GP Function Calls

Type Definition Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

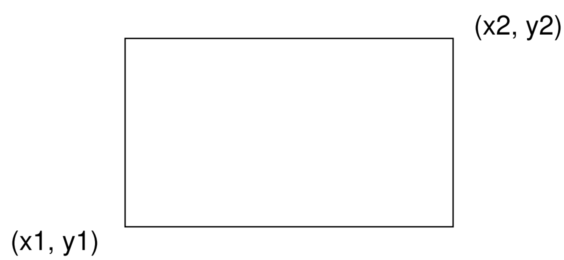
See Also: **`GP_setFont()`** page 90

GP_point GP_defPoint(double x, double y)

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

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

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



The corners of a GP rectangle

Mapping and Unmapping Rectangles and Points

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

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

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

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

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

Mapping →

Unmapping ←

`point GP_mapPoint(GP_point pt)`

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

`rectangle GP_mapRectangle(GP_rectangle r)`

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

`int GP_mapX (double x)`

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

`int GP_mapY (double y)`

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

`GP_point GP_unMapPoint (point pt)`

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

`GP_rectangle GP_unMapRectangle(rectangle rect)`

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

`double GP_unMapX (int x)`

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

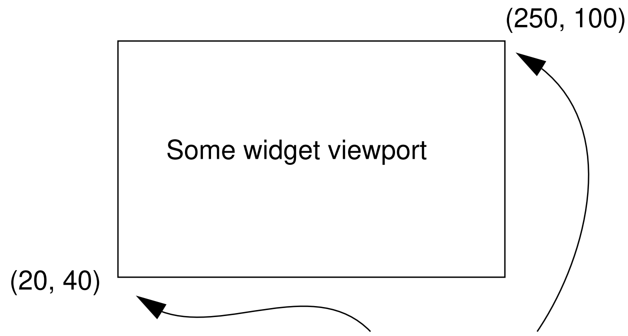
`double GP_unMapY (int y)`

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

Mapping and Unmapping Widths and Heights

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

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



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

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

Rarely Used

`int GP_mapHeight (double height)`

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

Rarely Used

`int GP_mapWidth (double width)`

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

Rarely Used

`double GP_unMapHeight (int height)`

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

Rarely Used

`double GP_unMapWidth (int width)`

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

Setting and Getting the Graphics State

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

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

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

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

Rarely Used

`void GP_popGraphicsState(void)`

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

Rarely Used

`void GP_pushGraphicsState(void)`

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

Example:

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

Rarely Used

`void GP_setClipRectangle(rectangle rect)`

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

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

void GP_setColor(GP_color clr)

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

void GP_setCursor(int cursorType)

Description: This function sets the current cursor shape.

Legal Values:

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

void GP_setFillBitmapPattern(int value)

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

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

void GP_setFillPixmapPattern(int value)

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

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

void GP_setFillStyle(int draw_style)

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

Legal Values:
SOLID
PIXMAP_PATTERN
BITMAP_PATTERN_TRANSPARENT
BITMAP_PATTERN_OPAQUE

int GP_setFont(GP_font newfont)

Description: This sets the value of the current font.

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

void GP_setInputMode(int device, int mode)

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

Legal Values: Devices:

NO_DEVICE	
KEYBOARD	
LOCATOR	This is the mouse

Modes:

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

void GP_setLineStyle(int line_style)

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

Legal Values: **CONTINUOUS**
DASHED
DOTTED
DOT_DASHED

void GP_setLineWidth(int newValue)

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

Rarely Used

void GP_setLocatorMeasure(point position)

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

void GP_setMarkerStyle(int marker_style)

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

Legal Values: **MARKER_CIRCLE**
MARKER_SQUARE
MARKER_X

See Also:

GP_setMarkerSize()	page 91
GP_marker()	page 102

void GP_setMarkerSize(int marker_size)

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

See Also:

GP_setMarkerStyle()	page 91
GP_marker()	page 102

`void GP_setPenBitmapPattern(int value)`

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

See Also: **`fill styles`** page 31

`void GP_setPenPixmapPattern(int value)`

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

See Also: **`fill styles`** page 31

`void GP_setPenStyle(int fill_style)`

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

Legal Values:
`SOLID`
`PIXMAP_PATTERN`
`BITMAP_PATTERN_TRANSPARENT`
`BITMAP_PATTERN_OPAQUE`

`void GP_setWindow (GP_rectangle newWorld)`

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

`void GP_setWriteMode(int write_mode)`

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

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

`void GP_setViewport (rectangle newView)`

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

Getting Graphics Attributes

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

`int GP_getColorIndex(char *name)`

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

---

`char *GP_getColorName(int index)`

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

`GP_color GP_getDepthColor (GP_color clr)`

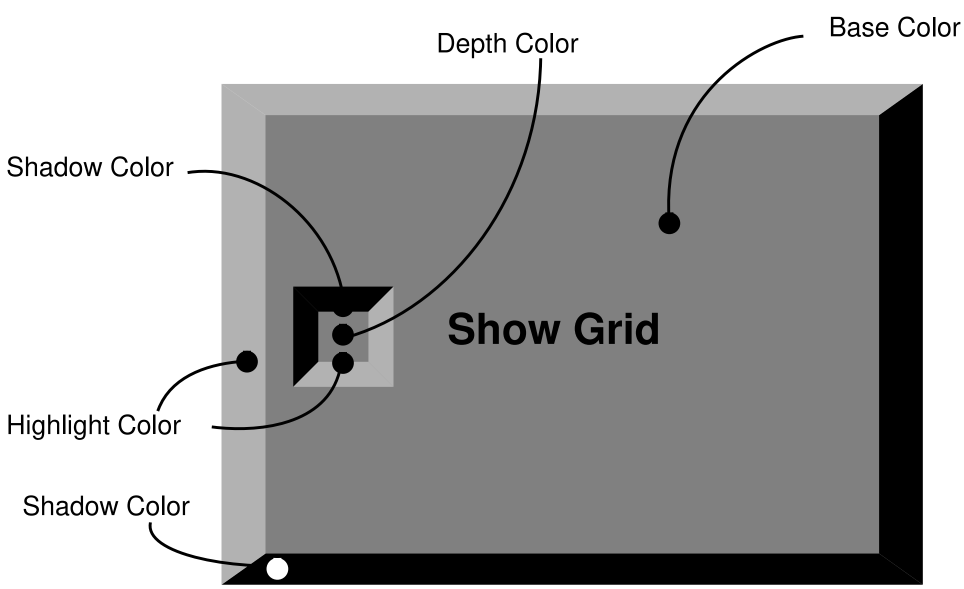
Description: Returns the appropriate motif depth color, given a base color. The depth color is the color one sees in a button during a mouse down event, when border raised is false.

`GP_color GP_getHighlightColor (GP_color clr)`

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

`GP_color GP_getShadowColor (GP_color clr)`

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



```
int GP_numColorsAllocated (void)
```

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

```
canvasID GP_inquireActiveCanvas(void)
```

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

```
rectangle GP_inquireCanvasExtent(canvasID whichCanvas)
```

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

```
int GP_inquireCanvasDepth(void)
```

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

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

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

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

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

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

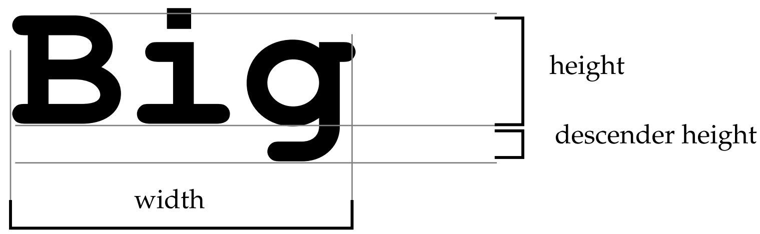
Example:

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

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

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

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



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

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

Canvas Control Functions

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

`canvasID GP_createCanvas(int width, int height)`

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

`void GP_deleteCanvas(canvasID whichCanvas)`

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

`void GP_useCanvas(canvasID whichCanvas)`

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

Color, Pattern, Cursor, And Font Table Functions

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

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

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

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

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

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

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

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

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

See Also: Names for the colors on page 84.

Drawing Primitives

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

`void GP_beep(void)`

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

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

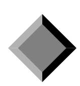
Description: This draws a beveled hollow rectangle.

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

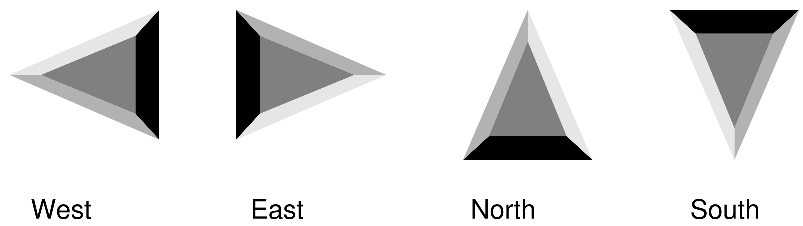
Description: This draws a beveled filled rectangle.

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

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



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




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

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

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

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

```
void GP_drawRectangle(GP_rectangle rect)
```

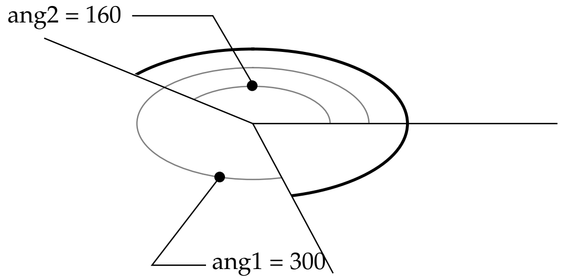
Description: This function draws an unfilled rectangle.

```
void GP_ellipse(GP_rectangle rect)
```

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

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

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



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

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

void GP_fillEllipse(GP_rectangle rect)

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

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

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

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

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

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

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

void GP_fillRectanglePt(GP_point bottom_left, GP_point top_right)

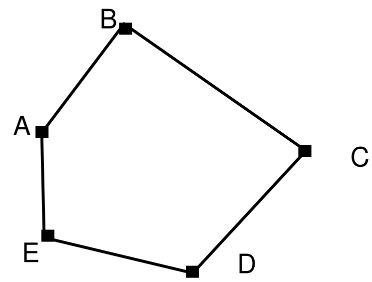
Description: This function draws a filled rectangle given the two points for the corners of the rectangle.

void GP_fillRectangle(GP_rectangle rect)

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

void GP_fillPolygon(int vertexCount, GP_point vertices[])

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



vertexCount = 5 (not 6)

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

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

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

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

void GP_justifyText(char *str, GP_justification just)

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

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

JUSTIFY_BOTTOM_LEFT	JUSTIFY_BOTTOM_CENTER	JUSTIFY_BOTTOM_RIGHT
JUSTIFY_CENTER_LEFT	JUSTIFY_CENTER	JUSTIFY_CENTER_RIGHT
JUSTIFY_TOP_LEFT	JUSTIFY_TOP_CENTER	JUSTIFY_TOP_RIGHT

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

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

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

JUSTIFY_BOTTOM_LEFT	JUSTIFY_BOTTOM_CENTER	JUSTIFY_BOTTOM_RIGHT
JUSTIFY_CENTER_LEFT	JUSTIFY_CENTER	JUSTIFY_CENTER_RIGHT
JUSTIFY_TOP_LEFT	JUSTIFY_TOP_CENTER	JUSTIFY_TOP_RIGHT

void GP_line(GP_point pt1, GP_point pt2)

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

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

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

`void GP_marker(GP_point)`

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

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

`void GP_markerCoord(double x, double y)`

Description: This draws a marker at the specified coordinates.

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

`void GP_pointCoord(double x, double y)`

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

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

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

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

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

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

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

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

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

`void GP_polyMarker(int numMarkers, GP_point vertices[])`

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

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

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

void GP_polyPoint(int vertexCount, GP_point vertices[])

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

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

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

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

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

void GP_rectanglePt(GP_point bottom_left, GP_point top_right)

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

GP_rectangle GP_text(GP_point pt, char *string)

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

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

On the screen:

This is <u>really</u> fun

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

GP Special Characters and Type Attributes

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

Special Characters

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

Styles

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

Accents

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

Advanced GP_text Functions

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

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

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

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

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

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

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


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

Time Functions

`GP_time GP_getCurrentTime()`

Description: This returns the current system time.

See Also: The definition of GP_time on page 34

`time_t GP_timeDifference (GP_time t1, GP_time t2)`

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

See Also: The definition of GP_time on page 34

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

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

See Also: The definition of GP_time on page 34