: Returns the index of the lowest element in the array **`obj`**.

`int DynDelete(DynArray obj, int index)`

Description: Effects: The element index is deleted from the array obj. Note that the element is actually removed permanently from the array. If you have the array “1 2 3 4 5” and delete the third element, you will have the array “1 2 4 5”. The order of elements in not affected.

Returns: Returns DYN_OK on success or DYN_BADINDEX if the element index does not exist in the array or is less than zero.

`int DynSize(DynArray obj)`

Description: Returns the number of elements in the array.

`int DynFindIndex(DynArray obj, void *key, int (*Compare)() )`

Description: **`Compare()`** is a function that takes two (void*) parameters: pointers to two elements in the DynArray. The function must typecast the input obj to the proper user-defined structure and return the character string name within the user-structure which is to be used in the comparison.

Returns: DynFind returns the index of the element in the array (DynArrays are zero based) or DYN_-NOT_FOUND.

`int DynQsort(DynArray obj, int (*compar)(), int first_el, int last_el)`

Description: Runs quicksort on the specified subset of obj. The parameter **`compar`** is the name of the comparison function, which is called with two arguments that point to the elements being compared. As the function must return an integer less than, equal to, or greater than zero, so must the first argument to be considered be less than, equal to, or greater than the second.

Returns: Returns either DYN_OK or DYN_BADINDEX.

`int DynIsort(DynArray obj, int (*Compare)(), int first_el, int last_el)`

Description: Runs insertion sort on the specified subset of obj. Normally, DynQsort should be used, because it uses a O(n log n) method rather than a O(n squared) method, but if the caller suspects the data is mostly in order, this routine will be faster. The parameter **`compar`** is the name of the comparison function, which is called with two arguments that point to the elements being compared. As the function must return an integer less than, equal to, or greater than zero, so must the first argument to be considered be less than, equal to, or greater than the second.

Returns: Returns either DYN_OK or DYN_BADINDEX.

DynArrays

DynArrays are dynamic arrays; SUIT uses this data type internally for keeping lists of items, but you do not need to be conversant in DynArrays in order to use SUIT. DynArrays are used rarely in the programmer interface to SUIT functions; usually only the most specialized functions dealing with input handling and hierarchy use them. Note that because DynArrays are pointers to arbitrary data structures, there is no way for SUIT to know how to print such data to a file, and thus ***properties that are in DynArrays are never saved to the ".sui" file***. If you need to save DynArray properties to the ".sui" file, you should register a new type with SUIT using the **`SUIT_registerType()`** function, detailed on page 40. Examples of the use of DynArrays are in the example files that come with the SUIT distribution.

`DynArray DynCreate(int size, int increment)`

Description: **`size`** and **`increment`** are greater than zero. This creates a new DynArray that will store elements of size **`size`** and will allocate memory in blocks large enough to hold exactly **`increment`** elements. For example, if you are storing 8-byte double precision numbers and **`increment`** is 5, each 5th element you add to the array will cause it to request 40 more bytes (8 * 5) from the operating system. If **`increment`** is zero, a default value is used (currently 100). This is the only time the programmer deals with a dynamic array's memory allocation.

Returns: Returns the new DynArray, or NULL if there is insufficient memory.

`int DynDestroy(DynArray obj)`

Description: Frees all memory associated with **`obj`**. The results of calling any Dyn function on a destroyed array are undefined (except for **`DynCreate()`**, which resets the array).

Returns: DYN_OK.

`int DynAdd(DynArray obj, char *el)`

Description: Adds the element pointed to by el to the array obj, resizing the array if necessary. The new element becomes the last element in obj's array.

Returns: Returns DYN_OK on success or DYN_NOMEM if there is insufficient memory.

`void* DynGet(DynArray obj, int index)`

Description: Returns the address of the element index in the array of obj. DynArray indices are zero based. (i.e. the first element has index zero). This pointer can be treated as a normal array of the type specified to DynCreate. The order of elements in this array is the order in which they were added to the array. The returned pointer is guaranteed to be valid only until obj is modified.

Returns: Returns NULL if index is larger than the number of elements in the array of less than zero.

`int DynHigh(DynArray obj)`

Description: Returns the index of the highest element in the array **`obj`**.

SUIT Enumerated Types

Enumerated types in SUIT are supported through a SUIT type called a SUIT_enum. This type is represented as a zero indexed list of strings which denote the different choices in the enumeration and a number which represents which item in that list is the current value.

`SUIT_enum SUIT_defEnum (char *currentChoice, int num choices, char *choices[])`

Description: This defines a variable of type SUIT_enum by using an array of character pointers.

Example:

```
/* This code defines an enumerated type called "MyShapes", initializ-
ing the current choice to be "circle" */

#define NUM_SHAPES    4
char *shapeList[NUM_SHAPES]={"circle", "square", "line", "polygon"};
SUIT_enum    MyShapes;

MyShapes = SUIT_defEnum("circle", NUM_SHAPES, shapeList);
```

`char *SUIT_getEnumSelection (SUIT_enum e)`

Description: Given a SUIT_enum, this function will return the string name of the current choice.

`void SUIT_setEnumSelection (SUIT_enum *e, char *member)`

Description: Given a SUIT_enum and the name of one of its members, this function will set the SUIT_enum's current selection number to be that member, if it exists. If the choice is not a member of the SUIT_enum, the SUIT_enum is left unchanged and a non-fatal run-time error occurs.

Example:

```
/* This code defines an enumerated type called "shapes", initializing
the current choice to be "circle" */

#define NUM_SHAPES    4
char shapeList[NUM_SHAPES] = {"circle", "square", "line", "polygon");
SUIT_enum    MyShapes;

MyShapes = SUIT_defEnum(shapeList, NUM_SHAPES, "circle");

SUIT_setEnumSelection(MyShapes,"polygon"); /*choice is now polygon
*/
```

Registering User Defined Types

```
void SUIT_registerType(char *name,
                        Pointer (*readproc)(char *buffer, boolean *error),
                        char *(*writeproc)(Pointer val),
                        int (*compareproc)(Pointer ptr1, Pointer ptr2),
                        void (*destroyproc)(Pointer val),
                        Pointer (*copyproc)(Pointer val),
                        Pointer (*defaultproc) (void),
                        char* widgetClass)
```

Description: This routine registers a type with SUIT. Registering types is done so that users can set and get properties that are of user-defined types (imagine wanting to set a property whose type was a user defined struct). This function must be passed: (1) a procedure to convert an ASCII string into a data object of the new type, (2) a procedure that writes a value of the type as an ASCII string, (3) a procedure to compare values of that type, (4) a procedure to destroy a data object of that type, and (5) a pointer to a default value for the type. The read and write procedures are used to display the type in the property editor and to read and write the SUI files.

Parameters: **`name:`** This is an ASCII representation of the name of the type. (e.g. "complex number"). SUIT registers several types on start up: "boolean", "double", "DynArray", "GP_color", "GP_font", "int", "SUIT_object", "SUIT_functionPointer", "SUIT_springiness", "SUIT_enum", "SUIT_textList", "text", "viewport", and "window".

`Pointer (*readproc)(char *buffer, boolean *error):` The read procedure must accept an ASCII string (**`buffer`**) and return a Pointer to the object of the specified type. *The read procedure should return a safe malloced object. This function should NOT return a Pointer to a local variable!* The read procedure should also return a boolean flag of TRUE (in the parameter **`error`**) if an error occurs converting from ASCII to the intended type.

`char *(*writeproc)(Pointer val):` The write procedure should accept a void pointer to an object of the specified type and convert it to a an ASCII text string. The ASCII string does not have to be SUIT_malloced. WARNING: This function should NOT return a Pointer to a local character string, but may return a static buffer

`int (*compareproc)(Pointer ptr1, Pointer ptr2):` The compare procedure should take as arguments two pointers to objects. The comparison routine should return an integer greater than 0 if the first item is greater than the second, 0 if the objects are the same, and an integer less than zero if the first item is less than the second.

`void (*destroyproc)(Pointer val):` The destroy procedure should take as arguments a Pointer (void *) to an instance of the object. *NOTE: This routine should free everything that the pointer points to.*

`Pointer (*copyproc)(Pointer val):` The copy procedure should take as arguments a Pointer to an instance of the object. It should return a Pointer to a safely SUIT_malloced data object of the newly registered type.

`Pointer (*defaultproc)(void):` This routine returns a pointer to the default value to be used whenever a new property of this type is created. Note that the routine should either allocate the memory for the pointer it returns, or return a pointer to static data.

`widgetClass:` This is a string that denotes a widget class that SUIT can use for exporting a property of the newly registered type. A widget of this class must carry a property called CURRENT_VALUE and this property must be of the newly registered type. Setting the "widgetClass" parameter to NULL denotes that there is no such widget and that exporting properties of this type can not be done.

See File: **`NewType.c`**

See Also: **`SUIT_convertType()`** page 78

`SUIT_viewport SUIT_defViewport(int x1, int y1, int x2, int y2)`

Description: This function defines a data object of type SUIT_viewport, given the coordinates of the lower left and upper right hand corners. All values are given in pixel coordinates as measured from the lower left hand corner of the application window.

Parameters:
`x1, y1:` the coordinates of the lower left hand corner of the window.
`x2, y2:` the coordinates of the upper right hand corner of the window.

See Also: The section on "Windows And Viewports″page 17

`SUIT_window SUIT_defWindow(double x1, double y1, double x2, double y2)`

Description: This function defines a data object of type SUIT_window, given the coordinates of the lower left and upper right hand corners.

Parameters:
`x1, y1`: the coordinates of the lower left hand corner of the window.
`x2, y2:` the coordinates of the upper right hand corner of the window.

See Also: The section on "Windows And Viewports″page 17

SUIT_springiness

Description: These constants control the way a child or employee widget will resize when its parent resizes. Specifying that a widget possesses springiness in a particular direction means that as the parent resizes in that direction, the child does not. These constants are fixed in a way that allows them to be combined with the bitwise and (&) and bitwise or (|) operators. Setting the springiness property from inside an application is best done interactively with the springiness widget.

Legal Values:

```
VERTICAL_SPRINGINESS        HORIZONTAL_SPRINGINESS
ABOVE_SPRINGINESS           BELOW_SPRINGINESS
LEFT_SPRINGINESS            RIGHT_SPRINGINESS
NO_SPRINGINESS              ALL_SPRINGINESS
```

Example:

```
SUIT_setSpringiness (obj, SPRINGINESS,
        LEFT_SPRINGINESS & BELOW_SPRINGINESS);
```

See Also: **SUIT_setSpringiness()** page 55

SUIT_trapperPtr

Description: This is a function pointer of the form

```
SUIT_object (*SUIT_trapperPtr) (SUIT_object obj, SUIT_event *ev)
```

This is a function that takes a SUIT_object and a pointer to a SUIT_event as parameters and returns a SUIT_object as a result.
This kind of function is used in conjunction with **SUIT_registerTrapper** (page 70).

SUIT_viewport

Description: This data type is exactly the same as a rectangle.

Fields:

```
typedef struct {
        point bottom_left
        point top_right;
} SUIT_viewport;
```

See Also: The section on "Windows And Viewports″page 17

SUIT_window

Description: This data type is exactly the same as a GP_rectangle.

Fields:

```
typedef struct {
        GP_point bottom_left
        GP_point top_right;
} SUIT_window;
```

See Also: The section on "Windows And Viewports″page 17

SUIT_event

Description: SUIT considers all mouse and keyboard strokes to be *events*. Each event has a type (see below) and a location. Some events, like keyboard strokes, also carry other information, such as which mouse or keyboard key was pressed.

Fields:

```
typedef struct input_event_str {
        SUIT_eventType      type;
        GP_point            worldLocation;
        point               relativePixelLocation;
        char                keyboard;
        int                 button;
        deluxe_locator_measurelocator;
} SUIT_event;
```

Remarks: **type:** The type of the SUIT event. Possible values here are:

MOUSE_DOWN	MOUSE_UP
MOUSE_MOTION	CLICK
KEYSTROKE	

worldLocation: This is the floating point location of the event, measured with respect to the window of the widget that received the event, which is usually a point between (0.0, 0.0) and (1.0, 1.0).

relativePixelLocation: This is the integer point location of the event, mapped from the current window to the current viewport of the widget that received the event. All pixel coordinated are measured with respect to the origin, which is the lower left hand corner of the application window.

keyboard: The keyboard character that was hit if the event was a keypress.

button: The mouse button that was pressed if the event was a mouse event. Legal values here are covered under the heading of mouse button names on page 32.

locator: The locator measure that describes this event in detail. For more information, see the description of a deluxe_locator_measure on page 30.

SUIT_permanence

Description: These values denote the two choices for property permanence. A property that is **PERMANENT** is one that is eventually written to the ".sui" file, meaning that the value of that property is preserved between invocations of the program. A value of a **TEMPORARY** property is lost between runs of the program because it is not written to the ".sui" file.

Legal Values: **PERMANENT**
TEMPORARY

See Also: These are used in all of the SUIT deluxe set and get calls. See page 57 and page 63.
SUIT_makePropertyPermanent() page 66
SUIT_makePropertyTemporary() page 66

SUIT_eventType

Description: These are the names of the different kinds of SUIT events. Do not confuse this with the structure called a SUIT_event.

Legal Values:

MOUSE_DOWN	**MOUSE_UP**
MOUSE_MOTION	**CLICK**
KEYSTROKE	

SUIT_saveStatus

Description: These codes are used in calls to **SUIT_done()**. The first exit code denotes that the ".sui" file is not rewritten and that all changes to the interface are lost. The second code denotes writing a new ".sui" file and making a backup copy of the old one.

Legal Values: **DO_NOT_SAVE_SUI_FILE**
SAVE_SUI_FILE

See Also: **SUIT_done()** page 50

SUIT_exitStatus

Description: These codes are used in calls to **SUIT_done()**. The first exit code denotes that SRGP, GP and SUIT should close down, but should leave the application running. What happens after that is the responsibility of the programmer. The second code forces SUIT to call **exit(0)**.

Legal Values: **DO_NOT_EXIT_APPLICATION**
EXIT_APPLICATION

See Also: **SUIT_done()** page 50

SUIT_objectInterestCallback

Description: This is a function pointer of the form

```
void (*SUIT_objectInterestCallback)(SUIT_object obj,
char* propertyName, char* propertyType,
Pointer newValue, Pointer oldValue)
```

This kind of function is called when an interest is registered with an object using **SUIT_resisterInterest()**. See page 74 for details.

SUIT_level

Description: These values denote the four levels at which a SUIT property may be found. In searching for the value of a property, SUIT first looks at the **OBJECT** level, followed by **CLASS** and then at the **GLOBAL** level.

Legal Values: **OBJECT**
CLASS
GLOBAL

See Also: These are used in all of the SUIT deluxe set and get calls. See page 57 and page 63.

SUIT Constants and Types

`boolean`

Legal Values: **`TRUE`**
`FALSE`

`Pointer`

Description: This is syntactic sugar for a generic pointer to a data object: (void *).

`Reply`

Description: These are the values that come back from the SUIT dialog box calls.

Legal Values: **`REPLY_NO`**
`REPLY_YES`
`REPLY_CANCEL`
`REPLY_OK`
`REPLY_BUTTON1`
`REPLY_BUTTON2`

See Also: The SUIT dialog box calls on page 123.

`SUIT_callbackFunctionPtr`

Description: This is a pointer to a function that takes a single **`SUIT_object`** as a parameter and returns **`void`**.

Example: **`void MyFunction(SUIT_object obj)`** is a callback. A pointer to this function is a **`SUIT_callbackFunctionPtr`**. Could be used, for example in a call to

```
SUIT_createDoneButton(MyFunction);
```

`SUIT_validationFunction`

Description: This is a pointer to a function that takes a single **`SUIT_object`** as a parameter and returns **`boolean`**.

See Also: **`SUIT_createOKCancelDialogBox()`**

`SUIT_mouseMotion`

Description: These values are used to describe the various ways that mouse motion can be reported to a widget. **`WHILE_MOUSE_DOWN`** (report mouse motion while the mouse button is down over the widget), and **`UNTIL_MOUSE_UP`** (report mouse motion until the mouse button comes up, regardless of the widget the mouse is over)

Legal Values: **`WHILE_MOUSE_DOWN`**
`UNTIL_MOUSE_UP`

See Also: **`SUIT_reportMouseMotion()`** page 69

GP_rectangle

Remarks: This is a rectangle in world coordinates.

Fields:

```
typedef struct {
        GP_point bottom_left;
        GP_point top_right;
} GP_rectangle;
```

See Also: **GP_defRectangle()** on page 85.

GP_time

Operations:

GP_getCurrentTime()	page 106
GP_timeDifference()	page 106
GP_convertTime()	page 106

write mode

Legal Values: **WRITE_REPLACE** **WRITE_XOR**
WRITE_OR **WRITE_AND**

GP_color

Fields:

```
typedef struct {
        char *colorName;
        boolean blackOnMonochrome;
} GP_color;
```

Remarks: Colors are described by a string name, which is one of the standard X windows strings for color names and whether the color appears as black on a monochrome screen. Colors may also be defined using an RGB triple. To make the code more readable, you can use **BLACK_-ON_MONO** and **WHITE_ON_MONO** rather than **TRUE** or **FALSE**.

See Also: **GP_defColor()** page 84
GP_defColorRGB() page 85
GP_describeColor() page 97

GP_font

Remarks: A GP font is a typeface, style, point size.

Operations: **GP_defFont()** page 85
SUIT_getFont() page 61
SUIT_setFont() page 55
SUIT_deluxeGetFont() page 64
SUIT_deluxeSetFont() page 58

GP_justification

Description: These values are used in the **SUIT_justifyText** function covered on page 101.

Legal Values:

JUSTIFY_BOTTOM_LEFT	**JUSTIFY_BOTTOM_CENTER**	**JUSTIFY_BOTTOM_RIGHT**
JUSTIFY_CENTER_LEFT	**JUSTIFY_CENTER**	**JUSTIFY_CENTER_RIGHT**
JUSTIFY_TOP_LEFT	**JUSTIFY_TOP_CENTER**	**JUSTIFY_TOP_RIGHT**

See Also: **SUIT_justifyText()** page 101

GP_point

Remarks: This is a point in world coordinates.

Fields:

```
typedef struct {
        double x;
        double y;
}GP_point;
```

See Also: **GP_defPoint()** on page 85.

`mouse button names`

Description: These constants name the three buttons of the mouse.

Legal Values:

```
LEFT_BUTTON
MIDDLE_BUTTON
RIGHT_BUTTON
```

`mouse button modifiers`

Description: These constants are used to index into an array contained inside of a SUIT_event (see page 37). The array a stores the status of each of the keys below when a mouse button was hit. In this way, the programmer can differentiate between a mouse "click" and a "shift click."

Legal Values:

```
SHIFT
CONTROL
META
```

Remarks: On some keyboards, the META key is labeled "Alt".

Example:

```
/* evt is a SUIT_event, passed into this function;
   a hit proc most likely */

/* test for shift click */
if (evt.locator.modifier_chord[SHIFT] && evt.type == CLICK) {
    /* user has SHIFT-clicked the mouse */
```

`point`

Description: This data structure describes a point in pixels.

Fields:

```
typedef struct {
        int x;
        int y;
} point;
```

`rectangle`

Description: This data structure describes a rectangle in pixels.

Fields:

```
typedef struct {
        point bottom_left;
        point top_right;
} rectangle;
```

`shapes of cursors`

Legal Values:

```
STANDARD_CURSOR     PIRATE_CURSOR
WATCH_CURSOR        PROMPT_CURSOR
RIGHT_ARROW_CURSOR
```

See Also: `GP_setCursor` on page 90.

fill styles

Description: The fill style controls the way filled polygons, rectangles and ellipses will appear on the screen. Drawing an entity in **SOLID** fill style replaces existing pixels with a solid flood in the current color. **PIXMAP_PATTERN** replaces pixels with the pattern for the currently defined pixmap pattern. **BITMAP_PATTERN_TRANSPARENT** uses the currently defined bitmap pattern, and considers the "background" color pixels in the pattern to be transparent; pixels that were on the screen will show through. **BITMAP_PATTERN_OPAQUE** replaces the existing region with the existing bitmap. "Background" colors in the bitmap are considered opaque and will replace existing screen pixels.

Legal Values:
SOLID
PIXMAP_PATTERN
BITMAP_PATTERN_TRANSPARENT
BITMAP_PATTERN_OPAQUE

See Also:
GP_setFillStyle() page 90
GP_setPenStyle() page 92

input devices

Description: These are the devices that GP understands.

Legal Values:
NO_DEVICE
KEYBOARD
LOCATOR

See Also: **GP_setInputMode()** page 91

inputMode

Description: These values denote the three ways that mouse input can be handled.

Legal Values:

INACTIVE	turns the mouse off
SAMPLE	mouse sends back events as it moves
EVENT	mouse only sends back button events

See Also: **GP_setInputMode()** page 91

line styles

Description: These constants are used to set the line style that will be used on subsequent calls to any of the primitive drawing functions.

Legal Values:

CONTINOUS	**DASHED**
DOTTED	**DOT_DASHED**

marker styles

Description: These constants are used to set the current style of marker that will be drawn on subsequent calls to **GP_marker()** or **GP_markerCoord()**.

Legal Values:
MARKER_CIRCLE
MARKER_SQUARE
MARKER_X

GP Constants and Types

buttonStatus

Description: These are the two positions that a mouse button can be in.

Legal Values: **UP**
DOWN

canvasID

Description: This is an integer.that identifies a canvas to draw on. By default, the canvas of the screen has a canvasID of 0.

See Also:

GP_createCanvas() page 96
GP_useCanvas() page 96
GP_deleteCanvas() page 96
GP_inquireActiveCanvas() page 94
GP_inquireCanvasExtent() page 94
GP_inquireActiveCanvasSize() page 94

deluxe_locator_measure

Description: A deluxe_locator_measure is one of the fields found inside a SUIT_event and it details exactly what happened with a mouse button or keyboard key. Examining this structure is sometimes useful in determining whether certain mouse buttons are up or down.

Fields:

```
typedef struct {
        point position;
        buttonStatus button_chord[3];
        int button_of_last_transition;
        buttonStatus modifier_chord[3];
        srgp_timestamp timestamp;
} deluxe_locator_measure;
```

keyboard_measure

Description: This is the struct that is returned for a keyboard event. Notice that the status of the buttons is recorded as well to aid in trapping “key+mouse” operations.

Fields:

```
typedef struct {
        char *buffer;
        int buffer_length;
        buttonStatus modifier_chord[3];
        srgp_timestamp timestamp;
} keyboard_measure;
```

SUIT and GP Types and Constants

What are the SUIT-command keys?

Below is a complete list of all the SUIT command keys:

Operation	**What it Does**	**Hot Key**[1]
SUIT menu	*invokes the SUIT menu, which contains most of the following functions*	SUIT-M
cycle	*change a widget's display style*	SUIT-C
align	*lines up selected widgets by tops, bottoms, etc.*	SUIT-A
send to back	*selected widget goes behind all others.*	SUIT-B
bring to front	*selected widget goes in front of all others*	SUIT-F
select widget	*marks a widget as selected; deselects a widget if already selected; selects all widgets if cursor is over no widget*	SUIT-S
redraw	*repaints all widgets*	SUIT-R
edit properties	*examine and alter a widget's properties*	SUIT-E
get info	*prints information about a widget in a dialog box*	SUIT-I
open widget	*opens up a parent widget so that the children may be accessed*	SUIT-O
close widget	*closes a parent widget that was opened with SUIT-o*	SUIT-K
create new widget	*creates a new SUIT object on the fly*	SUIT-N
destroy	*destroys a SUIT widget*	SUIT-X
version	*prints the version of SUIT you are using*	SUIT-V
duplicate	*creates a copy of a SUIT widget*	SUIT-D
quit application	*save the ".sui" file and leave the application.*	SUIT-q

1. "SUIT" is shorthand for holding down the SHIFT and CONTROL keys simultaneously.

SUIT provides a number of dialog box calls that will help you with some common “safe” moded interactions (e.g. getting an OK confirmation from the user for a dangerous action). If you absolutely must make panels come and go under program control, you can refer to the file called

See Example File: `moded.c`

What does “SRGP: Color Table Too Full to share” mean?

It means that there is some other application running that uses a lot of colors (a complex background with a lot of colors is a common culprit). Unless you tell SUIT otherwise, SUIT will attempt to allocate 7 bitplanes for the SUIT color table (which is maintained by the SRGP graphics package). This allocates $2^7 = 128$ colors for you, but in the presence of another color-hungry application, there may not be that many free colors available.

The solution is to hand edit the “.sui” file, changing the lines that read

```
#define THE_SCREEN_DEPTH  7
```

to

```
#define THE_SCREEN_DEPTH  <some number lower than 7>
```

What if the widget I am editing covers the whole Property Editor?

Press SUIT-B to send the widget to the back (behind the property editor).

How Do I set A Property to a Default Value?
How Do I keep the .sui File from Resetting a Property?

The “.sui” file is read at the beginning of the call to **`SUIT_beginStandardApplication()`**. If you need to reset some properties after the “.sui” file is read, you can use the expanded version of the SUIT main loop like so:

```
SUIT_beginDisplay();
/* Set widget defaults here */
SUIT_setBoolean(my_obj, HAS_BORDER, FALSE);
while (TRUE) {
  SUIT_checkAndProcessInput();
}
```

Of course, you can always call SUIT_makePropertyTemporary() to keep the property from being written to the “.sui” file at all.

How do I force the display to act like monochrome if I'm on a color system?

Hand edit the ".sui" file and change the line that reads

```
#define THE_SCREEN_DEPTH  7
```

to

```
#define THE_SCREEN_DEPTH  1
```

How do I resize the application window?

(FOR WINDOWING ENVIROMENTS ONLY)
There are two ways to do this:

1.) You can resize the application window as you usually would (depends on the window manager, but typically involves dragging one corner of the window to a new location), then exit the applicaton. SUIT remembers the last size of the application window and uses it at startup.

2.) You can edit the SUI file directly. There are two lines at the beginning of the SUI file that read

```
#define THE_SCREEN_WIDTH <some number>
#define THE_SCREEN_HEIGHT <some other number>
```

By changing the numbers here, you can get a precise value for the size of the application window.

How do I create a bounded value with a textual readout?

See the example file provided with SUIT called BVLabel.c

<u>**See Example File:**</u> **`bvlabel.c`**

Type in Boxes keep coming up with the text from the last time I ran the program. How do I make sure that the type-in boxes are empty?

You need to make the CURRENT_VALUE property temporary, so it won't get written to the ".sui" file each time the program ends. To see how to do this,

<u>**See Example File:**</u> **`tempprop.c`**

How do I make a panel of widgets appear and disappear under program control?

Be careful here. Making widgets come and go can lead down the dark and dangerous path to the dreaded "moded interface." A moded interface, loosely speaking, is one where the user is prevented from taking certain actions because certain other actions are taking place. For example, imagine a drawing/text-editing application with two panels of widgets -- one for drawing graphics and one for editing text. It would be a poor interface indeed if the user was prevented from bringing up both panels of widgets at once. If bringing up one panel caused the other one to vanish, the interface would be "moded."

This kind of "one at a time" interaction is attractive from a programming point of view in that it seems to make the program easier to code (in fact, it rarely does), but it is very difficult and confusing to *use* a program written this way. If you're programming this way, you might want to re-read the tutorial appendix article, "An Introduction to External Control."

What are PERMANENT and TEMPORARY properties?

A property is permanent if the value of the property is to be written to the ".sui" file before the program exits. This way, the value can be restored when the program is next invoked. TEMPORARY properties are not written to the .sui file. See **`SUIT_makePropertyPermanent()`** and **`SUIT_makePropertyTemporary()`**.

Are SUIT object names case sensitive?

No.

Are SUIT property names case sensitive?

No.

Are SUIT type names, when they are represented as strings, case sensitive?

No.

What does "SUIT has detected an error..." mean? Is this a bug in SUIT?

Messages that look like:

```
* SUIT has detected an error at line 62 in the SUIT source file tree.c:
*
* SUIT_createObject was called with the name of an existing object
(`foo').
*
* In most cases, this type of error indicates that you have
* made a mistake when calling the SUIT library routines.
```

are almost always caused by errors in the application program, not in SUIT. For example, you might pass **`GP_rectangle()`** two points out of order (e.g. top right, bottom left, instead of bottom left, top right). Each kind of error comes with a detailed error message that should help you discover what caused the problem.

What do I do if I find a bug in SUIT?

By all means, report the bug.

It is important that you assume that we never heard of the bug so that we can have the fullest description of the problem.

Reports can sent via electronic mail to:

`suit@uvacs.cs.Virginia.EDU`

We appreciate any and all bug reports you might have. In order to consider your report, we'll need the following information:

- Machine Type (Sun, SGI, Mac, RS/6000, etc.)
- Color or Monochrome System
- A code snippet from the program that exhibits the problem.
- A detailed desctription of the way that the problem can be brought forth.

Can I Turn off the Property Editor?

Yes, see the section on shipping an application on page 161.

How do I draw with pixels rather than GP's floating point numbers?

SUIT uses a floating point coordinate system entirely as a convenience. If you require access to the individual pixels of the screen, you can make calls to SRGP, the Simple Raster Graphics Package, which is SUIT's underlying graphics library. All of GP is written in SRGP. Documentation for SRGP is available as an addendum to the SUIT Reference Guide.

How do I make labels with more than one line of text?

Labels, by definition, can only display one line of text. To display several lines of text, use **`SUIT_createTextBox()`**. The text in this box is allowed to contain newlines as well as the special formatting characters detailed on page 104. For an example of **`SUIT_createTextBox()`**, see the file **`Events.c`**.

See Example File: **`events.c`**

How do I add "Info" help strings to the properties I create?

Use **`SUIT_registerHelp()`** to add help strings to SUIT's database of property information. You can retrieve help strings by using **`SUIT_getHelp()`**. Both functions are described on page 80.

How do I move / resize / remove the child widgets of a bulletin board?

Use "SUIT-o" to open the bulletin board (or any widget with children). A border will appear around the inside of parent widget, letting you know that the widget is open. Once the parent is open, you can SUIT-drag on the children to move and resize them. Use SUIT-k to close the bulletin board.

Why do the child widgets of my bulletin board show up in the right place, but the wrong size?

You're probably setting the SHRINK_TO_FIT property to TRUE and that's getting you into trouble. The widget gets sized with a **`SUIT_setViewport()`** call, but then the widget resizes to obey the SHRINK_TO_FIT property. This causes the widget's sides to collapse around the widget's center, making the widget the wrong size, according to the **`SUIT_setViewport()`** call. Solution: Set SHRINK_TO_FIT to be FALSE and throw away the VIEWPORT property in the property editor.

When can I make and destroy widgets in my code?

Widgets can be created at any time in your program after the call to **`SUIT_init().`**

When is the ".sui" file read?

SUIT reads the ".sui" file in the call to **`SUIT_beginDisplay()`**. This function is called for you in the "convenience function" **`SUIT_beginStandardApplication()`**.

What are Locked Properties?

A property is said to be locked when the property cannot be changed in the SUIT property editor. Locking a property has the side effect of not allowing that property to be exported. You can lock and unlock properties under program control using the **`SUIT_lockProperty()`** and **`SUIT_unlockPropoerty()`** calls. Properties that are locked can still be changed under program control.

How do I use SUIT's menu widget?

See Example File: `menu.c`

How do I use SUIT's radio button widget?

See Example File: `radio1.c and radio2.c`

How do I use SUIT's scrollable list widget?

See Example File: `list.c`

How can I tell when a user has pressed a SHIFT-click?

See Example File: `events.c`

How do I create "HotKeys" for my buttons?

SUIT implements "hot keys" with something called a "trapper function" If you register a trapper function with SUIT using the **`SUIT_registerTrapper()`** call (see page 70), you can test for the occurrence of certain "special" keyboard and mouse events.

See Example File: `trapper.c`

What are "interests" and what are they good for?

Suppose you wanted to be sure that widget A always remained 10 pixels away from widget B. Wouldn't it be nice if you could be informed somehow every time the viewport property of widget B changed? That way, you could change A's viewport so that A followed B around. In SUIT parlance, we say that A "has an interest in" B. We use **`SUIT_registerInterest()`** to register an interest function that will get called every time any property of B changes. Interests are a simple way of enforcing constraints in SUIT.

See Example File: `interest.c`

How can I make one word in a label italic while the others are not?

GP_text supports a small, extensible notation for changing the attributes of text within a text string. This notation allows for underlines, italic, boldface, accents and even special characters like yen (¥) and pounds sterling (£) signs. You can also register your own accent marks using the function calls provided. For more information on this GP_text notation, see page 104.

See Example File: `events.c (look at the text box widget)`

How do I center text inside a button or label?

Set the JUSTIFICATION property of the label to center and SHRINK_TO_FIT to false.

How do I keep a property from being written to the SUI file?

Use **`SUIT_makePropertyTemporary()`**. See page 66.

Frequently Asked Questions

In this section, we will answer the most popular SUIT related questions. The more involved answers are treated in great detail in the SUIT examples directory. Here you will find generously commented C programs that perform one function: the use of trappers, the use of menus, etc. The hope is that you will be able to copy the code directly into your own application with very little revision.

How Do I Create My Own Widgets?

There are several steps you need to take in order to create a widget of your own:

1.) Use **`SUIT_createObject()`** to create a new SUIT_object. You will provide a name for that widget and a new class that the widget will belong to. For example, you might create a new widget that implements a joystick-like control. Such a widget might be called “my joystick” and would belong to a class of widgets called “joystick”

2.) Use **`SUIT_addDisplayToObject()`** to register a display style for this new class of widget. All widget classes must have at least one display style. By convention, if a widget has but one display style, that style is called “standard”.

3.) Write a hit procedure for the widget. Such functions take the form:

```
void func (SUIT_object me, SUIT_event ev)
```

Hit procedures typically only set properties of a widget.

4.) Write a paint procedure for the widget. Such functions take the form:

```
void func (SUIT_object me)
```

Paint procedures typically use GP graphics calls to paint graphics based on the state of the widget’s properties. The widget’s property values are obtained through the SUIT_get-Property calls detailed onpage 60.

5.) If you want to be able to create your widget interactively, use **`SUIT_registerClass()`** to register the new widget class with SUIT. The creation proc this function takes as a parameter is the subject of the following note. Also see page 81.

NOTES:

1.) Usually, the **`SUIT_createObject()`** and **`SUIT_addDisplayToObject()`** calls are placed in a widget creation call of the form:

```
SUIT_object createNewWidget (char *name)
```

Notice that the function takes a string that can be used for this widget’s unique name and returns a **`SUIT_object`**. This will make your widget creation call as easy to use as SUIT’s own widget creation calls:

```
SUIT_object first, second;

first = createNewWidget (“first”);
second = createNewWidget (“second”);
```

2.) Paint procedures should NEVER set properties of a widget. When a widget property changes, the SUIT automatically flags the widget as needing a redisplay. This, in turn, will eventually cause the widget’s paint procedure to be called. Clearly, if a paint procedure is setting properties, this can cause the redisplay required flag to always be set which can send the application into an infinite loop. Remember, if a “SUIT_setXXX” calls doesn’t actually change a property value (i.e. it sets a property to the same value) then the redisplay flag is not set.

<u>See Example File:</u> **`NewWidgt.c`**

Converting From Windows to Viewports and Back

On occasion, it may be necessary to convert viewport pixel coordinates to the corresponding floating point coordinates in the SUIT_window where the graphics are drawn. To do this conversion, SUIT provides a complete set of conversion routines that map X and Y floating point coordinates to their corresponding integer coordinates as well as routines that unmap integers to floating point. For more information, see page 86.

Viewports And Hierarchy

SUIT objects can be nested inside other SUIT objects for purposes of creating composite widgets. When one widget is contained by some other widget, the **contained** widget's viewport is measured from the lower left hand corner of the **container** widget, not of the screen. There is a complete discussion of viewports and hierarchy in the section covering hierarchy, starting on page 153.

Summary of Terms

SUIT_viewport: A rectangular region on the screen, measured in integer (pixel) coordinates. The origin of pixel coordinates is usually the lower left hand corner of the application window, or in the case of non-windowing environments, of the screen. Viewports for hierarchical widgets are measured relative to the lower left hand corner of the parent, or container widget, not of the screen.

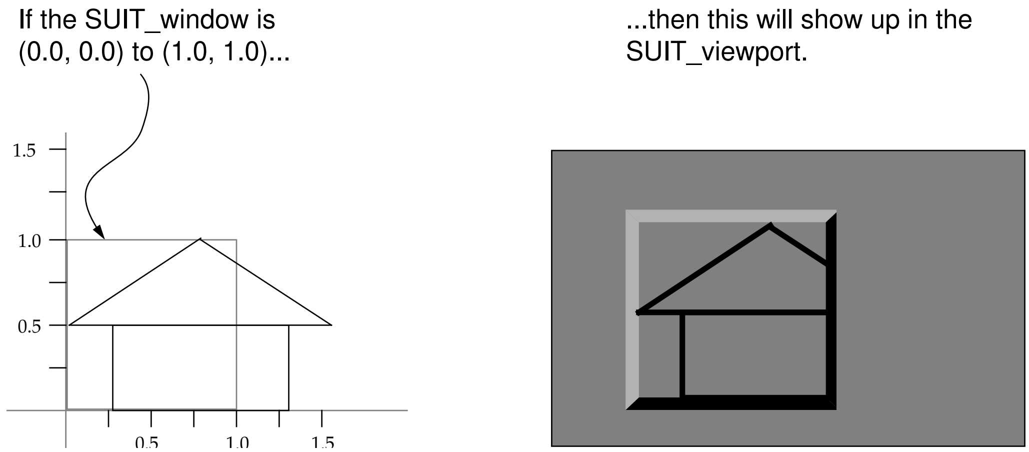
SUIT_window: A rectangular region of the world coordinate system, measured in floating point coordinates. A SUIT_window defines how much and what part of the world is to be mapped to the screen.

GP_rectangle: A rectangular region of the world coordinate system, measured in floating point units. Usually meant to refer to a graphical primitive supported by the GP drawing package.

rectangle: A rectangular region of the screen measured in pixel coordinates. The origin is always in the lower left hand corner of the application window, or of the screen (in non-windowing environments). This is a data type that only needs to be used when calling SRGP directly. SRGP is described in the section called "Drawing In Pixels: SRGP" on page 159.

SUIT_windows

So now, we have a drawing that is specified in world coordinates and a region of the screen that is specified in pixels. The only remaining question is: "How do we convert the floating point drawing into pixels on the screen?" Clearly, we can't transfer all of the world coordinate system to the screen; world coordinates are infinite in all directions, and we know that the viewport is finite. We must choose some finite part of the world coordinate system, a *SUIT_window*, that will get transferred to the viewport on the screen. The graphics that fall inside this world coordinate window will be the graphics that appear inside a widget's viewport. We usually arrange the drawing so that all of the graphics we are interested in will fall inside the boundaries of the SUIT_window; everything outside the window is blank. All SUIT widgets carry with them a property of type SUIT_window called WINDOW. Using the example from the previous page:



By default, all widgets are created with their SUIT_window set with the lower left hand corner at the world coordinates origin (0.0, 0.0) and the upper right hand corner at (1.0, 1.0). This means that in order for your GP graphics to show up on the screen, you need to make sure that the coordinates that you use fall in the range of 0.0 - 1.0. You can change this default by setting the "window" property of a widget to some other value:

```
SUIT_setWindow(mywidget, WINDOW, GP_defWindow(x1, y1, x2, y2));
```

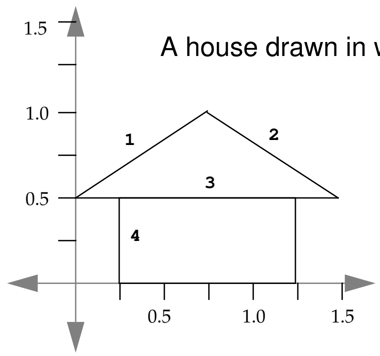
NOTE: If you are used to working with windowing systems (X windows, Macintosh, etc.), you may be accustomed to thinking that a window is a region on the screen. *This is not the case here; in SUIT, a window is a region of the world coordinate system.* Regions on the screen are viewports.

The Size of a Window

Choosing the size of a window is very much like choosing a scale at which to draw. Most of the time is doesn't matter because there are no "real world dimensions" associated with the widget; a coordinate like "0.50" means 50% of the way across the widget, no more. For widgets that refer to real world objects (e.g. a widget that allows the user to pan across the surface of a piece of paper as in a paint program like idraw or MacPaint) it may turn out that the default window size (0.0, 0.0) to (1.0, 1.0) is inappropriate. To scale the window to be in proportion to an 8-1/2" x 11" piece of paper you would choose a window size of: (0.0, 0.0) to (8.5, 11.0). To change the minimum and maximum values of a SUIT window belonging to a widget, use the **`SUIT_setWindow()`** and **`SUIT_getWindow()`** functions described on page 62 in the *SUIT Reference Manual*.

For example, here is the code for drawing a house:

```
GP_lineCoord (0.0, 0.5, 0.75, 1.0);
GP_lineCoord (0.75, 1.0, 1.5, 0.5);
GP_lineCoord (1.5, 0.5, 0.0, 0.5);
GP_rectangleCoord(0.25, 0.0, 1.25, 0.5);
```



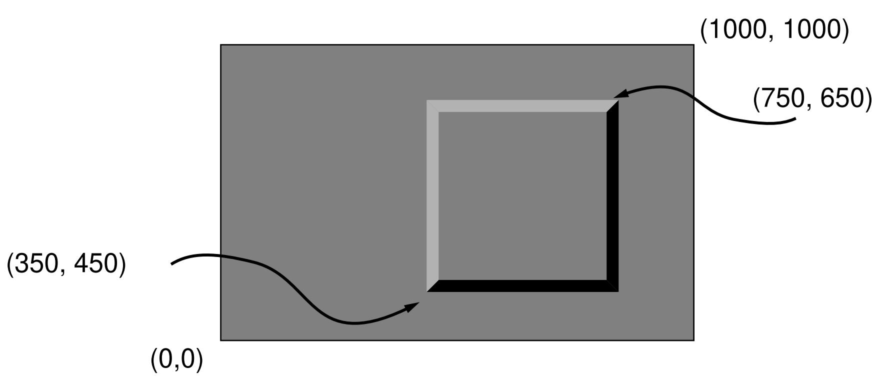
A house drawn in world coordinates.

The world coordinates are infinite in all directions.

The last entity drawn in this example is a *GP_rectangle,* which is just a rectangle defined by two pairs of floating point coordinates, one for the lower left hand corner, one for upper right. Notice that this coordinate system says nothing about where on the screen the house is to appear. Placing GP graphics on the screen is done by choosing a SUIT viewport.

SUIT_viewports

A SUIT_viewport is a *rectangular region on the screen*, measured in pixels. On a 1000x1000 pixel screen, a typical viewport might have its lower left hand corner at (350, 450) and its upper right hand corner at (750, 650). For nearly all widgets, the origin is at the lower left hand corner of the application window (or of the screen in a non-windowing environment). We say “nearly all” because things are slightly more complicated for SUIT widgets that are nested inside other widgets (so called “hierarchical widgets”). We will address this very minor point in the section on hierarchy on page 153. For now, just remember that a SUIT_viewport is nothing more than the rectangular piece of screen real estate that a widget occupies. Clearly, each widget in SUIT (buttons, sliders, labels, etc.) has its own viewport. This viewport information is something that each widget carries with it in the form of a SUIT property simply called VIEWPORT.



Typical Viewport Coordinates. Pixel Coordinates for viewports increase from the lower left to the upper right.

Windows, Viewports, and Rectangles

SUIT uses four different types of rectangles: SUIT_windows, SUIT_viewports, GP_rectangles, and rectangles. This section defines each of these data types and explains how they are related. Understanding the difference between these rectangles starts with understanding the way SUIT draws graphics.

If you already understand the process of window to viewport mapping, and you just need quick definitions of the terms, turn to page 20 for a summary.

The Case Against Pixel Based Drawing

Suppose you wanted to draw graphics on a bitmapped display. Probably the simplest way of doing this is to specify the pixel positions you want to light up. For example, you might easily envision a line drawing routine that looks like: "line(10, 10, 30 30)" which would draw a line from pixel (10, 10) to pixel (30, 30). There are at least two problems with this:

Not very portable: Graphics systems vary widely in resolution. If you specified lines that went from pixel (0, 0) to pixel (800, 800) such lines would run off the screen on a VGA equipped PC. Furthermore, a 600x400 image fills a Macintosh screen, but an image with the same pixel dimensions on a Sun Unix workstation approximately fills only the lower left hand corner of the screen.

Resize makes for blocky graphics: If a picture is displayed using pixel coordinates, and the pixel is scaled up to twice its size, the only real choice the software has is "pixel replication", whereby each pixel is replaced by a block two pixels wide and two pixels high. This leads to blocky, choppy pictures.

The Solution: World Coordinates

To avoid the problems of portability and image quality that a pixel based drawing system can have, we need a way to draw graphics that doesn't depend on the size or resolution of the screen. SUIT meets this need by avoiding drawing in pixel coordinates whenever possible; instead of an integer (pixel) based coordinate system, SUIT uses a *floating point* coordinate system called the *world coordinate system*.

The best way to picture the world coordinate system is to imagine that you are drawing the graphics on a large piece of graph paper, somewhere off screen. You can choose the relative sizes and locations of everything you draw using floating point numbers, without needing to be bothered about exactly which pixels need to be turned on in order to display your picture. The commands you use to create these pictures will come from SUIT's graphics library, a package called "GP." There are GP_lines, GP_ellipses, GP_rectangles and the like, all specified with floating point, rather than pixel coordinates for their parameters.

Example Programs

SUIT comes with a wide selection of example programs that demonstrate how some of the trickier facets of SUIT can be used. Below is a listing of the example programs that are available in the current release of SUIT in the src/examples directory :

Widget Demos

button	- Using buttons and callbacks
typein	- Using Type in boxes
dialog1	- Using DialogBoxes
dialog2	* Using dialog boxes with data validation functions
menu	- Using Menus
radio1	- Using RadioButtons with arrays
radio2	* Using RadioButtons with SUIT_enums
myWidget	- How to make a new widget
bulboard	- Using Bulletin Boards
scroll	- Using Scrollable lists of text
onoff	- Using on/off switches
bounded	- Using bounded values
color	- Using color chips
pattern	- Using pattern chips
filebox1	- Using SUIT_askForFilename()
filebox2	* Using more sophisticated filebox commands

Concept Demos

coords	- Using world and pixel coordiantes
events	- Code that tests for different kinds of SUIT events
trapper	- Implementing hotkeys in SUIT
move	- Using SUIT_moveRectangle()
resize	- Using SUIT_resizeRectangle()
interest	* Simple Constraints in SUIT
drag1	- Using SUIT"s dragging routines
drag2	* Using more advanced dragging routines'
dragdrop	- How to tell if the user drags one widget over another
3dfont	- A 3Dfont viewer -- shows 3D transformations
bvlabel	- Attaches a text readout to a bounded value
dyndemo	* Using DynArrays.
gpdemo	- How to use SUIT's GP graphics package
textlist	* Using SUIT_textLists
tempprop	* Making typein boxes that are blank at the start of an application
moded	- Making widgets come and go through the VISIBILE property. See Frequently Asked Questions.

(*) examples of SUIT"s more advanced and less often used features

Getting Started With SUIT

Read "An Introduction to External Control"

External control is unlike anything that a Pascal programmer is likely to have seen in the past, and because of that, it can be a tricky concept for new users. Because SUIT uses external control as a way of structuring program flow, we strongly recommend that new users take the time to read this short article that explains the ins and outs of this powerful programming model. "An Introduction to External Control" is an appendix to the SUIT tutorial.

Go Through The Tutorial

Don't fall into the trap of thinking that a tutorial is a waste of time for an experienced user. From the very beginning, we designed SUIT to be easy to learn and the tutorial was seen as the place where that learning would start. It takes most people only about an hour to complete, and contains a fair bit of the vocabulary that you will need to understand SUIT programming and the more advanced sections of the Reference Manual.

Look at the Example Programs

SUIT comes with a wide variety of example programs, all suggested by users like you. The next page has a list of the example programs that are available in the current release of SUIT. The code is meant to be simple to use and read; we encourage you to read the comments in these files and to copy the code you see into your own applications. Throughout the reference manual, you will find references to these helpful example programs. If you have an idea for a useful example, please send us e-mail (address below).

See the "Frequently Asked Questions" Section

Starting on page 21, there is a troubleshooting section that has answers to the most frequently asked questions about SUIT.

Send electronic mail to the mailing list

There is a mailing list for all current SUIT users. You can post questions and answers, make suggestions and swap code with other SUIT users worldwide. To become part of the mailing list, send your electronic mail address to

`suit-users-request@Virginia.EDU`

And to post messages to the list, send mail to

`suit-users@Virginia.EDU`

Send electronic mail to SUIT

When all else fails, feel free to send e-mail to:

`suit@uvacs.cs.Virginia.EDU`

We respond promptly to all requests, usually within 48 hours.

Concepts in SUIT

In this section you will find a series of chapters that can be read in most any order. They are meant to act as a collection of articles that present some of the more important ideas in the SUIT model that can be read on an as-needed basis.

Index To Frequently Asked Questions (see page 21)

T

U

V

W

Y

S

I

J

K

L

M

N

O

P

R

Index

The SUIT Widget Library

Appendix

Table of Contents

Acknowledgments

Thanks for SUIT are due to its original author, Nathaniel Young, and to Roderic Collins, Matt Conway, Dennis Cosgrove, Tom Crea, Jim Defay, Pramod Dwivedi, Robert DeLine, Brandon Furlich, Rich Gossweiler, Drew Kessler, Chris Long, William McClennan, Kim Passarella, Randy Pausch, Anne Shackelford and Scott Witherell. Thanks are also due to the UVa User Interface Group and the entire UVa CS department for their help in the extensive user testing that went into SUIT, The SUIT Reference Manual and the SUIT Tutorial.

This manual owes its existence to the generous support of many people, especially those in the UVa Department of Computer Science. We also would like to thank all the users of SUIT who took the time to make suggestions that improved the usefulness of the manual. Our thanks goes out to all of them and we encourage others to follow their lead. If you have a suggestion for a way to improve this manual, please send electronic mail to suit@uvacs.cs.virginia.edu.

Finally, we would like to thank those organizations who have helped fund SUIT:

National Science Foundation

Science Applications International Corporation

Virginia Center for Innovative Technology

Virginia Engineering Foundation

The SUIT Version 2.3 Reference Manual

This reference manual covers version 2.3 of the Simple User Interface Toolkit

---

NOTICE TO OUR READERS:

Any suggestions for improvements or bug reports are greatly appreciated and should be sent via e-mail to:

suit@uvacs.cs.Virginia.EDU
