# SUIT Function Calls

# 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.

## 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.

# 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.

# 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) ....
```

**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.

**`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);
}
```

# 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);
```

**`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.

### 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);
```

# 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_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)**

# 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);
```

# 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));
```

**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");
```

**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);`

# 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);`**

```
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)
```

# 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".

# 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.

# 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.

## 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.

# 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_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.

# 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.

# 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.

**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.

# 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.

# 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

# 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.

# 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;
```

**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.

**`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.

**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.

## 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.