
SRGP for THINK C (v1.0)

David Frederick Sklar

The Simple Raster Graphics Package is composed of a library of functions, and a header file (“srgp.h”) that defines custom data types and constants, and which prototypes all SRGP routines. This paper is a complete but *extremely terse* description of the ANSI-C SRGP binding — this is not a tutorial. If you are new to SRGP, you must read Chapter 2 of *Computer Graphics — Principles and Practice* (Foley, van Dam, Feiner, and Hughes, Addison-Wesley, 1990).

If you wish to use the THINK Pascal version of SRGP, this document is still your primary resource. The few important points you need to know are in the README file distributed with the THINK Pascal version.

0) Contrast with Textbook Specification

1) States of the System

2) Canvases

3) Output

- 1) color
- 2) geometric data types
- 3) control of the pattern and font tables
- 4) control of attributes affecting output
- 5) generation of primitives
- 6) generation of primitives

4) The copyPixel Procedure

5) Input

- 1) properties of input devices
- 2) input modes
- 3) input devices
- 4) control of attributes
- 5) control of measures
- 6) sample procedures
- 7) event procedures

6) Inquiry

7) Control of Table Sizes

8) Diagnostics, Debugging, and Optimization

9) Miscellaneous (Hints and Caveats and etc.)

0 Contrast with Textbook Specification

Chapter 3 of the textbook is an academic study of the issues involved in implementing any raster graphics package; it is *not* a description of the actual internal workings of any particular SRGP implementation. SRGP has been implemented on several major types of hardware platforms, and in all cases, low-level graphics utilities (QuickDraw on the Mac, X11 on workstations, and MetaGraphics on the PC) were used. Because the authors of SRGP were not involved in the implementation of the software actually drawing the primitives, there is no pixel-level compatibility between SRGP applications running on different platforms. Fortunately, the inconsistency in appearance will be typically noticeable only for extreme values for attributes (very thick primitives, rare combinations of write mode and pen/fill style, etc.).

The textbook spec did not discuss how SRGP would work in a windowing environment. This document describes how the windowing environment affects SRGP, and introduces a few new routines addressing window-specific problems.

The textbook's description of locator echo included "no echo" as an option available to the application. However, it was determined that an invisible cursor is frustrating to users of a multi-window system. The "no echo" option is thus honored only by the PC version of SRGP.

The Mac version currently does NOT support:

- pixmap patterns
- non-continuous line styles for ellipses and arcs
- refreshing of the screen-canvas window when it is uncovered after being partly or fully hidden

Because the design of Color QuickDraw represents a sharp turn away from the traditional "hands-on-the-hardware" approach of raster graphics, the Mac version simply cannot support a great deal of SRGP's functionality. In particular, the various non-trivial write modes (like XOR and OR) are supported but do not act precisely as documented. They will surprise with unpredictable results anytime you use them when: a) you've changed entries 0 and 1 of the color table from their default values of white and black, and/or b) your foreground/background colors are not currently 1/0. The most popular use of XOR'ing — providing a reversible method of highlighting — does work if you follow the above two rules AND you never change the last two entries (e.g., entries 255 and 254 on an 8-bit-deep machine) of the hardware color table. Also, be aware that write modes other than REPLACE will not make sense when drawing with pixmap patterns.

The textbook did not describe methods for loading bitmap and pixmap patterns into the respective pattern tables, for loading fonts into the font table, and for loading entries in the color table. See sections 3.1 and 3.3 of this document.

There are now "deluxe" versions of the measure records for each input device. These should be used when timestamps and modifier-key chords are needed.

The textbook specifies the size of the screen canvas cannot be changed by the application. That is no longer the case.

1 States of the System

SRGP must be enabled before use, and disabled after use.

```
void SRGP_begin (char *name, int width, int height, int planes,
                boolean enable_trace);
```

The window which will represent canvas #0 (the SRGP screen-canvas) is created; its initial size is determined by the values of the second and third parameters. The first parameter specifies a name for the application. The fifth parameter specifies the initial status of the tracing feature, which is described later in this section.

The fourth parameter is meaningful only on a display supporting color. It specifies how many planes of the color table should be reserved for SRGP's use; i.e., it places an upper bound on the number of colors that may be displayed simultaneously in the SRGP window. (The upper bound is 2^p colors, where p is the number of planes.) The fourth parameter is ignored when the program is run on a bilevel display.

If the program is being run on a color display, and you send the special value "0" as the fourth parameter, SRGP will take over the entire color table, giving your application color support as rich as the hardware can offer. (After initializing SRGP, you can inquire the "canvas depth" to determine how many planes are available.) The disadvantage: it will be impossible for the user to simultaneously see the SRGP window's proper coloring and the other clients' windows' proper coloring. Thus, you should request "0" planes only when your application truly needs full control of the color table.

If you request more planes than available, all available planes are allocated, just as if you requested "0" planes. Inquiry is thus your only way of determining exactly how many planes are available.

```
void SRGP_tracing (boolean);
```

When tracing is enabled, a message is sent to a logging file ('SRGPlogfile' in the current directory) each time the application calls an SRGP function; the message includes echoing of key parameters. **IMPORTANT:** Calls to some of the input functions are NOT traced; see section 5 for details. See section 7 for more information on execution with tracing.

The initial status of tracing is set when the application calls `SRGP_begin`, but it may be changed at any time via a call to `SRGP_tracing`.

```
void SRGP_allowResize (boolean);
```

By default, the screen-canvas window cannot be resized by the user. It is advisable that applications live with this restriction. The rare application that needs to allow resizing can use this routine. It is important to note that various artifacts occur when the user actually does take advantage of this freedom and perform a resize:

- 1) The window is cleared to color 0; any information that was on the screen just before the resize is lost.
- 2) The clip rectangle attribute is **not** changed automatically; the application must be responsible for changing it if necessary.

Obviously, an application that allows resizing must be informed whenever a resize occurs, to cope with the aforementioned problems and other application-specific ones. SRGP provides a callback utility, which allows an application to provide a function to be called whenever a resize has occurred:

```
typedef int (*funcptr)();
void SRGP_registerResizeCallback (funcptr);
```

The application-provided callback function referred to by the function-pointer parameter will be called whenever a resize has occurred. The callback function will receive two integers: the new width and the new height.

```
void SRGP_changeScreenCanvasSize (int newwidth, int newheight);
```

This routine allows the application to modify the size of the screen canvas. As a side-effect, the resize-callback function (if any) is called, just as if the user had requested the resize.

```
void SRGP_end (void);
```

The screen-canvas window is deleted, the Mac's color table is restored to normal, and the logging file is closed. Warning: ensure that your application calls this in all of its exit paths! See section 9 for related information on the adverse effects of failing to call this.

2 Canvases

SRGP procedures operate on *canvases*, a canvas being a 2D array of pixels (a virtual frame-buffer), whose depth is the number of planes requested by the application (via the fourth parameter to **SRGP_begin**).

Each canvas has its own local coordinate system. The origin (0,0) for the local coordinate system is the lower-left corner of the canvas, with the X-coordinate increasing to the right, and the Y-coordinate increasing towards the top. The coordinates passed to all primitive-generation procedures are in terms of the local coordinate system of the *currently-active canvas*.

At any given time, one canvas is active: it is the canvas being modified. Associated with each canvas is a group of *attributes* which affect all drawing into that canvas. Modification of these attributes is only possible when the corresponding canvas is currently active. When a canvas is created, its attribute group is initialized to standard default values.

Each canvas is identified by a unique integer *canvas index*. When SRGP is enabled, one canvas already exists and is active: the *screen canvas*, having index 0, whose height and width are determined from the parameters to **SRGP_begin**. The screen canvas is the only canvas which is *ever* visible. No more than **MAX_CANVAS_INDEX**+1 canvases (including the screen) may be extant simultaneously.

Canvases may be manipulated by the following procedures:

```
typedef int canvasID;
canvasID SRGP_createCanvas (int width, int height);
```

An invisible canvas of the specified dimensions is created and its unique index is returned. The new canvas' local-coordinate-system origin (0,0) forms the lower-left corner. (*width* - 1, *height* - 1) forms the upper-right corner. The pixels of a canvas initially store color index 0. Once a canvas is created, it can not be resized. (The screen canvas is an exception, but it can only be resized by the user, not by the application.) Upon return, the new canvas (and its corresponding attribute group) are active. If a new canvas cannot be created, 0 is returned.

```
void SRGP_deleteCanvas (canvasID);
```

No canvas may be deleted while it is active. Moreover, the screen canvas cannot be deleted with this routine.

```
void SRGP_useCanvas (canvasID);
```

The specified canvas becomes active. Primitives created subsequently are drawn in this canvas, and attributes set subsequently modify this canvas' attribute group.

3 Output

3.1 color

SRGP maintains a lookup table (LUT) that maps *color indices* (which are integers, used to index into the LUT) to actual colors. The number of entries available in the lookup table is based on the number of planes allocated for the application's use (via the fourth parameter to **SRGP_begin**). The number of planes available can be inquired via:

```
int SRGP_inquireCanvasDepth (void);
```

The legal color indices are numbers between (inclusive) 0 and $2^{canvasdepth} - 1$. The use of color indices beyond that range are clamped to $2^{canvasdepth} - 1$. (Negative values are illegal and will cause an error.) All implementations support two colors that may be referenced using names instead of numbers: **SRGP_WHITE** and **SRGP_BLACK**.

Macintosh users: See section 0 for a very important note on restrictions inherent in the Color Quickdraw version of SRGP!

On color nodes, **SRGP_WHITE** is 0 and **SRGP_BLACK** is 1, and they are the only initialized entries in the LUT. One should note, however, that if the first two entries of the LUT are changed by the application, the names (**SRGP_BLACK** and **SRGP_WHITE**) are no longer meaningful.

On monochrome displays, the two symbols `SRGP_BLACK` and `SRGP_WHITE` are implementation-dependent constants. Moreover, the use of a color index greater than 1 is clamped to 1 on a monochrome display.

An application may load a contiguous portion of the LUT by creating three arrays (one for red, one for blue, one for green) of *intensity values*, each value being an unsigned 16-bit integer. Intensity value 0 represents that primary's not contributing at all to the actual color, and $2^{16} - 1$ (65,535) represents that primary contributing its full glory to the actual color. Note that this method for specifying colors is machine-independent; workstations supporting only *C* bits per intensity value will ignore all but the *C* most significant bits of each intensity value.

To load `count` entries of the LUT, starting with entry `start`, create the three intensity value arrays and then call:

```
typedef unsigned short ush;
void SRGP_loadColorTable (int start, int count, ush *r, ush *g, ush *b);
```

An easy way to store “common” colors is provided by SRGP. Common colors are those colors which have been given names (like “Purple”, “MediumForestGreen”, and “Orange”) in the SRGP resource file. A complete list of the supported colors is provided below. SRGP supports only the setting of one LUT entry at a time when using common colors:

```
void SRGP_loadCommonColor (int entry, char *colorname)
```

The supported colors are shown in their lower-case compressed forms below. Be aware that you can insert spaces, and use upper-case arbitrarily, in the string you hand to `SRGP_loadCommonColor`.

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

Warning: The Mac's color table can sometimes be left in a disturbed state when an SRGP application exits prematurely. See section 9 of this document for more information.

3.2 geometric data types

The following SRGP data types allow storage of geometric entities:

```
typedef struct {
    int x, y;
} point;

typedef struct {
    point bottom_left, top_right;
} rectangle;
```

Instances of these data types may be created using these routines:

```
point SRGP_defPoint (int x, int y);
rectangle SRGP_defRectangle (int left_x, int bottom_y, int right_x, int top_y);
```

3.3 control of the pattern and font tables

Several of the SRGP attributes are patterns to be used for filling areas and for drawing lines and frames. Two pattern tables are supported: one storing bitmaps and one storing pixmaps.

The bitmap pattern table is initialized in this way: Pattern 0 is all background, and pattern 1 is all foreground. Patterns 1 through 38 are the standard Macintosh patterns shown in Volume 1 of Chernicoff's *Macintosh Revealed*, and page I-474 of *Inside Macintosh*. Patterns 40 through 104 are greyscale patterns increasing gradually in intensity from all background (40) to all foreground (104). All other entries in the bitmap pattern table have random patterns initially. To see an array of tiles showing the default bitmap pattern table, run the example program **show_patterns**.

Only entry 0 in the pixmap pattern table is defined, and it is simply an all-color-0 pattern.

SRGP provides two methods for changing entries in the pattern table:

- You can have SRGP load one or more patterns from a file which stores ASCII-text pattern specifications you can create using a text editor or convenient bitmap-editor programs (if available).
- You can give SRGP a pattern specification in the form of an array of numbers.

To load a bitmap pattern from a file, open the file and pass the stream to the function:

```
int SRGP_loadBitmapPatternsFromFile (FILE *stream);
```

The input may be composed of one or more pattern specifications ("specs"). Each spec must occupy exactly two lines and must match this format:

```
static char bitpat_Xy[] = {
    0x??, 0x??, 0x??, 0x??, 0x??, 0x??, 0x??, 0x??};
```

where **X** is a non-negative integer specifying the index of the entry to be set, **y** is any arbitrary garbage between the integer and the left square-brace, and **0x??** is any arbitrary byte value represented in hexadecimal.

As a convenience, any lines beginning with '#' are ignored, but these comment lines must not interrupt a single two-line spec sequence. Blank unused lines are not allowed anywhere in the file.

The closing of the input stream must be performed by the caller. This function returns 1 if any problems at all occurred; since its parser makes no attempt at error recovery, it is wise to check the return value.

To load a pixmap pattern from a file, use this routine whose functionality is similar to that of the one for bitmap patterns:

```
int SRGP_loadPixmapPatternsFromFile (FILE *stream);
```

Warning: the current Macintosh version does not support this. The input may be composed of one or more pattern specifications, each matching this format:

```
static int pixpat_Xy[] = {
    ?, ?, ?, ?, ?, ?, ?, ?,
    ?, ?, ?, ?, ?, ?, ?, ?,
    ?, ?, ?, ?, ?, ?, ?, ?,
    ?, ?, ?, ?, ?, ?, ?, ?,
    ?, ?, ?, ?, ?, ?, ?, ?,
    ?, ?, ?, ?, ?, ?, ?, ?,
    ?, ?, ?, ?, ?, ?, ?, ?,
    ?, ?, ?, ?, ?, ?, ?, ?};
```

In this specification, each ? represents a decimal integer color index.

Applications generating the patterns at runtime can use these routines to set one entry at a time:

```
void SRGP_loadBitmapPattern (int pattern_id, char *data);
void SRGP_loadPixmapPattern (int pattern_id, int *data);
```

The former routine expects an array of 8 characters; the latter an array of 64 integers. **Warning: the current Macintosh version does not support the loading of pixmap patterns.**

Another attribute is the font to be used for text. SRGP provides a table of fonts that may be used and modified by the application. Each entry is identified by a unique *font index* (ranging from 0 to **MAX_FONT_INDEX**). Initially, only entry #0 of the font table is defined. The application can modify the table via:

```
void SRGP_loadFont (int fontindex, char *name);
```

This function allows the application to load a font into a given entry of the table. The name of the font must comply with the format ‘font.size.style’ where *font* is the name of a Macintosh font family (e.g., “Chicago”), *size* is any positive number representing points, and *style* is a collection of single-letter style codes: **b** for bold, **i** for italic, **u** for underline, **o** for outline, **s** for shadow, **c** for condense, or **e** for extend. If plain text is desired, you may use the format ‘font.size’ with or without a terminating period.

3.4 control of attributes affecting output

These procedures allow control of the value of each attribute associated with the currently-active canvas.

```
typedef enum {WRITE_REPLACE, WRITE_XOR, WRITE_OR, WRITE_AND} writeModeType;
void SRGP_setWriteMode (writeModeType);
```

The write mode affects the writing of a pixel, the generation of a primitive, and the copying of a rectangular portion of a canvas. The default is **WRITE_REPLACE**. Please see the important note on Mac’s limited support of write modes, in section 0 of this manual.

```
void SRGP_setClipRectangle (rectangle);
```

All subsequently-created primitives and subsequent pixel-copyings are clipped to the specified rectangle. The default clipping rectangle is exactly the size of the associated canvas. It is illegal to set the clipping rectangle to a rectangle which does not lie completely within the boundaries of its associated canvas.

```
void SRGP_setFont (int font_index);
```

The application chooses from a font by giving the index (ranging from 0 to **MAX_FONT_INDEX**) into the font table. The default is 0, the only entry in the font table which is initialized when SRGP is launched.

```
void SRGP_setMarkerSize (int width_in_pixels);
```

This describes the dimensions of the imaginary square that circumscribes a marker’s image.

```
typedef enum {MARKER_CIRCLE, MARKER_SQUARE, MARKER_X} markerStyleType;
void SRGP_setMarkerStyle (markerStyleType);
```

SRGP supports three different marker shapes, circle being the default.

```
typedef enum {CONTINUOUS, DASHED, DOTTED, DOT_DASHED} lineStyleType;
void SRGP_setLineStyle (lineStyleType);
void SRGP_setLineWidth (int width_in_pixels);
```

The default line style is continuous, default line width is 1. On the Macintosh, the true width of a line may be greater than the width you specify, depending upon the line's slope. The true width will match the specified width only for horizontal and vertical lines. **Warning: in the Macintosh version, ellipses and arcs are always drawn using the “continuous” line style.**

```
void SRGP_setColor (int color_index);
```

This sets the foreground (drawing) color to the color that is stored at the given entry in the LUT; the default is 1.

```
void SRGP_setBackgroundColor (int color_index);
```

Default is 0. The background color is used to color the pixels denoted by 0 values in opaque bitmap pattern fills.

```
void SRGP_setPlaneMask (int bitmask);
```

Currently not implemented on Mac version!

```
typedef enum {
    SOLID, BITMAP_PATTERN_OPAQUE, BITMAP_PATTERN_TRANSPARENT, PIXMAP_PATTERN}
drawStyle;
void SRGP_setFillStyle (drawStyle);
void SRGP_setPenStyle (drawStyle);
```

Fill style affects filled primitives; pen style affects outlined (framed) primitives or lines. Text is not affected by either of these attributes. Default is **SOLID**. **Pixmap patterns are currently not implemented in Mac version.**

```
void SRGP_setFillBitmapPattern (int pattern_index);
void SRGP_setFillPixmapPattern (int pattern_index);
void SRGP_setPenBitmapPattern (int pattern_index);
void SRGP_setPenPixmapPattern (int pattern_index);
```

Denotes the entry in the appropriate pattern table which is to be used when the fill or pen style is not **SOLID**. **Pixmap patterns are currently not implemented on the Mac.**

The entire set of attributes may be set to a previously-stored group of values using the function described next. Later in this reference is described the function (**SRGP_inquireAttributes**) that allows inquiry of the current attribute group.

```
typedef struct ... attribute_group; /* see srgppublic.h for details */
void SRGP_setAttributes (attribute_group*);
```

The parameter's value should have been obtained from a previous call to **SRGP_inquireAttributes**.

3.5 generation of primitives

The functions described in this section perform drawing in the currently active canvas. For each primitive generator described, the list of attributes affecting its operation is presented.

An ellipse is specified in terms of the rectangle within which it is inscribed. Polygons, rectangles, and ellipses may be generated as *framed* or *filled*. A filled primitive is all-interior — the frame is not displayed.

```
void SRGP_point (point);
void SRGP_pointCoord (int x, int y);
```

Current write mode, foreground color, and plane mask apply.

```
void SRGP_marker (point)
void SRGP_markerCoord (int x, int y)
```

Current marker style, marker size, write mode, foreground color, and plane mask apply.


```
void SRGP_line (point pt1, point pt2);
void SRGP_lineCoord (int x1, int y1, int x2, int y2);

void SRGP_rectangle (rectangle);
void SRGP_rectanglePt (point lower_left, point upper_right);
void SRGP_rectangleCoord (int left_x, int lower_y, int right_x, int upper_y);
```

Current write mode, plane mask, colors, line width, line style, and pen style apply.

```
void SRGP_polyPoint (int vert_count, point *vertices);
void SRGP_polyMarker (int vert_count, point *vertices);
void SRGP_polyLine (int vert_count, point *vertices);
void SRGP_polygon (int vert_count, point *vertices);

void SRGP_polyPointCoord (int vert_count, int *x_coords, int *y_coords);
void SRGP_polyMarkerCoord (int vert_count, int *x_coords, int *y_coords);
void SRGP_polyLineCoord (int vert_count, int *x_coords, int *y_coords);
void SRGP_polygonCoord (int vert_count, int *x_coords, int *y_coords);
```

Current write mode, plane mask, colors, line width, line style, and pen style apply. `SRGP_polygon(Coord)` automatically connects the first and last vertices to form a closed polygon. Lists of vertices and coordinates are limited in size to `MAX_POINTLIST_SIZE`.

```
void SRGP_ellipse (rectangle bounds);
void SRGP_ellipseArc (rectangle bounds, double startangle, double endangle);
```

Current write mode, plane mask, colors, line width, line style, and pen style apply. An arc extends counterclockwise from the start angle to the end angle. Angles are in rectangular degrees and must lie between 0 and 360, with 0 degrees being a horizontal ray extending towards positive infinity.

```
void SRGP_fillPolygon (int vert_count, point *vertices);
void SRGP_fillPolygonCoord (int vert_count, int *x_coords, int *y_coords);
void SRGP_fillEllipse (rectangle);
void SRGP_fillEllipseArc (rectangle bounds, double startangle, double endangle);
void SRGP_fillRectangle (rectangle);
void SRGP_fillRectanglePt (point lower_left, point upper_right);
void SRGP_fillRectangleCoord (int left_x, int lower_y, int right_x, int upper_y);
```

Current write mode, plane mask, colors, and fill style apply.

```
void SRGP_text (point origin, char *str);
```

Current write mode, plane mask, foreground color, and font apply. The origin marks the leftmost point to be affected by the text, and marks the horizontal baseline for the text, under which only the text's characters' descenders will appear.

3.6 audio output

```
void SRGP_beep (void);
```

4 The copyPixel Procedure

This procedure allows a portion of a canvas to be copied into another part of itself or into another canvas. See the textbook for more information on this powerful feature.

```
void SRGP_copyPixel (canvasID source_canvas, rectangle source_rect,
                    point dest_corner);
```

The copying operation is composed of two parts. First, a *copy* of a rectangular portion of a canvas is created. Then, the copy is *placed* somewhere within the currently-active canvas. (The currently-active canvas may or may not also be the canvas providing the source of the copy.)

dest_corner describes the lower-left corner of the destination rectangle (lying inside the currently-active canvas) having the same size as **source_rect**.

Only the rectangular portion of **source_rect** which lies within the boundaries of the source canvas is copied. The placement operation is affected by the current clipping-rectangle and write-mode.

5 Input

An application program obtains input from an operator by controlling a set of *logical input devices*, each representing a unique input technique. Each device may be placed in a number of different *input modes*, each representing a unique type of interaction with the input device.

5.1 properties of input devices

Each input device is described in terms of these information: a *measure*, a *trigger set*, and a set of *attributes*.

The *measure* of an input device is the value currently associated with the device.

The *trigger* of an an input device is the action which indicates a significant moment associated with the device.

The *attributes* of an input device are the parameters of the device which are under application-control, primarily the echo characteristics.

At any given time, each device is either *active* or *inactive*. The process of *activation* places a device into an active state; the process of *deactivation* places it into an inactive state. Zero or more devices may be simultaneously active.

5.2 input modes

There are three modes in which input devices operate. (Initially, each device is inactive.) The modes' names are listed below, accompanied by a description:

INACTIVE	When device α is inactive, no events are posted concerning it, and its measure is not available to the application.
SAMPLE	When device α is in Sample mode, it is active. The application may call 'SRGP_sample α ' to immediately obtain the measure of input device α . The firings of α -triggers do not have any effects.
EVENT	When device α is in Event mode, it is active. The firing of an α -trigger causes an <i>input report</i> (containing the measure of the device at the time of the firing) to be appended to the <i>input queue</i> .

The following function allows control of the input modes and echoing for all input devices:

```
typedef enum {NO_DEVICE, LOCATOR, KEYBOARD} inputDevice;
typedef enum {INACTIVE, SAMPLE, EVENT} inputMode;
void SRGP_setInputMode (inputDevice, inputMode);
```

The specified input device is placed in the specified mode. Whenever device α 's mode is changed from Inactive to either Sample or Event, the device is *activated*: its measure is initialized (to a static default initial value, or to a value specified by the application while the device was inactive) and echoing begins. When α 's mode is set to Event, queueing of the device's event reports is enabled as well. When α 's mode is changed from Event, all queued events for that device are discarded.

When α 's mode is set to Inactive, the device is *deactivated*: trigger firings from the device are ignored and echoing is disabled.

5.3 input devices

The SRGP input devices are described in this section.

LOCATOR The measure of the Locator device incorporates a position expressed in the coordinate system of the screen canvas, a chord giving the status of the mouse buttons, and the number of the button which most recently experienced a transition. The button-chord array is indexed using three constants: **LEFT.BUTTON**, **MIDDLE.BUTTON**, and **RIGHT.BUTTON**. The Macintosh version treats the sole mouse button as the left button, and pretends the middle and right are always “up”.

The *button-mask* attribute of this device determines which of the buttons are of interest when the device is active in Event mode: only buttons specified in this mask can trigger an event.

```
typedef enum {UP, DOWN} buttonStatus;
typedef struct {
    point position;
    buttonStatus button_chord[3];
    int button_of_last_transition;
} locator_measure;
```

The “deluxe” version of the locator measure includes a chord giving the status of three primary modifier keys at the time of the last button transition, and a timestamp structure. The modifier chord array is indexed via **SHIFT**, **CONTROL**, and **META**. (The META key is labelled “option” on the Macintosh keyboards.) The timestamp specifies the time at which the most recent *change* to the measure occurred — i.e., successive sampling of a non-moving locator produces a “constant” timestamp.

```
typedef struct {
    int seconds;
    int ticks;
} srgp_timestamp;

typedef struct {
    point position;
    buttonStatus button_chord[3];
    int button_of_last_transition;
    buttonStatus modifier_chord[3];
    srgp_timestamp timestamp;
} deluxe_locator_measure;
```

*duration since application launch
a tick is 1/60th second*

status at last transition

KEYBOARD The measure of this device is a character string, storing either a single ASCII character code or a sequence of printable characters.

The “deluxe” version of the measure includes the modifier chord, a timestamp, and a locator position:

```
typedef struct {
    char *buffer;
    int buffer_length;
    buttonStatus modifier_chord[3];
    point position;
    srgp_timestamp timestamp;
} deluxe_keyboard_measure;
```

*ptr to space allocated by application
set by application*

The *processing-mode* attribute of this device determines which of the two meanings is given to the measure of the device:

- RAW:** When a key is hit, the measure stores a string of length 1 whose single element is the ASCII character code of the key hit (taking into account the status of the `shift` and `control` modifier keys) and a trigger-firing occurs. The modifier chord shows the status of the modifiers when the key was hit. No echo occurs in RAW mode.
- EDIT:** (the default) When a key representing a printable character is hit and the string is not yet full, the character is appended to the string. When the `backspace` key is hit, the last character of the string is deleted. When `return` is hit, an event is sent (representing the full value of the string) and the measure is set to the null string. In EDIT mode, the modifier chord is not maintained and should be ignored.

C programmers should take care to allocate a buffer large enough to include the null character that terminates the string measure. For example, two bytes are needed to store a RAW-mode measure.

5.4 control of attributes

The following procedures set the attributes for input devices. Attributes may be set at any time, regardless of whether the device is active or inactive.

```
void SRGP_setLocatorButtonMask (int value);
```

The value should be 0 or an OR combination of one or more of the following defined constants: `LEFT_BUTTON_MASK`, `MIDDLE_BUTTON_MASK`, and `RIGHT_BUTTON_MASK`. Initially the value is `LEFT_BUTTON_MASK`: meaning only the left button (which is the only button on a 1-button mouse) generates events.

```
SRGP_setLocatorEchoType (int value); /* CURSOR, RUBBER_LINE, or RUBBER_RECT */
```

An application can choose to have just the cursor, or to also have a rubber-primitive (anchored at a fixed point with the other end of the primitive following the cursor's movement). Note: the value `NO_ECHO` is also accepted, but it is ignored on all platforms except the IBM PC.

SRGP provides a cursor table whose 0th entry is initialized to an arrow; all other entries are unusable until loaded.

```
void SRGP_loadCursorTable (int cursor_index, int shape);
```

Legal cursor indices are numbers between 0 and `MAX_CURSOR_INDEX`, inclusive. The shape may be one of the following: 1 for the standard Mac "I-beam" text cursor, 2 for a cross, 3 for a "plus" cursor, and 4 for a watch icon. The shape may also be the resource ID of a customized cursor placed as a "CURS" resource in the application's resource file.

The attributes for the locator's echo are set via:

```
void SRGP_setLocatorEchoCursorShape (int cursor_index);
void SRGP_setLocatorEchoRubberAnchor (point position);
```

The keyboard's attributes are set via:

```
typedef enum {EDIT, RAW} keyboardMode;
void SRGP_setKeyboardProcessingMode (keyboardMode);
void SRGP_setKeyboardEchoColor (int color_index);
void SRGP_setKeyboardEchoFont (int font_index);
void SRGP_setKeyboardEchoOrigin (point position);
```

Keyboard echo attributes are only meaningful when the keyboard is active in EDIT processing mode. Setting the keyboard's processing mode (default EDIT) clears the keyboard's measure as a side-effect.

5.5 control of measures

The measure of a device may be changed by the application at any time. If the change is performed while the device is active, the measure immediately changes, as does as echoing concerning the device. If it is done while the device is inactive, the specified measure is used to initialize the device's measure the next time it is activated. NOTE: the button-related fields of the locator measure may not be changed by the application.

```
void SRGP_setLocatorMeasure (point value);
void SRGP_setKeyboardMeasure (char *value);
```

5.6 sample procedures

The involved input device must be in SAMPLE mode. Each function places in the provided place the current measure of the corresponding device. Calls to these routines are NOT traced.

The `SRGP_sampleKeyboard` function copies the keyboard measure into the given character-array buffer of size `buffer_length`. If the current keyboard measure is longer than $(buffer_length-1)$ bytes, it is truncated. Similarly, the keyboard-measure structure sent to `SRGP_sampleDeluxeKeyboard` must have pre-set values for its `buffer` and `buffer_length` fields.

```
void SRGP_sampleLocator (locator_measure *measure);
void SRGP_sampleKeyboard (char *buffer, int buffer_length);
void SRGP_sampleDeluxeLocator (deluxe_locator_measure *measure);
void SRGP_sampleDeluxeKeyboard (deluxe_keyboard_measure *measure);
```

5.7 event procedures

Calls to these functions are NOT traced.

```
inputDevice SRGP_waitEvent (int maximum_wait_time);
```

If, upon entry, the event queue is not empty, the procedure exits immediately, identifying the event report at the head of the queue and removing the report from the queue. Otherwise, the application enters a wait state, which is exited upon the first occurrence of a trigger-firing from any device which is currently in EVENT mode. The wait state never lasts for more than the number of ticks (1/60 seconds) given in the `maximum_wait_time` parameter (which, when negative, represents infinity). An application can "poll" the queue (avoiding a wait state) by specifying "0" as the maximum wait time.

The return value identifies the device causing the event. The special value `NO_DEVICE` is returned when the procedure exits due to timeout.

When an application discovers that an input event (not a timeout) caused the return of `SRGP_waitEvent`, it may obtain the data associated with the involved event by using the appropriate "get" function, whose parameters and return values mimic those of the sample functions:

```
void SRGP_getLocator (locator_measure *measure);
void SRGP_getKeyboard (char *measure, int buffer_length);
void SRGP_getDeluxeLocator (deluxe_locator_measure *measure);
void SRGP_getDeluxeKeyboard (deluxe_keyboard_measure *measure);
```

6 Inquiry

NOTE: Calls to these routines are NOT traced.

```
void SRGP_inquireAttributes (attribute_group *group);
```

The current states of all attributes are copied into the provided group structure. For information on the names of the fields in the structure, see 'srgppublic.h'.

```
canvasID SRGP_inquireActiveCanvas (void);
```

This function allows inquiry of the ID of the currently-active canvas.

```
rectangle SRGP_inquireCanvasExtent (canvasID);
```

```
void SRGP_inquireCanvasSize (canvasID, int *width, int *height);
```

Two functions allowing inquiry of the size of a canvas.

```
int SRGP_inquireCanvasDepth (void);
```

Returns the number of planes available in all canvases.

```
void SRGP_inquireTextExtent (char *str, int *width, int *ascent, int *descent);
```

This procedure allows inquiry of the rectangular extent which would be covered by the output of the given character string with the current font attribute.

7 Control of Table Sizes

The tables that store patterns, fonts, etc. have default sizes that in many cases are acceptable. You may, however, choose to reduce the size of a table to save memory (if you're working on a Mac Plus, for instance) or increase the size of a table if you need more entries. You may change the size of a table *only before* SRGP is initialized!

```
void SRGP_setMaxCanvasIndex (int i);
```

```
void SRGP_setMaxPatternIndex (int i);
```

```
void SRGP_setMaxCursorIndex (int i);
```

```
void SRGP_setMaxFontIndex (int i);
```

```
void SRGP_setMaxPointlistSize (int i);
```

See 'srgp_sphigs.h' for the defaults for these sizes. NOTE: Do not reduce the size of the pattern table!

8 Diagnostics, Debugging, and Optimization

SRGP offers two features that aid the developer in debugging an application. Both of these features may be disabled or enabled by the programmer and even by the user at run-time, in order to optimize the execution.

The first feature is tracing. When enabled, each call to an SRGP routine (except a few input-related routines) produces a detailed message in a log file. A parameter to **SRGP_begin()** controls the initial state of tracing; calls to **SRGP_tracing** can be used to control the state during runtime. Early test runs of an application should always be performed with tracing enabled. For more details on tracing, see Section 1.

The second feature is parameter verification. All SRGP routines perform verification of all parameters (except those that are pointers to an array or structure) before they commence operation. All errors are fatal and produce a crash with a detailed error message. Only when a program is fully debugged should optimization efforts include disabling parameter verification! You may permanently disable verification and tracing via:

```
void SRGP_disableDebugAids (void);
```

About SRGP's diagnostics: There are two types of run-time errors. The first type is parameter-verification errors; the verification can be turned off as mentioned earlier. The second type occurs when a problem unrelated to bad input occurs, like running out of memory when attempting to allocate a canvas. All errors – of both types – are considered fatal by default, and cause a crash after displaying an informative message to the user. Some programmers might wish to make all errors be non-fatal, so program execution can continue (with suitable recovery algorithms, of course). The following routine can be used to choose between fatal and non-fatal error handling:

```
typedef enum {FATAL_ERRORS, NON_FATAL_ERRORS} errorHandlingMode;
void SRGP_setErrorHandlingMode (errorHandlingMode);
```

When an error is detected by SRGP while the mode is `NON_FATAL_ERRORS`, no message is issued to the user; rather, a global variable is set to a positive integer that represents the error:

```
#include "srgp_errtypes.h"
extern int SRGP_errorOccurred;
```

The header file contains symbolic constants mapping the integers to error types. The global variable is never reset to 0 by SRGP; the application is responsible for examining and resetting it. Obviously, non-fatal mode should be used with great care, and only late in an application's development.

9 Miscellaneous (Hints, Caveats, etc.)

An SRGP application cannot control multiple windows: only canvas #0 is represented by a visible window.

When using this Macintosh version, you should keep the screen-canvas window uncovered and fully within the screen bounds at all times, because SRGP cannot perform “damage repair” in its Macintosh incarnation.

A great amount of memory is needed to support off-screen canvases when using Color QuickDraw. Even if the application requests the use of only a few of the available planes, the depth of the allocated canvas will match the depth of the display hardware. If you run out of memory when you know your machine is capable of a lot more, you may need to use the “Set Project Type...” dialog (under the “Project” menu in THINK C) to increase the partition size for your project.

If an application is terminated before it calls ‘`SRGP_end`’, you are returned to the Finder or to THINK C with a damaged color table. You can easily fix the color table by using the “Control Panel” “Monitors” desk accessory. Simply select a different item (e.g., 16) in the “characteristics” list, and then re-select the original item (e.g., 256).

— FIN —