SUIT and GP Types and Constants

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;
```

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

`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.

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.

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

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

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_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_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_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

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 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
*/
```

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`**.

`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.

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 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.