Screen Graphics Subsystem Developer's Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Monday, October 6, 2014

Table of Contents

About Screen

Screen Graphics Subsystem is a compositing windowing system that can composite graphics from several different rendering technologies.

Screen allows developers to create specific vertical applications using industry-standard tools in a UI development environment. UI technologies that Screen can combine include HTML5, Elektrobit GUIDE, Crank Storyboard, Qt, and native (e.g., OpenGL ES) code.

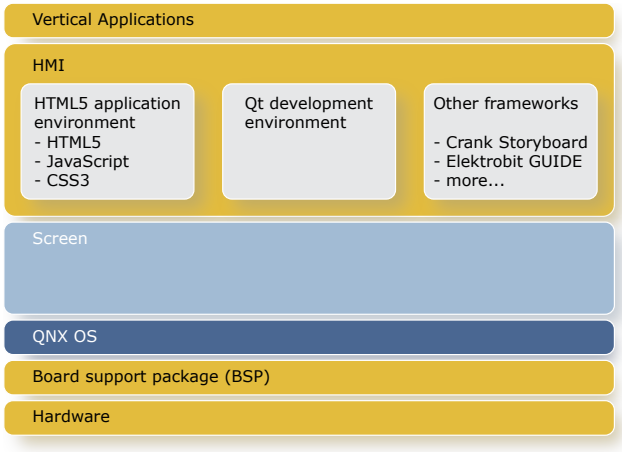


Figure 1: Screen

Screen enables developers to create separate windows for the output of each rendering technology (e.g., HTML5, Qt, Video, or OpenGL ES) so that each window can be transformed (e.g., scaling, translation, rotation, alpha blending, etc.) to build the final scene for display.

The *Screen Graphics Subsystem Developer's Guide* is intended for application developers. This table may help you find what you need in this guide:

To find out about:	See:
Overview of Screen	*Overview of Screen* (p. 17)
Composition	*Understanding composition* (p. 21)
Screen API	*Screen API* (p. 27)
Contexts	*Contexts* (p. 29)
Windows	*Windows* (p. 33)
Displays	*Displays* (p. 47)
Event Types	*Event Types* (p. 55)
Configuring Screen	*Configuring Screen* (p. 141)
Screen Tutorials	*Screen Tutorials* (p. 57)

To find out about:	See:
Screen Library Reference	*Screen Library Reference* (p. 179)

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	`NULL`
Data types	`unsigned short`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** **Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
About Screen

Screen Graphics Subsystem is a compositing windowing system that can composite graphics from several different rendering technologies.

Screen allows developers to create specific vertical applications using industry-standard tools in a UI development environment. UI technologies that Screen can combine include HTML5, Elektrobit GUIDE, Crank Storyboard, Qt, and native (e.g., OpenGL ES) code.

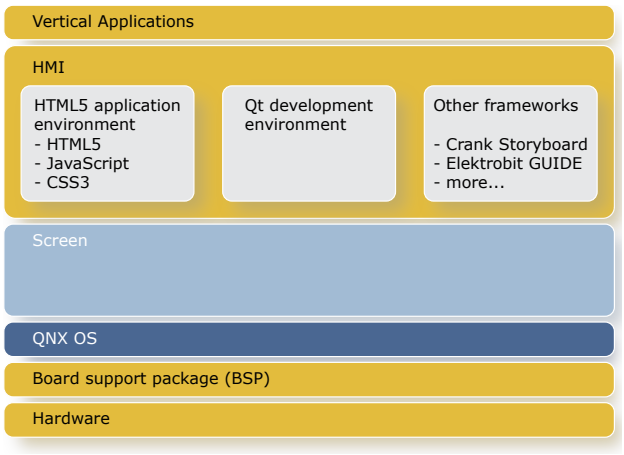


Figure 2: Screen

Screen enables developers to create separate windows for the output of each rendering technology (e.g., HTML5, Qt, Video, or OpenGL ES) so that each window can be transformed (e.g., scaling, translation, rotation, alpha blending, etc.) to build the final scene for display.

The *Screen Graphics Subsystem Developer's Guide* is intended for application developers. This table may help you find what you need in this guide:

To find out about:	See:
Overview of Screen	*Overview of Screen* (p. 17)
Composition	*Understanding composition* (p. 21)
Screen API	*Screen API* (p. 27)
Contexts	*Contexts* (p. 29)
Windows	*Windows* (p. 33)
Displays	*Displays* (p. 47)

To find out about:	See:
Event Types	*Event Types* (p. 55)
Configuring Screen	*Configuring Screen* (p. 141)
Screen Tutorials	*Screen Tutorials* (p. 57)
Screen Library Reference	*Screen Library Reference* (p. 179)

Chapter 2
Overview of Screen

Screen is a compositing windowing system.

Unlike traditional windowing systems that arbitrate access to a single buffer associated with a display, this compositing windowing system provides the means for applications to render off-screen.

Rendering to off-screen buffers allows the manipulation of window contents without having to involve the applications that are doing the rendering. Windows can be moved around, zoomed in, zoomed out, rotated, or have transparency effects applied to them, all without requiring the application to redraw or even be aware that such effects are taking place.

Screen is responsible for:

- running all drivers (e.g., input, display, OpenGL ES)
- allocating memory needed by application windows
- displaying content when rendering completes

Screen integrates multiple graphics and user interface (UI) technologies into a single scene. This scene is rendered into one image that is associated with a display.

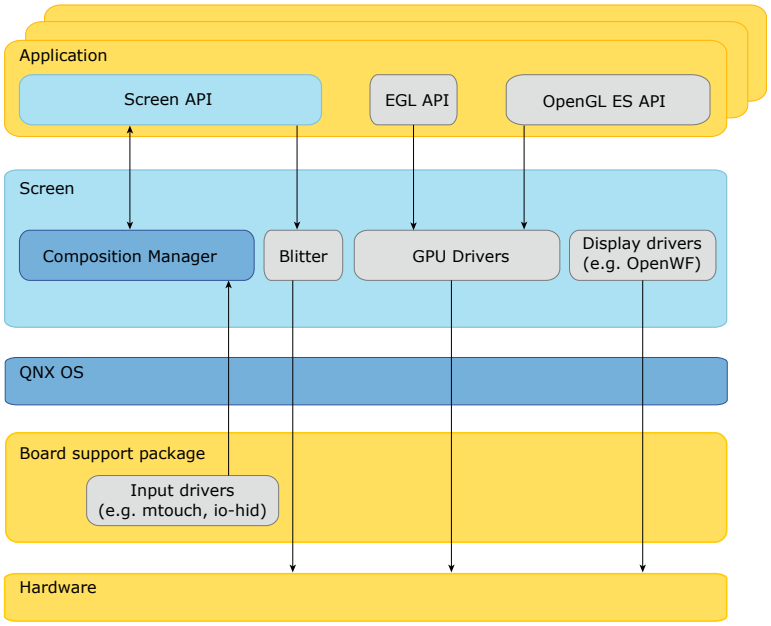


Figure 3: Screen: A composited windowing system.

Handling composition

The main responsibility of Screen is to combine all visible window buffers into one final image that is displayed. This responsibility is handled by the Composition Manager and is achieved using several classes of hardware. The Composition Manager can be configured to use available compositing hardware in a way that best meets the needs of a particular system.

Screen has a plug-in architecture that includes hardware-specific compositing modules and a module for OpenGL for Embedded Systems (OpenGL ES).

Screen uses GPU-accelerated operations to optimally build the final scene. You may resort to using software rendering if your hardware cannot satisfy the requests. The graphics drivers and display controllers that run within Screen are based on the OpenWF Display (WFD) API.

Communicating with applications

Applications communicate with the Composition Manager using Screen API to perform such tasks as the following:

- creating and destroying *windows*.
- creating and destroying *pixmaps*.
- using accessor functions to set and get native window, pixmap, display, device, and buffer properties.
- drawing into native buffers that are associated with windows and pixmaps
- making areas, within buffers, that can be displayed.
- Receiving and processing asynchronous events from Screen.
- sending events to other windowed applications.

Applications can render using:

- software; applications access the window buffer and write to it using the CPU.
- OpenGL ES; use EGL to target the window buffer(s) with OpenGL ES calls.

Sample Screen applications

Screen provides a set of sample native applications that demonstrate what you can do with the Screen API. These sample applications are installed on your host under each target-specific directory under ***$QNX_TARGET***.

Running these applications will help in determining whether or not you have set the appropriate Screen configuration.

`/usr/bin/calib-touch`

This utility loads or creates an `mtouch` calibration file, `/etc/system/config/calib.<hostname>`. If the calibration file needs to be created, the user will be prompted to calibrate.

/usr/bin/display_image

This application displays a specified image to the specified display.

/usr/bin/egl-configs

This utility queries and displays the number of all the frame buffer configurations for the specified display. If no display is specified, then the default display is used.

/usr/bin/events

This application uses a window manager connection to Screen. It prints all events to the console output as they are received.

/usr/bin/font-freetype

This application shows how to render text with FreeType and OpenGL ES 1.X.

/usr/bin/gles1-gears

This application shows windowed gears that use OpenGL ES 1.X for the rendering API.

/usr/bin/gles2-gears

This application shows windowed gears that use OpenGL ES 2.X for the rendering API.

/usr/bin/gles2-maze

This application uses OpenGL ES 2.X for the rendering; it demonstrates how to use texture as well as vertex and fragment shaders.

/usr/bin/print-gestures

This application detects and displays recognized gestures.

/usr/bin/screenshot

This application takes a screenshot of a specified size of the display and saves the output file in BMP format.

/usr/bin/sw-vsync

This application shows windowed vsync that uses software rendering.

/usr/bin/vcapture-test

This application demonstrates that you can connect to a device for video input, capture frames from that input source, and then display the captured frames using Screen.

/usr/bin/vkey

This application uses the privileged context, `SCREEN_INPUT_PROVIDER_CONTEXT`, to create a connection to the Composition Manager. A key sequence is injected to whichever window has input focus on the specified display. The application will exit as soon as the last character is sent.

/usr/bin/yuv-test

This application displays a YUV test pattern.

Chapter 3
Understanding composition

Composition is the process of combining multiple content sources together into a single image.

Screen, as much as possible, uses hardware layering (pipelines) for composition. When multiple pipelines and buffers are supported by the device driver, Screen takes advantage of these hardware capabilities to use each pipeline and to combine the pipelines at display time. For applications that require complex graphical operations, you can also use hardware-accelerated options such as OpenGL ES and/or bit-blitting hardware. Only when your platform does not support any hardware-accelerated options, will Screen then resort to using the CPU to perform composition.

The following forms of transparency are used for composition:

Destination view port

Allows any content on layers below to be displayed. This transparency mode has an implicit transparency in that anything outside the specified view port is transparent.

Source chroma

Allows source pixels of a particular color to be interpreted as transparent. Unlike a destination view port, source chroma allows for transparent pixels within the buffer.

Source alpha blending

Allows pixel blending based on the alpha channel of the source pixel. Source alpha blending is one of the most powerful forms of transparency because it can blend in the range from fully opaque to fully transparent.

There are two types of composition:

Hardware composition

Composes all visible (enabled) pipelines of the display controller and then displays them.

Screen composition

Composes mutliple elements that are combined into a single buffer that is associated to a pipeline and displayed. The composition is handled by the Composition Manager of Screen.

Hardware composition

Hardware composition capabilities are constrained by the display controller. Therefore, they vary from platform to platform.

All visible (enabled) pipelines are composed and displayed. Each layer, at the time it is displaying, has only one buffer associated to it.

The buffer belongs to a window that can be displayed directly on a pipeline. This window is considered autonomous because no composition was performed on the buffer by the Composition Manager. For a window to be displayed autonomously on a pipeline, this window buffer's format must be supported by its associated pipeline.

Figure 4: An example of hardware composition with two windows and two supported pipelines

This hardware composition example shows two windows. Each window posts a different buffer and binds to a different pipeline. The output from both pipelines is combined and fed into the associated display port and then onto the display hardware.

Access to hardware composition capabilites is system dependent.

In order to use hardware composition, you must:

- have the correct Screen configuration. Determine the pipelines that are on your display controller and choose the pipeline on which you want to display a window. The supported pipelines on your `wfd` device are configured in your `graphics.conf` file.
- use *screen_set_window_property_iv()* to set `SCREEN_PROPERTY_PIPELINE` to one of the supported pipelines you have configured in `graphics.conf`.

- use *screen_set_window_property_iv()* to set the SCREEN_USAGE_OVERLAY bit of your SCREEN_PROPERTY_USAGE window property.

Screen composition

Many of the composition capabilities that are used in hardware composition can be achieved in Screen composition by the Composition Manager.

When your platform doesn't have hardware capabilities to support a sufficient number of pipelines to compose a number of required elements, or to support a particular behavior, composition can still be achieved by the Composition Manager, an internal component of Screen.

The Composition Manager combines multiple window buffers into one resultant buffer. This is the composite buffer (Screen framebuffer).

Figure 5: An example of Screen composition with two windows and only one supported pipeline

This Screen composition example shows two windows; each window posts a different image. Those images are composed into one composite framebuffer, which binds to the single pipeline. This output is then fed into the associated display port and then onto the display hardware.

For Screen composition, don't set the SCREEN_PROPERTY_PIPELINE window property or the SCREEN_USAGE_OVERLAY bit in your SCREEN_PROPERTY_USAGE window property.

You can also maximize the advantages of both hardware and Screen composition capabilities. What this means is that you can combine multiple windows into one composite buffer, bind this buffer to a pipeline, and still take advantage of hardware capabilities to combine output from multiple pipelines.

Figure 6: An example of both Screen and hardware composition with three windows, one composite buffer, and two supported pipelines

This composition example shows three windows. The first window posts and binds to one specific pipeline. The second and third windows post to a framebuffer where the buffers from these windows are combined. The framebuffer binds to the second pipeline. The output from both pipelines is combined and fed into the associated display port and then onto the display hardware.

For this composition, similar to the hardware composition, you must have the correct Screen configuration and the appropriate window properties set.

Pipeline ordering and the z-ordering of windows on a layer are applied independently of each other.

Pipeline ordering takes precedence over z-ordering operations in Screen. Screen does not have control over the ordering of hardware pipelines. Screen windows are always arranged in the z-order that is specified by the application.

If your application manually assigns pipelines, you must ensure that the z-order values make sense with regard to the pipeline order of the target hardware. For example, if you assign a high z-order value to a window (meaning it is to

be placed in the foreground), then you must make a corresponding assignment of this window to a top layer pipeline. Otherwise the result may not be what you expect, regardless of the z-order value.

Comparing composition types

Both hardware and Screen composition types each have multiple advantages and disadvantages. Some are very subtle and sometimes depend on the rate at which the window's contents are refreshed.

	Hardware composition	Screen composition
Advantages	• Window buffers don't need to be copied to a composite framebuffer • No processing power of CPU and/or GPU required to compose buffers • Efficient in handling windows with high-frequency updates	• Not as limited by pipeline capabilities • Able to display a software cursor or to draw a background • Able to compose multiple buffers for display with only a single pipeline • May be able to create windows with a buffer format that is not supported by a pipeline (i.e., Screen composition may be able to convert the format to one supported by the pipeline when it copies the window buffer)
Disadvantages	• Limited by pipeline capabilities, which can vary from platform to platform • Limited by the number of supported pipelines, which can vary from platform to platform • Limited by format support on pipeline • Can't display more than one buffer per pipeline	• Window buffers, or part of them, need to be copied to a composite buffer • May require processing power of CPU and/or GPU to compose buffers

Chapter 4
Screen API

The Screen API is how your applications communicate with Screen.

The principal components of the Screen API are closely associated with each other.

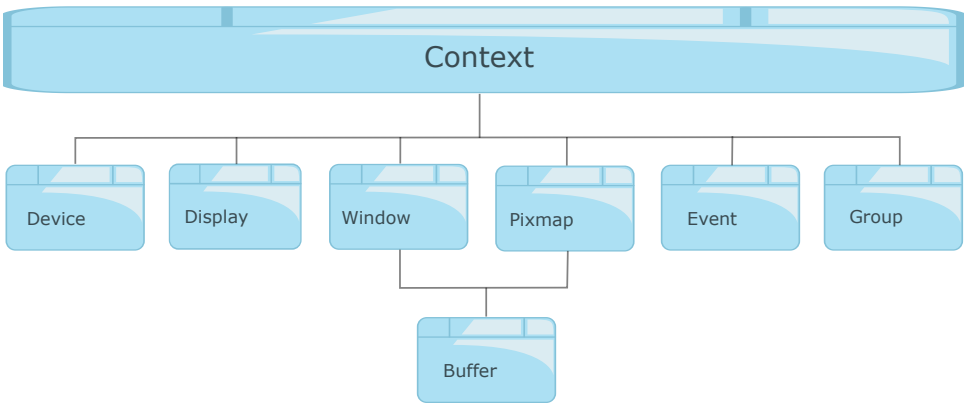


Figure 7: Screen API components

Context

A context provides the setting for graphics operations within the windowing environment.

All other API objects are created within the scope of a context and access to these objects is always with respect to the context associated with the object. You can identify and gain access to the objects on which you want to draw (e.g., windows, groups, displays, pixmaps) to set or change their properties and attributes.

Devices, displays and windows are dependent on the context, which is associated directly with events, groups, and pixmaps.

Device

A device refers to an input device. This input device (e.g., keyboard, mouse, joystick, gamepad, and multi-touch) can be focused to specific displays.

Display

A display refers to a physical device that presents images to viewers such as monitors, touchscreen and displays. Using the display-specific API components, you can gain access to display properties, modes, and vsync operations.

Window

A window represents the fundamental drawing surface. Windows can display different kinds of content for different purposes, and so there are multiple types of windows available: application windows, child windows, and embedded windows.

Pixmap

A pixmap is similar to a bitmap except that it can have multiple bits per pixel (a measurement of the depth of the pixmap) that store the intensity or color component values. Bitmaps, by contrast, have a depth of one bit per pixel.

You can draw directly onto a pixmap surface, outside the viewable area, and then copy the pixmap to a buffer later on.

Event

An event includes such actions as window creation, setting properties, keyboard events, and touch events. Events are associated with a context. Screen API manages one event queue per context.

Group

A group is used to organize and manage multiple windows in your application. Windows belonging to a group share the same properties; therefore, you apply sets of properties to all the windows that are in the same group.

Buffer

A buffer is an area of memory not displayed where you can move data around quickly without taking up CPU cycles. Although a buffer can be created in the scope of a context, it cannot be used unless attached to a window or pixmap.

Multiple buffers can be associated with a window whereas only one buffer can be associated with a pixmap.

Chapter 5
Contexts

The context defines the relationship with the underlying window system.

You can use the context to get and set display and window properties that define window idle times, keyboard and multi-touch focus settings. You can also use the context to return the number of displays on the current system. A context can be associated with a single window, with a group of windows, or with one or more displays.

The following tasks describe how to perform basic context operations.

Create a context

You must create a context before you create a window. When you call *screen_create_context()*, memory is allocated to store the context state. The composition manager creates an event queue and associates it with the connecting process.

To create a context:

1. Create and initialize the context variable.

   ```
   screen_context_t screen_context = 0;
   ```

2. Call *screen_create_context()* to create the context. The *screen_create_context()* function takes a reference to a variable of type `screen_context_t`, and a flag that represents the type of context. In the example below, the context is of type `SCREEN_APPLICATION_CONTEXT` indicating that the context can only create and modify windows within the scope of the current application.

   ```
   if (screen_create_context(&screen_context,
   SCREEN_APPLICATION_CONTEXT) != 0) {
      return EXIT_FAILURE;
   }
   ```

You must destroy each context and free up the memory whenever your application is done with it. To destroy a context, call the *screen_destroy_context()* function.

```
screen_destroy_context(screen_context);
```

Context types

When you create a context, you must specify a flag in order to define the type of context. The context defines the connection between your application and the underlying windowing system. Depending on the needs of your application, and in some cases, the permissions of your application, there are a number of different context types available.

The following context types are supported for your system:

Flag	Description	`root` permission required?
`SCREEN_APPLICATION_CONTEXT`	This context type enables a process to create its own windows and to control some of the window properties. An application cannot modify a window that was created by another application and it cannot send an event outside of its own process space. An application's context is unaware of other top-level windows in the system. An application context can parent another window, even if the window is created in another context within another processes.	No
`SCREEN_WINDOW_MANAGER_CONTEXT`	This context type enables a process to modify all windows in the system whenever new application windows are created or destroyed. The context also receives notifications when an application creates new windows, when existing application windows are destroyed, or when an application tries to change certain window properties.	Yes
`SCREEN_INPUT_PROVIDER_CONTEXT`	This context type enables a process to send an event to any application in the system. This context does not receive notifications when applications create new windows, when applications destroy existing windows, or when an application attempts to change certain window properties.	Yes
`SCREEN_POWER_MANAGER_CONTEXT`	This context type provides access to power management functionality.	Yes
`SCREEN_DISPLAY_MANAGER_CONTEXT`	This context type provides access to display properties.	Yes

Set a context property

You can get and set context properties in order to define how your application will behave within a window.

In Screen API, many get and set methods contain multiple variants, with each variant corresponding to the type that is associated with a property. For example, the *screen_get_context_property_iv()* method takes an integer, while the *screen_get_context_property_llv()* takes a long long integer.

To set a context property:

1. Create a variable to pass into the function. The type must match the variant of the function, and the value must represent a valid flag. In the example below, a Screen format flag is passed into the function.

   ```
   int context_idle = 5;
   ```

2. Call the variant function. The *screen_set_context_property_iv()* function takes a reference to an integer that determines the length of time in seconds before the window will timeout.

   ```
   if (screen_set_context_property_iv(screen_context,
   SCREEN_PROPERTY_IDLE_TIMEOUT, &context_idle) !=0) {
           return EXIT_FAILURE;
   }
   ```

You can flush the context of any delayed commands by calling the *screen_flush_context()* function. When you call the *screen_flush_context()* function, any delayed commands are processed from the buffer, and any associated displays are updated. If you specify the `SCREEN_WAIT_IDLE` parameter, the function will not return until all associated displays have been updated.

```
if (screen_flush_context(screen_context, SCREEN_WAIT_IDLE) !=0) {
      return EXIT_FAILURE;
};
```

When debugging your application, it's a good idea to call the *screen_flush_context()* function after you call any delayed function. This will help you to determine the exact function call that caused the error.

Chapter 6
Windows

You can create a window group to organize a set of windows into a hierarchy.

The concept of a window in Screen differs slightly from what you're probably used to in a traditional windowing system. In Screen, applications are split into several windows when content comes from different sources, when one or more parts of the application must be updated independently from others, or when the application tries to target multiple displays. For example, a user interface that was developed in Adobe AIR can be overlayed on top of a native document viewer, or a plug-in can be embedded within a web view or document. Adobe AIR components can be used to form user interface controls for navigation or media playback. These controls can be contained within a window that is overlayed over top of a map or multimedia. In this example, the background window must be updated independently from the foreground user interface controls.

You must create window groups in order to organize, display, and control the windows in your application. A window group consists of a parent window and at least one child window. To create a group, a window must call the *screen_create_window_group()* function and provide a name for the group. The name of the window group is then communicated to the other functions, threads, or even processes, that are responsible for creating child windows. Any child window can join this group as long as it has the associated group name. The parent window is notified each time a child window joins the group. A window handle is included in the notification to allow the parent window to control certain properties of a child window such as visibility, position, size, and z-order. A parent window cannot access any of the child windows' buffers. A child window remains invisible until it is added to a window group and is made visible by the owner of the group.

Window types

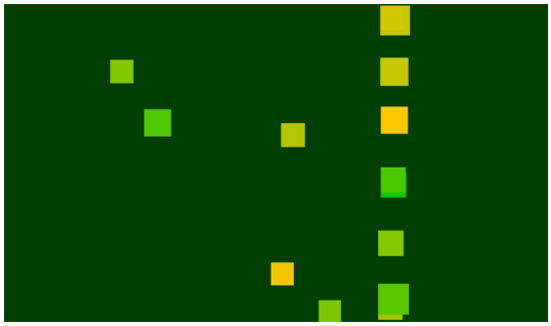
There are multiple window types in the Screen API. Each window type has a different use and different positioning rules, and each window is typically used to display different types of content.

You specify the window type at window creation time. The following types are available.

SCREEN_APPLICATION_WINDOW

The window type that's used to display the main application. The X and Y coordinates are always relative to the dimensions of the display.

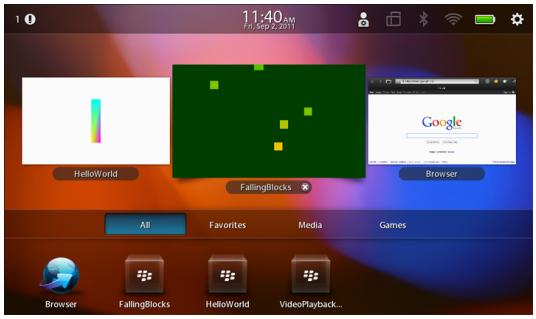
Application windows can be used to display an application in fullscreen mode:



SCREEN_CHILD_WINDOW

The subwindow type that's commonly used to display a dialog. You must add a child window to an application's window group; otherwise, the child window is invisible. A child window's display properties are relative to the application window to which it belongs. For example, the X and Y coordinates of the child window are all relative to the top-left corner of the application window.

Child windows can be used to display minimized applications:



SCREEN_EMBEDDED_WINDOW

Used to embed a window control within an object. Like the child window, the X and Y coordinates of the embedded window are all relative to the top-left corner of the application window. You must add an embedded window to an application's window group, otherwise the embedded window is invisible.

Window properties

Screen API window properties are the properties of a window API object.

Screen distinguishes between parent and owner window properties.

Parent window properties

Parent window properties are the properties of the window that can be changed by a parent application window or a window manager. A window manager's access to properties is more limited than that of a parent window. A window manager can only change the properties of top-level windows and windows that are created by the window manager itself.

The owner of the window is allowed to set the parent window properties only if the parent window has set its SCREEN_PROPERTY_SELF_LAYOUT property to true, or when there is no parent window or window manager present.

Only the parent window has permission to set the SCREEN_PROPERTY_SELF_LAYOUT property—even if SCREEN_PROPERTY_SELF_LAYOUT has been set to true.

Owner window properties

Owner window properties are properties that can be changed by the owner of the window; some owner window properties can be changed by both the owner and the parent of the window.

Window Property Permissions

The following table lists all of the window properties and indicates whether each can be changed by the parent window, the window manager or the window owner:

Window Property	Parent Window/Window Manager	Window Owner
SCREEN_PROPERTY_ALPHA_MODE	No	Yes
SCREEN_PROPERTY_ALTERNATE_WINDOW	No	Yes
SCREEN_PROPERTY_ALTERNATE_BRIGHTNESS	Yes	Yes
SCREEN_PROPERTY_BRUSH	No	Yes
SCREEN_PROPERTY_BRUSH_CLIP_POSITION	No	Yes
SCREEN_PROPERTY_BRUSH_CLIP_SIZE	No	Yes

Window Property	Parent Window/Window Manager	Window Owner
SCREEN_PROPERTY_BUFFER_COUNT	No	Yes
SCREEN_PROPERTY_BUFFER_SIZE	No	Yes
SCREEN_PROPERTY_CBABC_MODE	No	Yes
SCREEN_PROPERTY_CLASS	Yes	Yes
SCREEN_PROPERTY_CLIP_POSITION	Yes	No
SCREEN_PROPERTY_CLIP_SIZE	Yes	No
SCREEN_PROPERTY_COLOR	No	Yes
SCREEN_PROPERTY_COLOR_SPACE	No	Yes
SCREEN_PROPERTY_CONTRAST	Yes	Yes
SCREEN_PROPERTY_DEBUG	Yes	Yes
SCREEN_PROPERTY_DISPLAY	Yes	Yes
SCREEN_PROPERTY_FLIP	Yes	Yes
SCREEN_PROPERTY_FLOATING	Yes	Yes
SCREEN_PROPERTY_FORMAT	No	Yes
SCREEN_PROPERTY_GLOBAL_ALPHA	Yes	Yes
SCREEN_PROPERTY_HUE	Yes	Yes
SCREEN_PROPERTY_ID_STRING	No	Yes
SCREEN_PROPERTY_IDLE_MODE	No	Yes
SCREEN_PROPERTY_MIRROR	Yes	Yes
SCREEN_PROPERTY_PIPELINE	Yes	Yes
SCREEN_PROPERTY_POSITION	Yes	No
SCREEN_PROPERTY_PROTECTION_ENABLE	No	Yes
SCREEN_PROPERTY_ROTATION	Yes	Yes
SCREEN_PROPERTY_SATURATION	Yes	Yes
SCREEN_PROPERTY_SCALE_QUALITY	Yes	Yes
SCREEN_PROPERTY_SELF_LAYOUT	Yes	No
SCREEN_PROPERTY_SENSITIVITY	No	Yes
SCREEN_PROPERTY_SIZE	Yes	No

Window Property	Parent Window/Window Manager	Window Owner
SCREEN_PROPERTY_SOURCE_CLIP_POSITION	No	Yes
SCREEN_PROPERTY_SOURCE_CLIP_SIZE	No	Yes
SCREEN_PROPERTY_SOURCE_POSITION	Yes	Yes
SCREEN_PROPERTY_SOURCE_SIZE	Yes	Yes
SCREEN_PROPERTY_STATIC	No	Yes
SCREEN_PROPERTY_SWAP_INTERVAL	No	Yes
SCREEN_PROPERTY_TRANSPARENCY	No	Yes
SCREEN_PROPERTY_USAGE	No	Yes
SCREEN_PROPERTY_VIEWPORT_POSITION	No	Yes
SCREEN_PROPERTY_VIEWPORT_SIZE	No	Yes
SCREEN_PROPERTY_VISIBLE	Yes	No
SCREEN_PROPERTY_ZORDER	Yes	No

Window parenting and positioning

The window type determines what positioning rules are applied to a child window once the window joins a group. A window's type also determines whether or not it can parent another window.

The following window properties are relative to the parent window:

visibility

A child window is visible only when the associated parent window is visible.

z-order

The z-order of a child window is relative to the parent window. For example, a positive value will place the child on top of (or above) its associated parent window. Conversely, a negative z-order puts the child window underneath the parent window.

position

The position of the child window is relative to the position of the parent. Any translation of the parent also affects the child.

size

The size of the child is relative to the parent. Any scaling applied to the parent is also applied to the child.

transparency

The global alpha of a child window is combined with the global alpha of the parent.

Application windows

An application window is positioned according to absolute screen coordinates. Therefore, an application window cannot be the parent of another application window. An application window is implicitly part of a group that is owned by the window manager if the application window has registered with Screen.

Child windows

A child window is not visible until it has become the child of an application window. It does so by using the function, *screen_join_window_group()* to join the window group created by its parent application window.

Embedded windows

An embedded window can join a window group whose parent is an application window or a child window. An embedded window behaves like child window except that it is clipped to the parent window's destination rectangle.

The embedded window type provides the illusion that the contents of both the parent and child window represent a single logical view. When you scroll, the view of the content in the parent window and the position of the embedded window are updated synchronously.

When you zoom in the parent window, the embedded window will change size and be repositioned independently, without the need for the parent to update the embedded window. Thus, the position and size of embedded windows are relative to the source rectangle and the virtual viewport of the associated parent window.

When you pan the source rectangle of the parent window within a larger buffer, the position of any embedded window will be updated automatically. Alternatively, an application can move a virtual viewport instead of the source rectangle and achieve the same effect without requiring a window buffer that is larger than the source size.

All of these rules were created to abstract window managers and group parents from the underlying window hierarchy. The window manager can move, fade, or scale a window, and the results will be the same whether the window is a single window or a more complex hierarchy of several windows.

Create a window

Before you can render an animation or display video, you must create a window for your application. There are a number of different window types. The following procedure describes how to create a window that can be used to display video.

To create a window:

1. Create a variable for the context and window instances and create a variable to store the name of the windowgroup.

```
screen_context_t   screen_context = 0;
screen_window_t    screen_window = 0;
static const char  *window_group_name = "mainwindowgroup";
```

2. Create a context. The context describes the relationship between the application and the underlying windowing system.

```
screen_create_context(&screen_context,
SCREEN_APPLICATION_CONTEXT);
```

3. Create a window. The *screen_create_window()* function takes the window variable and the context variable that you created in the first step.

```
screen_create_window(&screen_window, screen_context);
```

4. Create a window group. The *window_group_name* variable stores the name of the main window group. Remember that the name of the window group must be unique. You must add your application window to a window group in order to make the window visible.

```
screen_create_window_group(screen_window, window_group_name);
```

5. Set the window properties. In the following step, the pixel format and usage values are set for the window. In this example, the window will be used to display a video.

```
int format = SCREEN_FORMAT_RGBA8888;
screen_set_window_property_iv(screen_window,
SCREEN_PROPERTY_FORMAT, &format);

int usage = SCREEN_USAGE_NATIVE;
screen_set_window_property_iv(screen_window,
SCREEN_PROPERTY_USAGE, &usage);
```

6. Create a window buffer. In this example, the buffer is used to store video data for the window. The *screen_create_window_buffers()* function takes the window and an integer that defines the number of buffers to create for this window.

   ```
   screen_create_window_buffers(screen_window, 1);
   ```

Although any instances created are destroyed when the application exits, it is best practice to destroy any window, pixmap and context instances that you created but no longer require.

The following code snippet is included at the end of the application above.

```
screen_destroy_window(screen_window);
screen_destroy_context(screen_context);
```

Create a child window

You can use the *screen_create_window_type()* function to create a child window.

To create a child window:

1. Create a variable for each of the context and window instances.

```
screen_context_t    screen_context = 0;
screen_window_t     screen_child_window = 0;
```

2. Create a context. The context describes the relationship between the application and the underlying windowing system.

```
screen_create_context(&screen_context,
SCREEN_APPLICATION_CONTEXT);
```

3. Create a child window. The *screen_create_window_type()* function takes the window and context variables and an integer representing the window type. In this case, the window is of type `SCREEN_CHILD_WINDOW`.

```
int wintype = SCREEN_CHILD_WINDOW;
screen_create_window_type(&screen_child_window, screen_context,
 wintype );
```

4. Join a window group. The *window_group_name* should be the name of the window group created by the parent (or main) window through the *screen_create_window_group()* function.

```
screen_join_window_group(screen_child_window, window_group_name);
```

Pixel formats

Window pixel formats define how color space information is stored in the GPU memory.

The RGBA color space uses the Red Green Blue (RGB) color model with extra information about the alpha (transparency or opacity) channel. Applications that want to disregard the alpha channel can choose a pixel format with an X.

The Screen API supports the following window pixel formats (pixel format descriptions from *www.fourcc.org*):

Format	Description
SCREEN_FORMAT_BYTE	
SCREEN_FORMAT_RGBA4444	16 bits per pixel (4 bits per channel) RGB with alpha channel.
SCREEN_FORMAT_RGBX4444	16 bits per pixel (4 bits per channel) RGB with alpha channel disregarded.
SCREEN_FORMAT_RGBA5551	16 bits per pixel, 2 bytes containing R, G, and B values (5 bits per channel with single bit alpha channel).
SCREEN_FORMAT_RGBX5551	16 bits per pixel, 2 bytes containing R, G, and B values (5 bits per channel with single bit alpha channel, disregarded).
SCREEN_FORMAT_RGB565	16 bits per pixel; uses five bits for red, six bits for green and five bits for blue. This pixel format represents each pixel in the following order (high byte to low byte): RRRR RGGG GGGB BBBB.
SCREEN_FORMAT_RGB888	24 bits per pixel (8 bits per channel) RGB.
SCREEN_FORMAT_RGBA8888	32 bits per pixel (8 bits per channel) RGB with alpha channel.
SCREEN_FORMAT_RGBX8888	32 bits per pixel (8 bits per channel) RGB with alpha channel disregarded.
SCREEN_FORMAT_YVU9	9 bits per pixel planar YUV format. 8-bit Y plane and 8-bit 4x4 subsampled V and U planes. Registered by Intel.
SCREEN_FORMAT_YUV420	Standard NTSC TV transmission format.
SCREEN_FORMAT_NV12	12 bits per pixel planar YUV format. 8-bit Y plane and 2x2 subsampled, interleaved U and V planes.
SCREEN_FORMAT_YV12	12 bits per pixel planar YUV format. 8-bit Y plane and 8-bit 2x2 subsampled U and V planes.
SCREEN_FORMAT_UYVY	16 bits per pixel packed YUV format. YUV 4:2:2 — Y sample at every pixel, U and V sampled at every second pixel horizontally on each line. A macropixel contains 2 pixels in 1 u_int32.

Format	Description
SCREEN_FORMAT_YUY2	16 bits per pixel packed YUV format. YUV 4:2:2 as in UYVY, but with different component ordering within the u_int32 macropixel.
SCREEN_FORMAT_YVYU	16 bits per pixel packed YUV format. YUV 4:2:2 as for UYVY, but with different component ordering within the u_int32 macropixel.
SCREEN_FORMAT_V422	Packed YUV format. Inverted version of UYVY.
SCREEN_FORMAT_AYUV	Packed YUV format. Combined YUV and alpha.

Chapter 7
Displays

A display represents the physical display hardware such as a monitor or touchscreen display.

You can use display API functions to:

- query and set display properties
- get display modes that are specific to a given hardware display
- perform vsync operations

Note that to have full access to the display properties of the system, you must be working within a privileged context. You create a privileged context by calling the function *screen_create_context()* with a context type of `SCREEN_DISPLAY_MANAGER_CONTEXT`. Your process must have an effective user ID of root to be able to create this context type. Some API functions will fail to execute if you are not the correct context.

Multiple displays

It can be quite tricky to create and manage an application that uses multiple displays, especially when you consider threading, performance, and graphics optimization. Fortunately, Screen API provides the necessary functionality to let you create applications that write to multiple windows and displays simultaneously.

In our vsync application, an hour glass is placed in the top-left corner of an application window while a vertical bar sweeps from left to right across the screen. This code sample queries the context to determine the number of displays that are currently attached to the system. When the bar reaches the right edge of the application window, instead of returning to the left-hand side of the current display, the application sends focus to the next display in the list and the bar continues its sweep at the left-hand side of that other display.

Figure 8: The sample vsync application

The application uses a `struct` to store the state (either detached, attached, or focused) of each display.

```
struct {
   pthread_mutex_t mutex;
   pthread_cond_t cond;
   enum { detached, attached, focused } state;
} *displays;
```

Before any of the drawing is done, the application iterates though each attached display, and uses the *screen_get_display_property_iv()* property to return the state of the current display. For each attached display, the application initializes a mutex and calls the *pthread_create()*, passing in a *display()* function, to spawn a child thread

This *display()* function handles all graphics operations for the current display, meaning that each display will be written to and updated within its own process. This allows the graphics processor to handle any intensive operations, and ensures that if an error occurs or a display becomes detached, the application will not fail.

```
displays = calloc(ndisplays, sizeof(*displays));
for (i = 0; i < ndisplays; i++) {
   int active = 0;
   screen_get_display_property_iv(screen_dpy[i], SCREEN_PROPERTY_ATTACHED, &active);
```

```
   if (active) {
      if (idx == -1) {
         displays[i].state = focused;
         idx = i;
      } else {
        displays[i].state = attached;
      }
   } else {
      displays[i].state = detached;
   }
   pthread_mutex_init(&displays[i].mutex, NULL);
   pthread_cond_init(&displays[i].cond, NULL);

   pthread_t thread;
   pthread_create(&thread, NULL, display, (void *)i);
}
```

The *display()* function sets up the current display and window, then locks the mutex to determine whether or not the current display is active and has focus.

```
pthread_mutex_lock(&displays[idx].mutex);
attached = displays[idx].state != detached ? 1 : 0;
focus = displays[idx].state == focused ? 1 : 0;
pthread_mutex_unlock(&displays[idx].mutex);
```

A while loop checks conditions and handles the flow of execution for ach display. If the display is currently attached, the display and window properties are initialized. A buffer is created, a handle to the buffer is returned, and the background color is blitted to the buffer. If the display has focus, the bar is blitted and written to the buffer at the current position. Next, the hourglass is written to the buffer and the window is posted.

The *pos* variable is incremented continuously, causing the bar to scan from left to right across the current application window. When the bar reaches the right-most edge of the screen, the mutex of the next display in the `displays` structure is locked and the state of the next attached display is set to *focused*. This causes the bar to appear at the left-most edge of the next display in the display list. It then scans across the screen and repeats this behavior on the next display in the list of displays.

```
while (1) {
    if (attached) {
        if (!realized) {
            screen_get_display_property_iv(screen_dpy[idx], SCREEN_PROPERTY_SIZE, rect+2);
            screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);
            screen_create_window_buffers(screen_win, 2);
            realized = 1;
        }

        screen_buffer_t screen_buf[2];
        screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)screen_buf

        int bg[] = { SCREEN_BLIT_COLOR, 0xffffff00, SCREEN_BLIT_END };
        screen_fill(screen_ctx, screen_buf[0], bg);

        if (focus > 0) {
            int bar[] = {
                SCREEN_BLIT_COLOR, 0xff0000ff,
                SCREEN_BLIT_DESTINATION_X, pos,
                SCREEN_BLIT_DESTINATION_WIDTH, barwidth,
                SCREEN_BLIT_END };

            screen_fill(screen_ctx, screen_buf[0], bar);

            if (++pos > rect[2] - barwidth) {
                for (i = (idx+1) % ndisplays; i != idx; i = (i+1) % ndisplays) {
                    pthread_mutex_lock(&displays[i].mutex);
                    if (displays[i].state == attached) {
                        displays[i].state = focused;
                        pthread_cond_signal(&displays[i].cond);
                        pthread_mutex_unlock(&displays[i].mutex);
                        break;
                    }
                    pthread_mutex_unlock(&displays[i].mutex);
```

```
                    }
                    if (i != idx) {
                        pthread_mutex_lock(&displays[idx].mutex);
                        displays[idx].state = attached;
                        pthread_mutex_unlock(&displays[idx].mutex);
                        focus = -1;
                    }
                    pos = 0;
                }
            } else {
                focus = 0;
            }

            int hg[] = {
                SCREEN_BLIT_SOURCE_WIDTH, 100,
                SCREEN_BLIT_SOURCE_HEIGHT, 100,
                SCREEN_BLIT_DESTINATION_X, 10,
                SCREEN_BLIT_DESTINATION_Y, 10,
                SCREEN_BLIT_DESTINATION_WIDTH, 100,
                SCREEN_BLIT_DESTINATION_HEIGHT, 100,
                SCREEN_BLIT_TRANSPARENCY, SCREEN_TRANSPARENCY_SOURCE_OVER,
                SCREEN_BLIT_END
            };

            screen_blit(screen_ctx, screen_buf[0], screen_pbuf, hg);
            screen_post_window(screen_win, screen_buf[0], 1, rect, 0);
        }

        if (!attached && realized) {
            screen_destroy_window_buffers(screen_win);
            screen_flush_context(screen_ctx, 0);
            realized = 0;
        }

        if (focus != -1) {
            pthread_mutex_lock(&displays[idx].mutex);
            if (!focus) {
                printf("%s[%d]: idx=%d\n", __FUNCTION__, __LINE__, idx);
                pthread_cond_wait(&displays[idx].cond, &displays[idx].mutex);
                pos = 0;
            }
            attached = displays[idx].state != detached ? 1 : 0;
            focus = displays[idx].state == focused ? 1 : 0;
            pthread_mutex_unlock(&displays[idx].mutex);
        }
    }
```

The main body of the application handles window and display events. It updates the `displays` struct every time a display is attached or detached. In addition to controlling the flow of execution for the application, it also prints out debug information about the current execution.

While this is the most complicated of the sample applications, it does provide many useful best practices for handling and writing to multiple displays.

Complete sample: Using multiple displays

The complete code sample is listed below.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <screen/screen.h>

const int barwidth = 32;

int ndisplays = 0;

struct {
 pthread_mutex_t mutex;
 pthread_cond_t cond;
 enum { detached, attached, focused } state;
} *displays;

screen_pixmap_t screen_pix = NULL;

screen_buffer_t screen_pbuf = NULL;

void pixmap(screen_context_t screen_ctx)
{
```

```
 int i, j;

 screen_create_pixmap(&screen_pix, screen_ctx);

 int format = SCREEN_FORMAT_RGBA8888;
 screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_FORMAT, &format);

 int usage = SCREEN_USAGE_WRITE | SCREEN_USAGE_NATIVE;
 screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_USAGE, &usage);

 int size[2] = { 100, 100 };
 screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_BUFFER_SIZE, size);

 screen_create_pixmap_buffer(screen_pix);
 screen_get_pixmap_property_pv(screen_pix, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_pbuf);

 unsigned char *ptr = NULL;
 screen_get_buffer_property_pv(screen_pbuf, SCREEN_PROPERTY_POINTER, (void **)&ptr);

 int stride = 0;
 screen_get_buffer_property_iv(screen_pbuf, SCREEN_PROPERTY_STRIDE, &stride);

 for (i = 0; i < size[1]; i++, ptr += stride) {
  for (j = 0; j < size[0]; j++) {
   ptr[j*4] = 0xa0;
   ptr[j*4+1] = 0xa0;
   ptr[j*4+2] = 0xa0;
   ptr[j*4+3] = ((j >= i && j <= size[1]-i) || (j <= i && j >= size[1]-i)) ? 0xff : 0;
  }
 }
}

void *display(void *arg)
{
 const int idx = (int)arg;
 int rect[4] = { 0, 0 };
 int realized = 0;
 int pos = 0;
 int attached;
 int focus;
 int i;

 screen_context_t screen_ctx;
 screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);

 screen_display_t *screen_dpy = calloc(ndisplays, sizeof(screen_display_t));
 screen_get_context_property_pv(screen_ctx, SCREEN_PROPERTY_DISPLAYS, (void **)screen_dpy);

 screen_window_t screen_win;
 screen_create_window(&screen_win, screen_ctx);
 screen_set_window_property_pv(screen_win, SCREEN_PROPERTY_DISPLAY, (void **)&screen_dpy[idx]);

 int usage = SCREEN_USAGE_NATIVE;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &usage);

 pthread_mutex_lock(&displays[idx].mutex);
 attached = displays[idx].state != detached ? 1 : 0;
 focus = displays[idx].state == focused ? 1 : 0;
 pthread_mutex_unlock(&displays[idx].mutex);

 screen_event_t screen_ev;
 screen_create_event(&screen_ev);

 while (1) {
  if (attached) {
   if (!realized) {
    screen_get_display_property_iv(screen_dpy[idx], SCREEN_PROPERTY_SIZE, rect+2);
    screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);
    screen_create_window_buffers(screen_win, 2);
    realized = 1;
   }

   screen_buffer_t screen_buf[2];
   screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)screen_buf);

   int bg[] = { SCREEN_BLIT_COLOR, 0xffffff00, SCREEN_BLIT_END };
   screen_fill(screen_ctx, screen_buf[0], bg);

   if (focus > 0) {
    int bar[] = {
     SCREEN_BLIT_COLOR, 0xff0000ff,
     SCREEN_BLIT_DESTINATION_X, pos,
     SCREEN_BLIT_DESTINATION_WIDTH, barwidth,
     SCREEN_BLIT_END };

    screen_fill(screen_ctx, screen_buf[0], bar);

    if (++pos > rect[2] - barwidth) {
     for (i = (idx+1) % ndisplays; i != idx; i = (i+1) % ndisplays) {
      pthread_mutex_lock(&displays[i].mutex);
      if (displays[i].state == attached) {
```

```
          displays[i].state = focused;
          pthread_cond_signal(&displays[i].cond);
          pthread_mutex_unlock(&displays[i].mutex);
          break;
         }
         pthread_mutex_unlock(&displays[i].mutex);
        }
        if (i != idx) {
         pthread_mutex_lock(&displays[idx].mutex);
         displays[idx].state = attached;
         pthread_mutex_unlock(&displays[idx].mutex);
         focus = -1;
        }
        pos = 0;
       }
      } else {
       focus = 0;
      }

      int hg[] = {
       SCREEN_BLIT_SOURCE_WIDTH, 100,
       SCREEN_BLIT_SOURCE_HEIGHT, 100,
       SCREEN_BLIT_DESTINATION_X, 10,
       SCREEN_BLIT_DESTINATION_Y, 10,
       SCREEN_BLIT_DESTINATION_WIDTH, 100,
       SCREEN_BLIT_DESTINATION_HEIGHT, 100,
       SCREEN_BLIT_TRANSPARENCY, SCREEN_TRANSPARENCY_SOURCE_OVER,
       SCREEN_BLIT_END
      };

      screen_blit(screen_ctx, screen_buf[0], screen_pbuf, hg);
      screen_post_window(screen_win, screen_buf[0], 1, rect, 0);
     }

     if (!attached && realized) {
      screen_destroy_window_buffers(screen_win);
      screen_flush_context(screen_ctx, 0);
      realized = 0;
     }

     if (focus != -1) {
      pthread_mutex_lock(&displays[idx].mutex);
      if (!focus) {
       printf("%s[%d]: idx=%d\n", __FUNCTION__, __LINE__, idx);
       pthread_cond_wait(&displays[idx].cond, &displays[idx].mutex);
       pos = 0;
      }
      attached = displays[idx].state != detached ? 1 : 0;
      focus = displays[idx].state == focused ? 1 : 0;
      pthread_mutex_unlock(&displays[idx].mutex);
     }
    }

    free(screen_dpy);

    return NULL;
}

int main(int argc, char **argv)
{
    int i, j, idx = -1;

    screen_context_t screen_ctx;
    screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);
    screen_get_context_property_iv(screen_ctx, SCREEN_PROPERTY_DISPLAY_COUNT, &ndisplays);

    pixmap(screen_ctx);

    screen_display_t *screen_dpy = calloc(ndisplays, sizeof(screen_display_t));
    screen_get_context_property_pv(screen_ctx, SCREEN_PROPERTY_DISPLAYS, (void **)screen_dpy);

    displays = calloc(ndisplays, sizeof(*displays));
    for (i = 0; i < ndisplays; i++) {
     int active = 0;
     screen_get_display_property_iv(screen_dpy[i], SCREEN_PROPERTY_ATTACHED, &active);
     if (active) {
      if (idx == -1) {
       displays[i].state = focused;
       idx = i;
      } else {
       displays[i].state = attached;
      }
     } else {
      displays[i].state = detached;
     }

     pthread_mutex_init(&displays[i].mutex, NULL);
     pthread_cond_init(&displays[i].cond, NULL);

     pthread_t thread;
     pthread_create(&thread, NULL, display, (void *)i);
```

```
    }

    screen_event_t screen_ev;
    screen_create_event(&screen_ev);

    while (1) {
     int type = SCREEN_EVENT_NONE;
     screen_get_event(screen_ctx, screen_ev, ~0);
     screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &type);
     if (type == SCREEN_EVENT_DISPLAY) {
      screen_display_t tmp = NULL;
      screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_DISPLAY, (void **)&tmp);
      for (i = 0; i < ndisplays; i++) {
       if (tmp == screen_dpy[i]) {
        int active = 0;
        screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_ATTACHED, &active);

        if (active) {
         int size[2];
         screen_get_display_property_iv(tmp, SCREEN_PROPERTY_SIZE, size);
         if (size[0] == 0 || size[1] == 0) {
          active = 0;
         }
        }

        pthread_mutex_lock(&displays[i].mutex);
        if ((active && displays[i].state == detached) ||
         (!active && displays[i].state != detached)) {
         if (active) {
          for (j = 0; j < ndisplays; j++) {
           printf("%s[%d]: j=%d\n", __FUNCTION__, __LINE__, j);
           if (i != j) {
            pthread_mutex_lock(&displays[j].mutex);
            if (displays[j].state == focused) {
             displays[i].state = attached;
             pthread_mutex_unlock(&displays[j].mutex);
             break;
            }
            pthread_mutex_unlock(&displays[j].mutex);
           }
          }
          if (displays[i].state == detached) {
           displays[i].state = focused;
          }
         } else {
          if (displays[i].state == focused) {
           for (j = (i+1) % ndisplays; j != i; j = (j+1) % ndisplays) {
            printf("%s[%d]: j=%d\n", __FUNCTION__, __LINE__, j);
            pthread_mutex_lock(&displays[j].mutex);
            if (displays[j].state == attached) {
             displays[j].state = focused;
             pthread_cond_signal(&displays[j].cond);
             pthread_mutex_unlock(&displays[j].mutex);
             break;
            }
            pthread_mutex_unlock(&displays[j].mutex);
           }
          }
          displays[i].state = detached;
         }
         pthread_cond_signal(&displays[i].cond);
        }
        pthread_mutex_unlock(&displays[i].mutex);

        break;
       }
      }
     }
    }

    screen_destroy_pixmap(screen_pix);
    screen_destroy_event(screen_ev);
    screen_destroy_context(screen_ctx);

    free(displays);

    return EXIT_SUCCESS;
}
```

Chapter 8
Event Types

The event type is used to determine what caused the event to be dispatched.

You can query the event type by calling the *screen_get_event_property_iv()*, specifying the SCREEN_PROPERTY_TYPE constant.

```
int type;

screen_event_t screen_ev;
screen_create_event(&screen_ev);

screen_get_event(screen_ctx, screen_ev, -1);
screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &type);
```

The following event types are supported by the Screen API.

Event	Description
SCREEN_EVENT_NONE	A blocking event indicating that no events are currently in the queue.
SCREEN_EVENT_CREATE	Dispatched when a child window is created.
SCREEN_EVENT_PROPERTY	Dispatched when a property is set.
SCREEN_EVENT_CLOSE	Dispatched when a child window is destroyed.
SCREEN_EVENT_INPUT	Dispatched when an unknown input event occurs.
SCREEN_EVENT_JOG	Dispatched when a jog dial input event occurs.
SCREEN_EVENT_POINTER	Dispatched when a pointer input event occurs.
SCREEN_EVENT_KEYBOARD	Dispatched when a keyboard input event occurs.
SCREEN_EVENT_USER	Dispatched when a user event is detected.
SCREEN_EVENT_POST	Dispatched when a child window has posted its first frame.
SCREEN_EVENT_EFFECT_COMPLETE	Dispatched to the window manager indicating that a rotation effect has completed.
SCREEN_EVENT_DISPLAY	Dispatched when an external display is detected.
SCREEN_EVENT_IDLE	Dispatched when the window enters idle state.
SCREEN_EVENT_UNREALIZE	Dispatched when a handle to a window is lost.
SCREEN_EVENT_GAMEPAD	Dispatched when a gamepad input event occurs.
SCREEN_EVENT_JOYSTICK	Dispatched when a joystick input event occurs.

Event	Description
SCREEN_EVENT_DEVICE	Dispatched when an input device is detected.
SCREEN_EVENT_MTOUCH_TOUCH	Dispatched when a multi-touch event is detected.
SCREEN_EVENT_MTOUCH_MOVE	Dispatched when a multi-touch move event is detected, for example when the user moves their fingers to make an input gesture.
SCREEN_EVENT_MTOUCH_RELEASE	Dispatched when a multi-touch release event occurs, or when the user completes the multi-touch gesture.

Chapter 9
Screen Tutorials

Screen tutorials aim to help you understand how to use the API in your own applications by providing step-by-step guides.

Tutorial: Draw and perform vsync operations using windows

This simple sample application uses three windows to illustrate a basic drawing and vsync operation. The sample provides reusable functions that are called in the application to create and initialize windows. The sample also includes a complete event loop that you can use in your own application.

In the sample application, an hourglass is placed in the top-left corner of an application window while a vertical bar sweeps from left to right across the screen. The background window is of type `SCREEN_APPLICATION_WINDOW`, while the hourglass and the bar are implemented as windows of type `SCREEN_CHILD_WINDOW`. The application is shown below:

Figure 9: The sample vsync application

Each window in the application is given an ID at creation time. The ID is used at runtime to determine which window dispatched the event.

This sample application is used throughout this documentation to illustrate different ways to perform basic drawing and windowing tasks. The complete sample application is included below. You can copy and paste the complete application into a new project, or you can follow along with the tutorial below.

Create the background window

The background window is the parent window for the hourglass child window and the bar child window. This quick tutorial walks you through the process of creating the `create_bg_window()` function that is used to create the background window.

First, create and initialize the background window variable.

```
screen_window_t screen_bg_win = NULL;
```

Next, create the `create_bg_window()` function. This function is called to create the background window. The function takes a char that is used as a window group

name, an array of integers that define the size of the window, and a context for the window to use.

You can create the window with any buffer size. Note that the windowing system will simply scale the contents if the buffers are larger or smaller than the size of the window on the screen. However, it is preferable to make the buffer size match the on-screen dimensions of the window.

The *group* variable is used in the `screen_create_window_group()` function to specify the name for the window group. Since this is the parent window, all child windows must use this group ID in order to join the group.

```
screen_window_t create_bg_window(const char *group, int dims[2],
                                 screen_context_t screen_ctx)
{
    screen_window_t screen_win;
    screen_create_window(&screen_win, screen_ctx);
    screen_create_window_group(screen_win, group);

    ... more code not shown.
 }
```

Next, in the `create_bg_window()` function, set the window visibility to 0, indicating that the window will be invisible. It's important to hide all windows until the window and any associated buffer are properly initialized; otherwise the window will display incomplete results on the screen.

```
int vis = 0;
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);
```

Next, in the `create_bg_window()` function, set the background color to yellow. We're using a small trick here by filling the entire window with a solid color without requiring a large buffer to back it up, and without scaling.

```
int color = 0xffffff00;
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_COLOR, &color);
```

The second part of the trick is to create a 1x1 buffer. Currently, the Screen API doesn't support visible windows without at least one buffer. Instead, create the smallest buffer possible. The format and usage don't apply since the buffer will never be used. Next, to avoid scaling, the source viewport size is set to match the on-screen dimensions of the window.

```
int rect[4] = { 0, 0, 1, 1 };
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_SIZE, dims);
```

The final part of the trick is to move the source viewport completely outside the bounds of the 1x1 buffer. The Screen API allows this and replaces all areas outside the buffer with the window's color.

```
int pos[2] = { -dims[0], -dims[1] };
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_POSITION, pos);
```

Finally, in the `create_bg_window()` function, create the single 1x1 window buffer by calling the *screen_create_window_buffers()* function. Call the

screen_get_window_property_pv() function and specify the SCREEN_PROPERTY_RENDER_BUFFERS constant to return a handle to the buffer. This buffer must still be created in order to make it visible, even though it won't be used. Remember that the window is still invisible so nothing will appear on the display. The window will be made visible within the event loop, later on in the tutorial.

```
screen_buffer_t screen_buf;
screen_create_window_buffers(screen_win, 1);
screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_buf);
screen_post_window(screen_win, screen_buf, 1, rect, 0);
return screen_win;
```

Create the child windows

In the sample application, the hourglass and bar are implemented as child windows. This short walkthrough describes how to create the child windows for the hourglass and bar. Like the background window, the bar and hourglass windows never change; we need to fill a single buffer and only post it once.

Create the child window for the bar

First, create and initialize variables to store the hourglass and bar child windows. Also, create variables to store IDs for each of the window types. The IDs are used to identify each window during the event loop.

```
screen_window_t screen_hg_win = NULL;
screen_window_t screen_bar_win = NULL;

const char *hg_id_string = "hourglass";
const char *bar_id_string = "bar";
```

Next, create the `create_bar_window()` function. This will be used to create the child window for the bar. The function takes a string that is used as a window group name, a char that defines the ID of the window, and an array of integers that define the size of the window.

Note that you create a new `screen_context_t` instance for each window. Creating a separate context for each child window allows us to go over the steps required to deal with child windows that are created by other processes. Note that window permissions are handled per context, and not per process.

```
void create_bar_window(const char *group, const char *id, int dims[2])
{
 screen_context_t screen_ctx;
 screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);
}
```

Next, in the `create_bar_window()` function, create the child window by calling the *screen_create_window_type()* function and by specifying the SCREEN_CHILD_WINDOW constant. This establishes the window as a child window. Each child window is passed an ID (created earlier) so that it can be identified by the event loop.

After the window is created, the *screen_join_window_group()* function is called by specifying the ID of the main window group to which this child window will belong. Remember that the group ID was passed into the `create_bg_window()` function and used as the group ID for the parent window.

```
screen_window_t screen_win;
screen_create_window_type(&screen_win, screen_ctx, SCREEN_CHILD_WINDOW);
screen_join_window_group(screen_win, group);
screen_set_window_property_cv(screen_win, SCREEN_PROPERTY_ID_STRING, strlen(id), id);
```

Next, in the `create_bar_window()` function, set the window visibility to 0, making the window invisible. Setting the visibility is a responsibility of the parent. Parent windows must always set the visibility of each child window to true when appropriate.

```
int vis = 0;
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);
```

Next, in the `create_bar_window()` function, use the trick from the previous tutorial to set the background color of the bar.

```
int color = 0xff0000ff;
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_COLOR, &color);

int rect[4] = { 0, 0, 1, 1 };
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);

int pos[2] = { -rect[2], -rect[3] };
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_POSITION, pos);

screen_buffer_t screen_buf;
screen_create_window_buffers(screen_win, 1);
screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_buf);
screen_post_window(screen_win, screen_buf, 1, rect, 0);
```

Create the child window for the hourglass

The child window for the hourglass is created in much the same manner as the child window that contains the bar. You can see the complete code sample later on in the tutorial. The main differences are described below.

First, because the window will never change, use the static window property to tell Screen that the contents of the buffer won't change and aren't ever expected to post. This allows Screen to optimize the work required to put this window on the display.

```
int flag = 1;
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_STATIC, &flag);
```

Next, in the `create_hg_window()` function, set the pixel format. Because the hourglass shape will use transparency, you must use a pixel format with an alpha channel. Below, RGBA8888 is used.

```
int format = SCREEN_FORMAT_RGBA8888;
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_FORMAT, &format);
```

Next, in the `create_hg_window()` function, set the transparency property so that the hourglass window buffer source rectangle will appear over top of the background window. By default, RGBA8888 formats will have the transparency mode set to SCREEN_TRANSPARENCY_SOURCE_OVER. The windowing system assumes that if an

application chooses RGBA over RGBX, it intends to do at least some blending. Note that it is always good practice to set the transparency mode.

```
int transparency = SCREEN_TRANSPARENCY_SOURCE_OVER;
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_TRANSPARENCY, &transparency);
```

Next, set the buffer size. Since the hourglass shape is 100x100, simply set the buffer size to match those dimensions. The source rectangle will default to 100x100 once the buffer size is set, so there is no need to set it. The on-screen dimensions of the child window will also default to 100x100.

Remember that parent windows are responsible for setting the position and size of each child window. Do not set those properties here. Instead, let the event loop do that once all windows are ready to be made visible.

Memory is allocated for the buffer, then a handle to the buffer is returned by calling the *screen_get_window_property_pv()* function and specifying the SCREEN_PROPERTY_RENDER_BUFFERS constant. A pointer to the buffer is returned by calling the *screen_get_buffer_property_pv()* property and specifying the buffer handle. This pointer will be used to fill the hourglass shape.

```
int rect[4] = { 0, 0, 100, 100 };
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);

screen_buffer_t screen_buf;
screen_create_window_buffers(screen_win, 1);
screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_buf);

char *ptr = NULL;
screen_get_buffer_property_pv(screen_buf, SCREEN_PROPERTY_POINTER, (void **)&ptr);
```

Next, in the `create_hg_window()` function, draw the shape of the hourglass in the buffer.

The stride is the number of bytes between pixels on different rows. That is, if a pixel at position (x,y) is at `ptr`, the pixel at location (x,y+1) will be at `ptr+stride`. There is no guarantee that each line is 400 bytes in this case. Drivers often have constraints that will require the stride to be larger than the width in pixels times the number of bytes per pixel.

The hourglass shape is simple enough that it can be calculated. Below, the alpha channel is adjusted to be transparent or opaque based on a test that determines if a pixel is inside or outside of the hourglass shape.

```
int stride = 0;
screen_get_buffer_property_iv(screen_buf, SCREEN_PROPERTY_STRIDE, &stride);

for (i = 0; i < rect[3]; i++, ptr += stride) {
       for (j = 0; j < rect[2]; j++) {
            ptr[j*4] = 0xa0;
            ptr[j*4+1] = 0xa0;
            ptr[j*4+2] = 0xa0;
            ptr[j*4+3] = ((j >= i && j <= rect[3]-i)
               || (j <= i && j >= rect[3]-i)) ? 0xff : 0;
        }
}
```

Finally, call the *screen_post_window()* function to post the buffer. This will allow the hourglass child window to become visible when the event loop decides to make it

visible. It is customary for the first post to have a single dirty `rect` that covers the entire buffer.

```
screen_post_window(screen_win, screen_buf, 1, rect, 0);
```

Now that you've created functions to create a parent window and two child windows, you can implement the logic of the sample application. The logic, which creates the windows and sets up an event loop, is defined in the `main()` function.

Create the main() function

The main function calls the window creation functions, defines the size of the application window, and defines the event loop that controls the flow of the sample application.

Call the window functions

First, create a context of type `SCREEN_APPLICATION_CONTEXT`. The `SCREEN_APPLICATION_CONTEXT` type can be created by any process, regardless of permission level. The context sets up a connection with Screen that lets you create windows and control some of their properties. This parent window type (`SCREEN_APPLICATION_CONTEXT`) can only control a child window that was created by the same context.

```
int main(int argc, char **argv)
{
    int pos[2], size[2];
    int vis = 0;
    int type;

    screen_context_t screen_ctx;
    screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);
 ...
```

Next,you must specify the dimensions for each child window. Remember that the parent window should always determine the size and position of each child window; otherwise, the application may face potential race conditions. For example, the video mode might change during the initialization phase, or the display can be rotated. Either of these situations can lead to bad layouts. Instead, the parent should choose a layout and provide the dimensions when creating each child window.

After each window has joined the group and posted, the parent can request that each window resize if the layout changed. Once everything is perfect, each window can be made visible.

Below, each display that is associated with the context is queried. The application chooses the first display to run the sample application on.

```
int count = 0;
screen_get_context_property_iv(screen_ctx, SCREEN_PROPERTY_DISPLAY_COUNT, &count);
screen_display_t *screen_disps = calloc(count, sizeof(screen_display_t));
screen_get_context_property_pv(screen_ctx, SCREEN_PROPERTY_DISPLAYS, (void **)screen_disps);

screen_display_t screen_disp = screen_disps[0];
free(screen_disps);
```

Next, query the size of the display. The display size parameters (*dims*) are used as the dimensions for our windows and are passed into each window creation function. Note that the display size changes when it is rotated. The display size may also change when the video mode is changed.

```
int dims[2] = { 0, 0 };
screen_get_display_property_iv(screen_disp, SCREEN_PROPERTY_SIZE, dims);
```

Next, in the *main()* function, create a `String` by calling *getpid()* to return the process ID. This ID is passed into the `create_bg_window()`, `create_bar_window()`, and `create_hg_window()` functions where it is used as the window group ID. Even though process IDs are unique, there is no guarantee that this `string` will be unique. An application must check error codes for `EEXIST` and change its group name if it encounters such an error.

The *bar_id_string* and *hg_id_string* variables are passed into the child window functions. These ID strings are used to identify a window during the event loop.

```
char str[16];
snprintf(str, sizeof(str), "%d", getpid());
screen_bg_win = create_bg_window(str, rotation, dims, screen_ctx);

create_bar_window(str, bar_id_string, rotation, dims);
create_hg_window(str, hg_id_string, rotation, dims);
```

Create the event loop

The event loop defines how the application responds to and processes events. The sample application listens for window events to determine when to display a window, when to close a window, and when to carry on with normal application processing.

```
screen_event_t screen_ev;
screen_create_event(&screen_ev);
while (1) {
    do {
        ...
    }
}
```

In the event loop, a `screen_event_t` object is instantiated and used to capture an event. There is no need to repeatedly create and destroy event objects. The same event can be reused several times. You should avoid passing an event object to another thread for processing. Calling *screen_get_event()* with a 0 timeout returns immediately. The event type will be `SCREEN_EVENT_NONE` if there were no events in the queue. Calling *screen_get_event()* with a timeout of -1, or ~0 will block until an event is put into the event queue.

A handle to the event is returned by calling the *screen_get_event_property_iv()* function.

Next, an `if...else` clause is set up to process the event. The application traps the `SCREEN_EVENT_POST` event type to ensure that the window has been properly created and initialized before it is made visible. This event is sent when a child window posts for the first time, or when a child window joins our group after having successfully posted at least once. Remember that the *screen_post_window()* function was called

as the last step in the `create_hg_window()`, `create_bar_window()`, and `create_bg_window()` functions.

Once the event is trapped, the *screen_get_event_property_pv()* function is called to return a handle to the window that dispatched the event. The *screen_get_window_property_cv()* is then called to return the ID `String` of the child window. In theory, any process could choose to join our group, or do so accidentally. You can kick unwanted windows out of your group simply by calling *screen_leave_group()* on those window handles.

```
screen_get_event(screen_ctx, screen_ev, vis ? 0 : ~0);
screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &type);

if (type == SCREEN_EVENT_POST) {
    screen_window_t screen_win;
    screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&screen_win);
    screen_get_window_property_cv(screen_win, SCREEN_PROPERTY_ID_STRING, sizeof(str), str);

    if (!screen_bar_win && !strcmp(str, bar_id_string)) {
        screen_bar_win = screen_win;
    } else if (!screen_hg_win && !strcmp(str, hg_id_string)) {
        screen_hg_win = screen_win;
}
```

When all windows have been posted, the window properties are set by the parent window. Remember that all properties are relative to the parent. This includes the size, position, and z-order of each window. All these changes will be atomic, so the user won't see frames without the bar or the hourglass. The following code simply sets the screen size to fullscreen, except for the hourglass window which will be 100x100, positioned at 10,10. The z-order is set to 0 for the background, 1 for the vertical bar, and 2 for the hourglass.

```
if (screen_bar_win && screen_hg_win) {
    vis = 1;

    screen_get_window_property_iv(screen_hg_win, SCREEN_PROPERTY_BUFFER_SIZE, size);
    screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_SIZE, size);

    pos[0] = pos[1] = 10;
    screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_POSITION, pos);

    pos[0] = pos[1] = 0;
    screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_POSITION, pos);
    screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_POSITION, pos);

    size[0] = barwidth;
    size[1] = dims[1];
    screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_SIZE, size);

    size[0] = dims[0];
    screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_SIZE, size);

    int zorder = 0;
    screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_ZORDER, &zorder);

    zorder++;
    screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_ZORDER, &zorder);

    zorder++;
    screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_ZORDER, &zorder);
}
```

Finally, the child windows are set to visible by calling the *screen_set_window_property_iv()* function specifying the SCREEN_PROPERTY_VISIBLE constant. Since this function is called within the while loop, we know that all windows will appear on the screen. As usual, all the requests that were made so far have been batched in a command buffer. To ensure that those commands are flushed out and

applied, you must call *screen_flush_context()*. The SCREEN_WAIT_IDLE flag is passed in to make sure that at least one vsync or refresh period has elapsed, with the bar at 0,0 before moving it by 1 to the right.

```
screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_VISIBLE, &vis);
screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_VISIBLE, &vis);
screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_VISIBLE, &vis);
screen_flush_context(screen_ctx, SCREEN_WAIT_IDLE);
```

Next, the SCREEN_EVENT_CLOSE event is trapped in order to process window close events. When a window is closed, the handle to the window is set to NULL and the window is destroyed by calling the *screen_destroy_window()* function. The application keeps track of which window leaves the group so that it can start again if the missing window were to join the group and post.

```
else if (type == SCREEN_EVENT_CLOSE) {
    screen_window_t screen_win;
    screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&screen_win);
    if (screen_win == screen_bar_win) {
        screen_bar_win = NULL;
    } else if (screen_win == screen_hg_win) {
        screen_hg_win = NULL;
    }
    screen_destroy_window(screen_win);

    if (!screen_bar_win || !screen_hg_win) {
        vis = 0;
    }
}
```

While no close window events are trapped, the X position of the `screen_bar_win` child window is incremented by one and the *screen_set_window_property_iv()* function is called to update the position of the bar. When the bar reaches the right side of the screen, it automatically starts over from the left. Note that this sample produces an animation without actually rendering anything.

To prevent the animation from moving the bar too fast, the *screen_flush_context()* function is called with flags set to SCREEN_WAIT_IDLE. This will rate-limit the animation to the refresh rate of the display.

```
if (vis) {
    if (++pos[0] > dims[0] - barwidth) {
        pos[0] = 0;
    }
    screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_POSITION, pos);
    screen_flush_context(screen_ctx, SCREEN_WAIT_IDLE);
}
```

The windowing system has termination handlers that will release any resource created by a process when it exits, whether it exits normally or abruptly. Although any instances created are destroyed when the application exits, it is best practice to destroy any window, pixmap, and context instances that you created but no longer require.

```
screen_destroy_event(screen_ev);
screen_destroy_context(screen_ctx);
return EXIT_SUCCESS;
```

Complete sample: A vsync application using windows

The complete code sample is listed below.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <screen/screen.h>

const char *hg_id_string = "hourglass";

const char *bar_id_string = "bar";

const int barwidth = 32;

screen_window_t screen_bg_win = NULL;

screen_window_t screen_hg_win = NULL;

screen_window_t screen_bar_win = NULL;

screen_window_t create_bg_window(const char *group, int dims[2], screen_context_t screen_ctx)
{
 /* Start by creating the application window and window group. */
 screen_window_t screen_win;
 screen_create_window(&screen_win, screen_ctx);
 screen_create_window_group(screen_win, group);


 int vis = 0;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);


 int color = 0xffffff00;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_COLOR, &color);


 int rect[4] = { 0, 0, 1, 1 };
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_SIZE, dims);


 int pos[2] = { -dims[0], -dims[1] };
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_POSITION, pos);


 screen_buffer_t screen_buf;
 screen_create_window_buffers(screen_win, 1);
 screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_buf);
 screen_post_window(screen_win, screen_buf, 1, rect, 0);

 return screen_win;
}


void create_bar_window(const char *group, const char *id, int dims[2])
{

 screen_context_t screen_ctx;
 screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);


 screen_window_t screen_win;
 screen_create_window_type(&screen_win, screen_ctx, SCREEN_CHILD_WINDOW);
 screen_join_window_group(screen_win, group);
 screen_set_window_property_cv(screen_win, SCREEN_PROPERTY_ID_STRING, strlen(id), id);


 int vis = 0;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);


 int color = 0xff0000ff;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_COLOR, &color);


 int rect[4] = { 0, 0, 1, 1 };
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);


 int pos[2] = { -rect[2], -rect[3] };
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_POSITION, pos);

 screen_buffer_t screen_buf;
 screen_create_window_buffers(screen_win, 1);
 screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_buf);
 screen_post_window(screen_win, screen_buf, 1, rect, 0);
}


void create_hg_window(const char *group, const char *id, int dims[2])
{
 int i, j;


 screen_context_t screen_ctx;
 screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);

 screen_window_t screen_win;
 screen_create_window_type(&screen_win, screen_ctx, SCREEN_CHILD_WINDOW);
 screen_join_window_group(screen_win, group);
 screen_set_window_property_cv(screen_win, SCREEN_PROPERTY_ID_STRING, strlen(id), id);
```

```
 int flag = 1;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_STATIC, &flag);

 int vis = 0;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);


 int format = SCREEN_FORMAT_RGBA8888;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_FORMAT, &format);


 int usage = SCREEN_USAGE_WRITE;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &usage);


 int transparency = SCREEN_TRANSPARENCY_SOURCE_OVER;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_TRANSPARENCY, &transparency);


 int rect[4] = { 0, 0, 100, 100 };
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);


 screen_buffer_t screen_buf;
 screen_create_window_buffers(screen_win, 1);
 screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_buf);


 char *ptr = NULL;
 screen_get_buffer_property_pv(screen_buf, SCREEN_PROPERTY_POINTER, (void **)&ptr);


 int stride = 0;
 screen_get_buffer_property_iv(screen_buf, SCREEN_PROPERTY_STRIDE, &stride);


 for (i = 0; i < rect[3]; i++, ptr += stride) {
  for (j = 0; j < rect[2]; j++) {
   ptr[j*4] = 0xa0;
   ptr[j*4+1] = 0xa0;
   ptr[j*4+2] = 0xa0;
   ptr[j*4+3] = ((j >= i && j <= rect[3]-i) || (j <= i && j >= rect[3]-i)) ? 0xff : 0;
  }
 }


 screen_post_window(screen_win, screen_buf, 1, rect, 0);
}

int main(int argc, char **argv)
{
 int pos[2], size[2];
 int vis = 0;
 int type;


 screen_context_t screen_ctx;
 screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);


 int count = 0;
 screen_get_context_property_iv(screen_ctx, SCREEN_PROPERTY_DISPLAY_COUNT, &count);
 screen_display_t *screen_disps = calloc(count, sizeof(screen_display_t));
 screen_get_context_property_pv(screen_ctx, SCREEN_PROPERTY_DISPLAYS, (void **)screen_disps);

 screen_display_t screen_disp = screen_disps[0];
 free(screen_disps);


 int dims[2] = { 0, 0 };
 screen_get_display_property_iv(screen_disp, SCREEN_PROPERTY_SIZE, dims);


 char str[16];
 snprintf(str, sizeof(str), "%d", getpid());
 screen_bg_win = create_bg_window(str, dims, screen_ctx);


 create_bar_window(str, bar_id_string, dims);
 create_hg_window(str, hg_id_string, dims);


 screen_event_t screen_ev;
 screen_create_event(&screen_ev);

 while (1) {
  do {

   screen_get_event(screen_ctx, screen_ev, vis ? 0 : ~0);
   screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &type);


   if (type == SCREEN_EVENT_POST) {

    screen_window_t screen_win;
    screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&screen_win);
    screen_get_window_property_cv(screen_win, SCREEN_PROPERTY_ID_STRING, sizeof(str), str);


    if (!screen_bar_win && !strcmp(str, bar_id_string)) {
     screen_bar_win = screen_win;
    } else if (!screen_hg_win && !strcmp(str, hg_id_string)) {
     screen_hg_win = screen_win;
    }
```

```
    if (screen_bar_win && screen_hg_win) {
     vis = 1;

     screen_get_window_property_iv(screen_hg_win, SCREEN_PROPERTY_BUFFER_SIZE, size);
     screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_SIZE, size);

     pos[0] = pos[1] = 10;
     screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_POSITION, pos);

     pos[0] = pos[1] = 0;
     screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_POSITION, pos);
     screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_POSITION, pos);

     size[0] = barwidth;
     size[1] = dims[1];
     screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_SIZE, size);

     size[0] = dims[0];
     screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_SIZE, size);

     int zorder = 0;
     screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_ZORDER, &zorder);
     zorder++;
     screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_ZORDER, &zorder);
     zorder++;
     screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_ZORDER, &zorder);

     screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_VISIBLE, &vis);
     screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_VISIBLE, &vis);
     screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_VISIBLE, &vis);
     screen_flush_context(screen_ctx, SCREEN_WAIT_IDLE);
    }
   } else if (type == SCREEN_EVENT_CLOSE) {

    screen_window_t screen_win;
    screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&screen_win);

    if (screen_win == screen_bar_win) {
     screen_bar_win = NULL;
    } else if (screen_win == screen_hg_win) {
     screen_hg_win = NULL;
    }

    screen_destroy_window(screen_win);

    if (!screen_bar_win || !screen_hg_win) {
     vis = 0;
    }
   }
  } while (type != SCREEN_EVENT_NONE);

  if (vis) {
   if (++pos[0] > dims[0] - barwidth) {
    pos[0] = 0;
   }
   screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_POSITION, pos);
   screen_flush_context(screen_ctx, SCREEN_WAIT_IDLE);
  }
 }

 screen_destroy_event(screen_ev);
 screen_destroy_context(screen_ctx);

 return EXIT_SUCCESS;
}
```

Tutorial: Draw and perform vsync operations using blits, pixmaps, and buffers

This section describes how you can create the familiar hourglass and bar example using pixmaps, buffers, and blits.

The samples in this documentation demonstrate how to accomplish tasks using a variety of techniques. The result is usually a moving blue bar over a yellow background with an hourglass positioned at the top left of the application.

This sample show you how you can create such a simple application by copying pixmaps to buffers, using the `screen_blit()` function to move data among buffers, and finally making the images visible on a display.

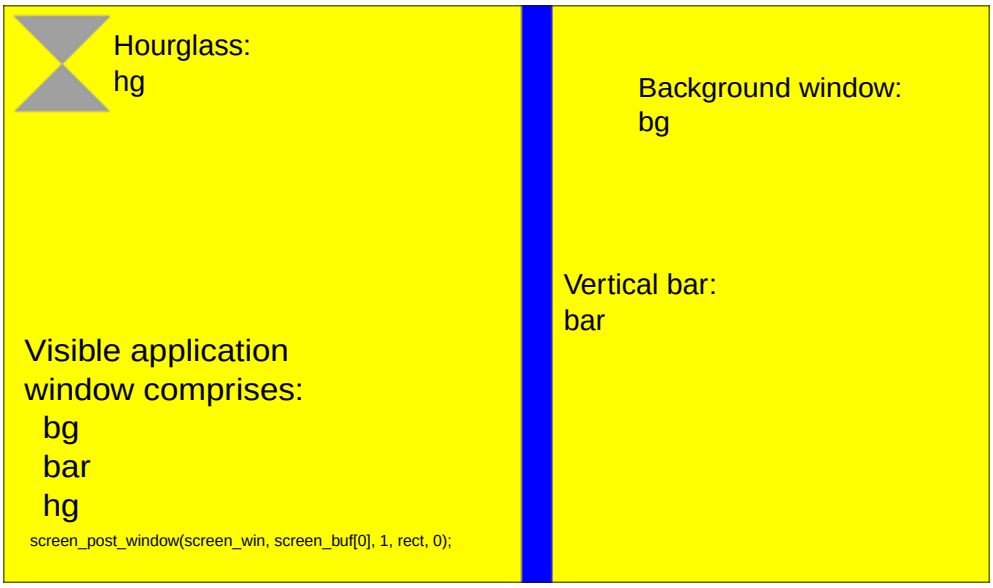


Figure 10: The result of the `blit-vsync` sample application

Create a context and initialize a window

Before you can create your application's background window, you must create a context.

Call the *screen_create_context()* function with the `SCREEN_APPLICATION_CONTEXT` flag to set up a connection with the windowing system that lets you create windows and control some of their properties. When you use the `SCREEN_APPLICATION_CONTEXT` flag, you cannot use the resulting context to control windows created by other applications.

```
screen_context_t screen_ctx;
screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);
```

Create the application window—this sample uses a single application window.

```
screen_window_t screen_win;
screen_create_window(&screen_win, screen_ctx);
```

The default buffer usage is read/write. This sample uses blits and fills, so change the usage to native. You don't need a pointer to the buffers, so there's no need to add read/write to the usage.

```
int usage = SCREEN_USAGE_NATIVE;
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &usage);
```

Create buffers and a pixmap

To let the application process a frame while the windowing system updates the frame buffer with earlier changes, the sample uses a double-buffered window.

Using a double-buffered window has the added advantage of preventing flickering if an alpha-blended window is placed on top of the application window.

The default buffer size is usually fullscreen, but you can confirm that by querying the buffer size.

```
int rect[4] = { 0, 0 };
screen_create_window_buffers(screen_win, 2);
screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);
```

The sample stores the hourglass bitmap in a pixmap letting you use the `screen_blit()` function to copy the bitmap to the window. This technique is faster than moving the pixels manually in a `for` loop or with `memcopy`.

```
screen_pixmap_t screen_pix;
screen_create_pixmap(&screen_pix, screen_ctx);
```

The sample blends the hourglass on top of the yellow background and blue bar letting the bar show through the areas of the bitmap that are not covered by the hourglass. To do this, the pixmap must have an alpha channel — RGBA8888.

```
int format = SCREEN_FORMAT_RGBA8888;
screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_FORMAT, &format);
```

Unlike the window buffers, to load the hourglass image you must use a pointer to the pixmap buffer. Later, when you copy the contents of the pixmap to the window using the `screen_blit()` function, you'll need a combined usage property of SCREEN_USAGE_WRITE and SCREEN_USAGE_NATIVE.

```
usage = SCREEN_USAGE_WRITE | SCREEN_USAGE_NATIVE;
screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_USAGE, &usage);
```

Specify the size of the hourglass — 100px x 100px.

```
int size[2] = { 100, 100 };
screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_BUFFER_SIZE, size);
```

Pixmaps can have only a single buffer, but you must use *screen_create_pixmap_buffer()* to create that buffer explicitly. This lets you change several key properties, such as usage and buffer size, before creating the pixmap buffer. When you've created the

buffer, you can get a handle to it by querying the pixmap's SCREEN_PROPERTY_RENDER_BUFFERS property.

```
screen_buffer_t screen_pbuf;
screen_create_pixmap_buffer(screen_pix);
screen_get_pixmap_property_pv(screen_pix, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_pbuf);
```

To fill in the hourglass shape, get a pointer to the pixmap buffer by querying the pointer property of the handle you obtained in the previous step. If you forgot to add read or write to the usage, this operation returns `NULL`.

```
unsigned char *ptr = NULL;
screen_get_buffer_property_pv(screen_pbuf, SCREEN_PROPERTY_POINTER, (void **)&ptr);
```

The *stride* is the number of bytes between pixels on different rows, represented by the *stride* variable. If a pixel at position `(x,y)` is at *ptr*, the pixel at location `(x, y+1)` will be at `(ptr + stride)`. Because you set write and native usage, there's no guarantee that each line is 400 bytes in this case. Drivers often have constraints that require the stride to be larger than `width * bytes` per pixel.

```
int stride = 0;
screen_get_buffer_property_iv(screen_pbuf, SCREEN_PROPERTY_STRIDE, &stride)
```

Rather than load an image, as a real application might at this stage, the hourglass is simple enough to calculate. The calculation adjusts the alpha channel to be transparent or opaque based on a test that determines whether a pixel is inside or outside the hourglass.

```
for (i = 0; i < size[1]; i++, ptr += stride) {
    for (j = 0; j < size[0]; j++) {
        ptr[j*4] = 0xa0;
        ptr[j*4+1] = 0xa0;
        ptr[j*4+2] = 0xa0;
        ptr[j*4+3] = ((j >= i && j <= size[1]-i) || (j <= i && j >= size[1]-i)) ? 0xff : 0;
    }
}
```

Combine buffers with blit functions and properties

To create the illusion of the moving blue bar, you can use blit functions and properties to combine the buffers you've created and make them visible by posting them to the application window.

The sample doesn't bother listening for events, so if you want to break the `while` loop, just press `Ctrl+C` to exit the sample or kill the process.

```
while (1) {
    screen_buffer_t screen_buf[2];
    screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)screen_buf);
```

The sample fills the buffer with the yellow background color — the format of the color argument is AARRGGBB. By default a fill operation covers the entire destination buffer.

For the background window, this is what you want to see, so there's no need to specify more arguments.

```
int bg[] = { SCREEN_BLIT_COLOR, 0xffffff00, SCREEN_BLIT_END };
screen_fill(screen_ctx, screen_buf[0], bg);
```

The vertical blue bar is also a rectangular area, so you can use the *screen_fill()* function again. This time you must specify the size, position, and color of the rectangle. The vertical bar covers the entire height of the window, but not its width.

```
int bar[] = {
SCREEN_BLIT_COLOR, 0xff0000ff,
SCREEN_BLIT_DESTINATION_X, pos,
SCREEN_BLIT_DESTINATION_WIDTH, barwidth,
SCREEN_BLIT_END };
screen_fill(screen_ctx, screen_buf[0], bar);
```

To complete the application's graphics, blend the hourglass in the pixmap with the window buffer. Specify the dimensions of a rectangle to contain the hourglass and set the transparency mode to indicate that the hourglass should blend with whatever is already in the window buffer.

```
int hg[] = {
    SCREEN_BLIT_SOURCE_WIDTH, 100,
    SCREEN_BLIT_SOURCE_HEIGHT, 100,
    SCREEN_BLIT_DESTINATION_X, 10,
    SCREEN_BLIT_DESTINATION_Y, 10,
    SCREEN_BLIT_DESTINATION_WIDTH, 100,
    SCREEN_BLIT_DESTINATION_HEIGHT, 100,
    SCREEN_BLIT_TRANSPARENCY, SCREEN_TRANSPARENCY_SOURCE_OVER,
    SCREEN_BLIT_END
};
```

Calling *screen_post_window()* will:

1. Draw the background, the hourglass, and the blue bar onto the window buffer.
2. Make the background, the hourglass and the blue bar visible on the display.
3. Signal the windowing system to redraw the screen.

This function also flushes the blits, so it's not necessary to flush the blits before calling a post operation.

```
screen_blit(screen_ctx, screen_buf[0], screen_pbuf, hg);
screen_post_window(screen_win, screen_buf[0], 1, rect, 0);
```

Finally, to make the blue bar appear to move from left to right across the background, increment its position by one after each frame, and then wrap the position back to the origin of the buffer before the bar begins to move off the right-hand edge of the screen.

```
if (++pos > rect[2] - barwidth) {
    pos = 0;
    }
}
```

Complete sample: A vsync application using blits, pixmaps, and buffers

This is the complete listing for the blit, pixpmap, and buffers sample.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <screen/screen.h>

const int barwidth = 32;

int main(int argc, char **argv)
{
 int i, j, pos = 0;

 screen_context_t screen_ctx;
 screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);

 screen_window_t screen_win;
 screen_create_window(&screen_win, screen_ctx);

 int usage = SCREEN_USAGE_NATIVE;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &usage);

 int rect[4] = { 0, 0 };
 screen_create_window_buffers(screen_win, 2);
 screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);

 screen_pixmap_t screen_pix;
 screen_create_pixmap(&screen_pix, screen_ctx);

 int format = SCREEN_FORMAT_RGBA8888;
 screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_FORMAT, &format);

 usage = SCREEN_USAGE_WRITE | SCREEN_USAGE_NATIVE;
 screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_USAGE, &usage);

 int size[2] = { 100, 100 };
 screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_BUFFER_SIZE, size);

 screen_buffer_t screen_pbuf;
 screen_create_pixmap_buffer(screen_pix);
 screen_get_pixmap_property_pv(screen_pix, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_pbuf);

 unsigned char *ptr = NULL;
 screen_get_buffer_property_pv(screen_pbuf, SCREEN_PROPERTY_POINTER, (void **)&ptr);

 int stride = 0;
 screen_get_buffer_property_iv(screen_pbuf, SCREEN_PROPERTY_STRIDE, &stride);

 for (i = 0; i < size[1]; i++, ptr += stride) {
   for (j = 0; j < size[0]; j++) {
    ptr[j*4] = 0xa0;
    ptr[j*4+1] = 0xa0;
    ptr[j*4+2] = 0xa0;
    ptr[j*4+3] = ((j >= i && j <= size[1]-i) || (j <= i && j >= size[1]-i)) ? 0xff : 0;
   }
 }

 while (1) {
  screen_buffer_t screen_buf[2];
  screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)screen_buf);

  int bg[] = { SCREEN_BLIT_COLOR, 0xffffff00, SCREEN_BLIT_END };
  screen_fill(screen_ctx, screen_buf[0], bg);

  int bar[] = {
   SCREEN_BLIT_COLOR, 0xff0000ff,
   SCREEN_BLIT_DESTINATION_X, pos,
   SCREEN_BLIT_DESTINATION_WIDTH, barwidth,
   SCREEN_BLIT_END };

  screen_fill(screen_ctx, screen_buf[0], bar);

  int hg[] = {
   SCREEN_BLIT_SOURCE_WIDTH, 100,
   SCREEN_BLIT_SOURCE_HEIGHT, 100,
   SCREEN_BLIT_DESTINATION_X, 10,
   SCREEN_BLIT_DESTINATION_Y, 10,
   SCREEN_BLIT_DESTINATION_WIDTH, 100,
   SCREEN_BLIT_DESTINATION_HEIGHT, 100,
   SCREEN_BLIT_TRANSPARENCY, SCREEN_TRANSPARENCY_SOURCE_OVER,
   SCREEN_BLIT_END
  };

  screen_blit(screen_ctx, screen_buf[0], screen_pbuf, hg);
  screen_post_window(screen_win, screen_buf[0], 1, rect, 0);

  if (++pos > rect[2] - barwidth) {
   pos = 0;
  }
 }

 return EXIT_SUCCESS;
}
```

Tutorial: Write an application using OpenGL ES

This simple sample application uses a native window to create an EGL on-screen rendering surface. This surface is the target of the OpenGL ES 1.X rendering.

This sample application uses the components of a grey hourglass, a moving blue vertical bar, and a yellow background. It aims to demonstrate how to integrate the use of OpenGL ES 1.X and Screen in one application.

Figure 11: OpenGL Application

You will learn to:

- establish a connection to and initialize the display
- choose an appropriate EGL configuration
- create an OpenGL ES rendering context
- create a native context
- create a native window
- set the appropriate properties for your native window
- create an EGL on-screen rendering surface
- create a main application loop to:
  - process events in the native context
  - render using OpenGL ES 1.X
- release resources

Use OpenGL ES in a windowed vsync application

The following walkthrough takes you through the process of writing a native application that uses OpenGL ES for the rendering API.

1. Create the variables you'll need for your application.

   Some important variables are:

```
/* a connection to screen windowing system */
screen_context_t screen_ctx;
```

```
/* a native handle for our window */
screen_window_t screen_win;
/* a handle used to pop events from our queue */
screen_event_t screen_ev;
/* the EGL swap interval */
EGLint interval = 1
/* the EGL configuration string */
const char *conf_str = NULL;
/* the array where you'll store the vertices */
GLshort points[20];
/* the abstract display on which graphics are drawn */
EGLDisplay egl_disp;
/* the configuration describing the color and ancillary buffers */
EGLConfig egl_conf;
/* your window's rendering surface */
EGLSurface egl_surf;
/* a handle to a rendering context */
EGLContext egl_ctx;
/* the resulting EGL config */
EGLConfig egl_conf = (EGLConfig)0;
/* describes the color and ancillary buffers */
EGLConfig *egl_configs;
/* number of configs that match our attributes */
EGLint egl_num_configs;
/* an EGL integer value */
EGLint val;
/* the return value of EGL functions */
EGLBoolean eglrc;
```

You'll use the surface attributes to choose between single-buffered and double-buffered rendering. To avoid having to keep track of indexes in a one-dimensional array of attribute/value pairs, you can use an aggregate of named attribute/value pairs of type EGLint (an integer of 32 bits).

```
struct {
    EGLint render_buffer[2];
    EGLint none;
} egl_surf_attr = {
    /* double-buffering */
    .render_buffer = { EGL_RENDER_BUFFER, EGL_BACK_BUFFER },
    /* End of list */
    .none = EGL_NONE
};
```

Here is the list of attributes that will be passed to EGL to get you a pixel format configuration. An EGL configuration is required by EGL when creating surfaces and rendering contexts. Since you will modify certain values in this list when certain command-line arguments are provided, you will organize the attributes as an aggregate of named key/value pairs of EGL integers (*EGLint*). This way you won't have to track the index locations of the various attributes in a one-dimensional array.

```
struct {
   EGLint surface_type;
   EGLint red_size;
   EGLint green_size;
   EGLint blue_size;
   EGLint alpha_size;
   EGLint samples;
   EGLint config_id;
} egl_conf_attr = {
    /* Ask for displayable and pbuffer surfaces */
   .surface_type = EGL_WINDOW_BIT,
    /* Minimum number of red bits per pixel */
   .red_size = EGL_DONT_CARE,
    /* Minimum number of green bits per pixel */
   .green_size = EGL_DONT_CARE,
    /* Minimum number of blue bits per pixel */
   .blue_size = EGL_DONT_CARE,
   /* Minimum number of alpha bits per pixel */
   .alpha_size = EGL_DONT_CARE,
   /* Minimum number of samples per pixel */
   .samples = EGL_DONT_CARE,
   /* Used to get a specific EGL config */
   .config_id = EGL_DONT_CARE,
};
```

2. Process the command-line arguments, if any.

   In this sample application, command-line arguments are accepted so that the user may specify some configuration values. The valid options include:

   - `-single-buffer` (rendering done to a single on-screen buffer)
   - `-double-buffer` (rendering done on alternating back buffers)
   - `-interval=`*int* (swap interval)
   - `-config=`*string* (comma-separated list of EGL configuration specifiers)
   - `-size=`*width*x*height* (size of the viewport)
   - `-pos=`*x*, *y* (position of the viewport)
   - `-verbose` (displays EGL configuration used by application)

   If no options are indicated as command-line arguments, this application will assume the following defaults:

Option	Default
Number of buffers used for rendering	2
Swap interval	1
EGL configuration	RGB pixel format with the smallest depth supported by the hardware
Viewport size	fullscreen
Verbose	off

```
for (i = 1; i < argc; i++) {
   if (strncmp(argv[i], "-config=", strlen("-config=")) == 0) {
      /** EGL configuration **/
      conf_str = argv[i] + strlen("-config=");
   } else if (strncmp(argv[i], "-size=", strlen("-size=")) == 0) {
       /** size of viewport **/
       tok = argv[i] + strlen("-size=");
       size[0] = atoi(tok);
       while (*tok >= '0' && *tok <= '9') {
          tok++;
       }
       size[1] = atoi(tok+1);
   } else if (strncmp(argv[i], "-pos=", strlen("-pos=")) == 0) {
       /** position of viewport**/
       tok = argv[i] + strlen("-pos=");
       pos[0] = atoi(tok);
       while (*tok >= '0' && *tok <= '9') {
          tok++;
       }
       pos[1] = atoi(tok+1);
    } else if (strncmp(argv[i], "-interval=", strlen("-interval=")) == 0) {
        /** swap interval **/
        interval = atoi(argv[i] + strlen("-interval="));
    } else if (strcmp(argv[i], "-single-buffer") == 0) {
        /** single-buffer rendering **/
        nbuffers = 1;
    } else if (strcmp(argv[i], "-double-buffer") == 0) {
        /** double-buffer rendering **/
        nbuffers = 2;
    } else if (strncmp(argv[i], "-verbose", strlen("-verbose")) == 0) {
        /** verbose option selected **/
        verbose = EGL_TRUE;
    } else {
        /** unsupported option **/
        fprintf(stderr, "Invalid command-line option: %s\n", argv[i]);
    }
}
```

3. Establish a connection to the EGL display.

   Before you can do any kind of rendering, you must establish a connection to a display.

   In this sample application, you will use the default display.

   ```
   egl_disp = eglGetDisplay(EGL_DEFAULT_DISPLAY);
   ```

4. Initialize the EGL display.

   You will be able to do little with the EGL display until it's been initialized. The second and third arguments of *eglInitialize()* are both set to NULL because OpenGL ES 1.X is supported by all versions of EGL; therefore it isn't necessary to check for the major and minor version numbers.

   ```
   rc = eglInitialize(egl_disp, NULL, NULL);
   ```

5. Choose an EGL configuration.

   Choosing an appropriate EGL config is an important part of the initialization procedure. This is especially true in embedded systems where the difference in performance between pixel formats can make or break an application.

   On desktop systems, a pixel format of RGB111 or better is usually sufficient because this pixel format returns the best configuration supported by the hardware.

   On embedded systems, RGBA8888 may not be an option. Even if RGBA8888 is supported by the rendering hardware, the system may not be able to handle the memory bandwidth required by the display controller to paint the display at 60 frames per second.

   Specifying an EGL configuration by its ID (`EGL_CONFIG_ID`) gives you the ability to get exactly what you want. However, this approach is far from being user-friendly when there are multiple platforms and windowing systems to consider.

   In this sample application the user can choose an appropriate EGL config by specifiying one of the following:

   - a pixel format
   - a pixel format and a number of per-pixel samples
   - an EGL configuration ID

   a. Parse the configuration string from the command-line argument

      You establish your EGL configuration attributes from the command-line arguments.

      ```
      if (str != NULL) {
          tok = str;
          while (*tok == ' ' || *tok == ',') {
             tok++;
           }
          while (*tok != '\0') {
              if (strncmp(tok, "rgba8888", strlen("rgba8888")) == 0) {
                  egl_conf_attr.red_size = 8;
                  egl_conf_attr.green_size = 8;
      ```

```
                egl_conf_attr.blue_size = 8;
                egl_conf_attr.alpha_size = 8;
                tok += strlen("rgba8888");
            } else if (strncmp(tok, "rgba5551", strlen("rgba5551")) == 0) {
                egl_conf_attr.red_size = 5;
                egl_conf_attr.green_size = 5;
                egl_conf_attr.blue_size = 5;
                egl_conf_attr.alpha_size = 1;
                tok += strlen("rgba5551");
            } else if (strncmp(tok, "rgba4444", strlen("rgba4444")) == 0) {
                egl_conf_attr.red_size = 4;
                egl_conf_attr.green_size = 4;
                egl_conf_attr.blue_size = 4;
                egl_conf_attr.alpha_size = 4;
                tok += strlen("rgba4444");
            } else if (strncmp(tok, "rgb565", strlen("rgb565")) == 0) {
                egl_conf_attr.red_size = 5;
                egl_conf_attr.green_size = 6;
                egl_conf_attr.blue_size = 5;
                egl_conf_attr.alpha_size = 0;
                tok += strlen("rgb565");
            } else if (isdigit(*tok)) {
                val = atoi(tok);
                while (isdigit(*(++tok)));
                if (*tok == 'x') {
                    egl_conf_attr.samples = val;
                    tok++;
                } else {
                    egl_conf_attr.config_id = val;
                }
            } else {
                fprintf(stderr, "Invalid configuration specifier: ");
                while (*tok != ' ' && *tok != ',' && *tok != '\0') {
                    fputc(*tok++, stderr);
                }
                fputc('\n', stderr);
            }
            /**
             ** Skip any spaces and separators between this token and the next one.
             **/
            while (*tok == ' ' || *tok == ',') {
               tok++;
            }
        }
    }
```

b. Use *eglGetConfigs()* to find an EGL conguration that matches attributes specified at the command line.

Here, *eglGetConfigs()* is used instead of *eglChooseConfigs()*, which is probably the most complicated function of EGL. There are many attributes that can be specified, each with its own matching rules, default value, and sorting order. It's easy to get confused with all the special rules ending up with the wrong configuration, or no configuration, without understanding why. So instead of using *eglChooseConfigs()*, you will use *eglGetConfigs()* to get all the EGL configurations and search for one that matches your specified attributes.

```
rc = eglGetConfigs(egl_disp, NULL, 0, &egl_num_configs);
```

c. Allocate sufficient memory to hold all possible matching configurations.

The total number of EGL configurations is stored in *egl_num_configs* after you've called the *eglGetConfigs()* function. The number is used to calculate how much memory needs to be allocated. You need enough memory to hold all the configurations so that you can traverse through the configurations to find a match.

```
egl_configs = malloc(egl_num_configs * sizeof(*egl_configs));
```

d. Call *eglGetConfigs()* a second time to store the configurations in the recently allocated memory.

The list of EGL configurations is expected to be static. Therefore, your list of configurations for the purpose of matching should be static as well. As long as the call to *eglGetConfigs()* succeeds, it isn't necessary to check for *egl_num_configs* again.

```
rc = eglGetConfigs(egl_disp, egl_configs, egl_num_configs, &egl_num_configs);
```

e. Go through the list of EGL configurations to find one that has all the attributes specified on the command line.

Some attributes such as surface type or the renderable type are masks, but all others are integers that you need to compare.

```
for (i = 0; i < egl_num_configs; i++) {
    /* EGL Configuration ID */
    if (egl_conf_attr.config_id != EGL_DONT_CARE) {
        eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_CONFIG_ID, &val);
        if (val == egl_conf_attr.config_id) {
            egl_conf = egl_configs[i];
            break;
        } else {
            continue;
        }
    }
    /* Surface Type */
    eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_SURFACE_TYPE, &val);
    if ((val & egl_conf_attr.surface_type) != egl_conf_attr.surface_type) {
        continue;
    }
    / ** Renderable type has the OpenGL ES bit. */
    eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_RENDERABLE_TYPE, &val);
    if (!(val & EGL_OPENGL_ES_BIT)) {
        continue;
    }
    /* Red Bits */
    if (egl_conf_attr.red_size != EGL_DONT_CARE) {
        eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_RED_SIZE, &val);
        if (val != egl_conf_attr.red_size) {
            continue;
        }
    }
    /* Green Bits */
    if (egl_conf_attr.green_size != EGL_DONT_CARE) {
        eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_GREEN_SIZE, &val);
        if (val != egl_conf_attr.green_size) {
            continue;
        }
    }
    /* Blue Bits */
    if (egl_conf_attr.blue_size != EGL_DONT_CARE) {
        eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_BLUE_SIZE, &val);
        if (val != egl_conf_attr.blue_size) {
            continue;
        }
    }
    /* Alpha Bits */
    if (egl_conf_attr.alpha_size != EGL_DONT_CARE) {
        eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_ALPHA_SIZE, &val);
        if (val != egl_conf_attr.alpha_size) {
            continue;
        }
    }
    /* Number of Samples */
    if (egl_conf_attr.samples != EGL_DONT_CARE) {
        eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_SAMPLES, &val);
        if (val != egl_conf_attr.samples) {
            continue;
        }
    }
    /* This config has the pixel format we asked for, so we can keep it
       and stop looking.  */
     egl_conf = egl_configs[i];
     break;
}
```

f. Free the array that you allocated for the purpose of finding a matching EGL configuration.

```
free(egl_configs);
```

6. Create an OpenGL ES rendering context.

Now, create an OpenGL ES rendering context. Among other things, this context keeps track of the OpenGL ES state. You don't need to specify the current rendering API with the *eglBindApi()* function because OpenGL ES is the default rendering API.

The third argument to *eglCreateContext()* is another EGL rendering context with which you wish to share data. Pass EGL_NO_CONTEXT to indicate that you won't need any of the textures or vertex buffer objects created in another EGL rendering context.

The last argument to *eglCreateContext()* is an attribute list that you can use to specify an API version number. You would use it to override the `EGL_CONTEXT_CLIENT_VERSION` value from `1` to `2` if you were writing an OpenGL ES 2.X application.

```
egl_ctx = eglCreateContext(egl_disp, egl_conf, EGL_NO_CONTEXT, NULL);
```

7. Create your native context.

```
rc = screen_create_context(&screen_ctx, 0);
```

8. Create your native window.

```
rc = screen_create_window(&screen_win, screen_ctx);
```

9. Set your native window properties based on the command-line arguments or defaults.

```
EGLint buffer_bit_depth, alpha_bit_depth;
eglGetConfigAttrib(egl_disp, egl_conf, EGL_BUFFER_SIZE, &buffer_bit_depth);
eglGetConfigAttrib(egl_disp, egl_conf, EGL_ALPHA_SIZE, &alpha_bit_depth);
switch (buffer_bit_depth) {
    case 32: {
        return SCREEN_FORMAT_RGBA8888;
    }
    case 24: {
        return SCREEN_FORMAT_RGB888;
    }
    case 16: {
        switch (alpha_bit_depth) {
            case 4: {
                return SCREEN_FORMAT_RGBA4444;
            }
            case 1: {
                return SCREEN_FORMAT_RGBA5551;
            }
            default: {
                return SCREEN_FORMAT_RGB565;
            }
        }
        break;
    }
    default: {
        return SCREEN_FORMAT_BYTE;
    }
}
rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_FORMAT, &format);
rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &usage);
```

```
rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SWAP_INTERVAL,
&interval);
if (size[0] > 0 && size[1] > 0) {
    rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SIZE, size);
} else {
    rc = screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_SIZE, size);
}
if (pos[0] != 0 || pos[1] != 0) {
    rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_POSITION, pos);
}
```

10. Create your window buffers for rendering.

```
rc = screen_create_window_buffers(screen_win, nbuffers);
```

11. Create a Screen event so that the application can listen for and react to relevant events from the windowing system.

```
rc = screen_create_event(&screen_ev);
```

12. Create the EGL on-screen rendering surface.

    Now that you've created a native platform window, you can use it to create an EGL on-screen rendering surface. You'll be able to use this surface as the target of your OpenGL ES rendering. You'll use the same EGL display and EGL configuration to create the EGL surface as you used to set the properties on your native window. The EGL configuration needs to be compatible with the one used to create the window.

```
egl_surf = eglCreateWindowSurface(egl_disp, egl_conf, screen_win,
(EGLint*)&egl_surf_attr);
```

13. Bind the EGL context to the current rendering thread and to a draw-and-read surface.

    In this application, you want to draw to the EGL surface and not really care about where you read from. Since EGL doesn't allow specifying EGL_NO_SURFACE for only the read surface, you will use *egl_surf* for both drawing and reading. Once *eglMakeCurrent()* completes successfully, all OpenGL ES calls will be executed on the context and the surface you provided as arguments.

```
rc = eglMakeCurrent(egl_disp, egl_surf, egl_surf, egl_ctx);
```

14. Set the EGL swap interval.

    The *eglSwapInterval()* function specifies the minimum number of video frame periods per buffer swap for the window associated with the current context. So, if the interval is 0, the application renders as fast as it can. Interval values of 1 or more limit the rendering to fractions of the display's refresh rate. (For example, 60, 30, 20, 15, etc. frames per second in the case of a display with a refresh rate of 60 Hz.)

```
rc = eglSwapInterval(egl_disp, interval);
```

15. Initialize the viewport, the geometry, and the color for your application.

This application is fairly simple, so you just need to initialize the OpenGL ES viewport and projection matrix. You need to compute the positions of vertices that will be used to do the rendering. The positions of these vertices are based on the window's dimensions instead of using scale and translation matrices. It's unlikely that the size will change often, so this method is more efficient than applying transformations every time a frame is rendered.

```
/**
 ** The first four vertices take up 8 shorts. These vertices define a
 ** rectangle that goes from (0,0) to (barwidth,height). A translation
 ** matrix will be used to slide this rectangle across the viewport.
 **/
points[0] = 0;
points[1] = height;
points[2] = barwidth;
points[3] = height;
points[4] = 0;
points[5] = 0;
points[6] = barwidth;
points[7] = 0;
/**
 ** The last six vertices take up 12 shorts. These vertices define two
 ** triangles that share a vertex. Because the OpenGL ES co-ordinate system
 ** starts at the bottom left instead of the top left corner, all y values
 ** need to be inverted. In other words, the hourglass needs to be
 ** translated up and down as the window height increases and decreases
 ** respectively.
 **/
points[8] = 10;
points[9] = height - 10;
points[10] = 110;
points[11] = height - 10;
points[12] = 60;
points[13] = height - 60;
points[14] = 60;
points[15] = height - 60;
points[16] = 110;
points[17] = height - 110;
points[18] = 10;
points[19] = height - 110;
/* Update the viewport and projection matrix */
glViewport(0, 0, width, height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrthof(0.0f, (GLfloat)width, 0.0f, (GLfloat)height, -1.0f, 1.0f);
glMatrixMode(GL_MODELVIEW);
/* Set the clear color to yellow for the background*/
glClearColor(1.0f, 1.0f, 0.0f, 1.0f);
/* You will use one vertex array for all of the rendering */
glVertexPointer(2, GL_SHORT, 0, (const GLvoid *)points);
glEnableClientState(GL_VERTEX_ARRAY);
```

16. Set up the main application loop to handle events and to perform the rendering.

    The main application loop runs continuously until either an error occurs or you receive a SCREEN_EVENT_CLOSE event from the windowing system.

    The main application loop consists of two main functions:

    - processing of relevant events
    - performing the rendering

```
while (1) {
...
    /* Part 1: Process events */
    while (!screen_get_event(screen_ctx, screen_ev, vis ? 0 : ~0)){...}

    /* Part 2: Render if window is visible */
    if (vis && !pause) {...}
}
```

    a. Process relevant events.

The first part of the main applcation loop processes any events that might be put on your context's queue. The only events that are of interest to you are the resize and close events. The timeout variable is set to 0 (no wait) or forever depending on whether the window is visible or not.

```
while (!screen_get_event(screen_ctx, screen_ev, vis ? 0 : ~0)) {
    rc = screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &val);
    if (rc || val == SCREEN_EVENT_NONE) {
  break;
    }
 switch (val) {
       case SCREEN_EVENT_CLOSE:
          /**
           ** All you have to do when you receive the close event is
           ** exit the application loop. Because there is a loop
           ** within a loop, a simple break won't work - just use a goto
           ** to get out.
           **/
          goto end;
       case SCREEN_EVENT_PROPERTY:
          /**
           ** You are interested in visibility changes so you can pause
           ** or unpause the rendering.
           **/
          screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_NAME, &val);
          switch (val) {
              case SCREEN_PROPERTY_VISIBLE:
                /**
                 ** The visibility status is not included in the
                 ** event, so you need to call screen_get_window_property_iv()
                 ** to retrieve the value.
                 **/
                screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE,
 &vis);
                 break;
          }
          break;
       case SCREEN_EVENT_POINTER:
          /**
           ** To provide a way of gracefully terminating your application,
           ** exit if there is a pointer select event in the upper
           ** right corner of your window. This should happen if the mouse's
           ** left button is clicked or if a touch screen display is pressed.
           ** The event will come as a screen pointer event, with an (x,y)
           ** coordinate relative to the window's upper left corner and a
           ** select value. You have to verify ourselves that the co-ordinates
           ** of the pointer are in the upper right hand area.
           **/
          screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_BUTTONS, &val);
          if (val) {
              screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_POSITION,
pos);
              if (pos[0] >= size[0] - exit_area_size &&
                  pos[1] < exit_area_size) {
                      goto end;
              }
          }
          break;
       case SCREEN_EVENT_KEYBOARD:
         screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_FLAGS, &val);

          if (val & KEY_DOWN) {
              screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_SYM,
&val);
              switch (val) {
                  case KEYCODE_ESCAPE:
                      goto end;
                  case KEYCODE_F:
                      pause = !pause;
                      break;
                  default:
                      break;
              }
           }
           break;
 }
}
```

a. Perform the rendering.

The second part of the main application loop is the rendering. You want to skip the rendering part if your window isn't visible, so as to leave the CPU and GPU

to other applications. This approach will enable the system to be more responsive while the window is invisible.

```
if (vis && !pause) {
    /* Start by clearing the window */
    glClear(GL_COLOR_BUFFER_BIT);
    /** You could use glLoadIdentity or glPushMatrix here. If you use
     ** glLoadIdentity, you would need to call glLoadIdentity again when
     ** you draw the hourglass. The assumption is that glPushMatrix,
     ** glTranslatef, glPopMatrix are more efficient than
     ** glLoadIdentity, glTranslatef, and glLoadIdentity.
     **/
    glPushMatrix();
    /* Use translation to animate the vertical bar */
    glTranslatef((GLfloat)(i++ % (size[0] - barwidth)), 0.0f, 0.0f);
    /* Make the animated vertical bar to be drawn in blue */
    glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
    /* Render the vertical bar */
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    /* You don't want the hourglass to be translated */
    glPopMatrix();
    /* Make the hourglass to be drawn in gray */
    glColor4f(0.62745f, 0.62745f, 0.62745f, 1.0f);
    /* Render the hourglass */
    glDrawArrays(GL_TRIANGLES, 4, 6);
    /**
     ** Posting of the new frame requires a call to eglSwapBuffers.
     ** For now, this is true even when using single buffering. If an
     ** event has occured that invalidates the surface we are currently
     ** using, eglSwapBuffers will return EGL_FALSE and set the error
     ** code to EGL_BAD_NATIVE_WINDOW. At this point, you could destroy
     ** the EGL surface, close the native window, and start again.
     ** This application will simply exit when any errors occur.
     **/
    rc = eglSwapBuffers(egl_disp, egl_surf);
}
```

17. Release resources.

Before you can destroy any of the resources you've created for this application, you must deactivate the rendering context used and release the surfaces from where you were drawing and reading.

This deactivation and release is done by calling *eglMakeCurrent()* with EGL_NO_SURFACE and EGL_NO_CONTEXT as arguments. Note that the call to *eglMakeCurrent()* will generate an error unless all arguments are EGL_NO_SURFACE and EGL_NO_CONTEXT, or all arguments are valid EGLSurface and EGLContext objects.

You will also need to terminate the connection to the EGL display and release any resources that were allocated for this thread. On most systems, these resources would likely be released automatically when the program exits, but it's good practice to do so by calling *eglReleaseThread()*.

```
eglMakeCurrent(egl_disp, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);
eglDestroySurface(egl_disp, egl_surf);
screen_destroy_event(screen_ev);        /* Destroy the Screen event */
screen_destroy_window(screen_win);      /* Destroy the native window */
screen_destroy_context(screen_ctx);     /* Destroy the Screen context */
eglDestroyContext(egl_disp, egl_ctx);   /* Destroy the EGL render context */
eglTerminate(egl_disp);                 /* Terminate connection to display */
eglReleaseThread();                     /* Release resources for thread */
```

Complete sample: A windowed vsync application using OpenGL ES

This code sample uses Screen Graphics Subsystem with OpenGL ES 1.X as the rendering API.

This sample application uses the components of a grey hourglass, a moving blue vertical bar, and a yellow background. It aims to demonstrate how to integrate the use of OpenGL ES 1.X and Screen in one application.

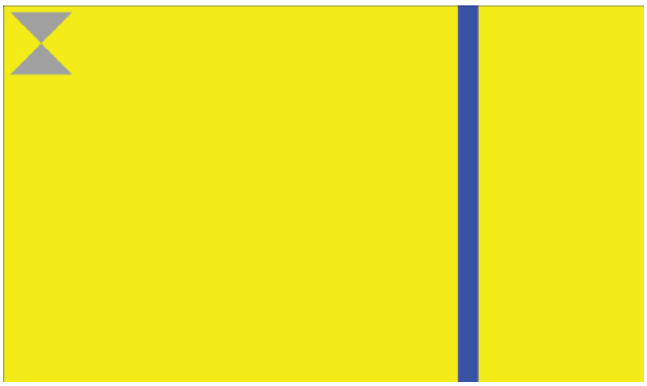


Figure 12: OpenGL Application

```
/**
 ** gles1-vsync
 ** Windowed vsync that uses OpenGL ES 1.X for rendering API.
 **
 ** Features:
 **  - configurable window size through the -size=(width)x(height) option
 **  - configurable window position through the -pos=(x),(y) option
 **  - adjustable swap interval through the -interval=(n) option;
 **    a swap interval of 0 lets the app run as fast as possible
 **    numbers of 1 or more limit the rate to the number of vsync periods
 **  - application responds to size changes from window manager or delegate
 **  - rendering is suspended if window is not visible
 **
 ** Copyright 2010, QNX Software Systems Ltd. All Rights Reserved
 **
 ** Permission to use, copy, modify, and distribute this software and its
 ** documentation for any purpose and without fee is hereby granted,
 ** provided that the above copyright notice appear in all copies and that
 ** both that copyright notice and this permission notice appear in
 ** supporting documentation.
 **
 ** This file is provided AS IS with no warranties of any kind.  The author
 ** shall have no liability with respect to the infringement of copyrights,
 ** trade secrets or any patents by this file or any part thereof.  In no
 ** event will the author be liable for any lost revenue or profits or
 ** other special, indirect and consequential damages.
 */

/**
 ** Include the header files for libraries we are using.
 **/

#include <ctype.h>         /* Header file for isdigit */
#include <stdio.h>         /* Header file for fprintf */
#include <stdlib.h>        /* Header file for EXIT_FAILURE, EXIT_SUCCESS, atoi */
#include <string.h>        /* Header file for strncmp */
#include <sys/keycodes.h>  /* Header file for KEYCODE_ESCAPE */
#include <screen/screen.h> /* Header file for the native screen API */
#include <EGL/egl.h>       /* Header file for EGL */
#include <GLES/gl.h>       /* Header file for OpenGL ES 1.X */

/**
 ** Here is the list of attributes that will be passed to EGL to get us
 ** a pixel format configuration. An EGL configuration is required by
 ** EGL when creating surfaces and rendering contexts. Since we will
 ** modify certain values in this list when certain command line arguments
 ** are provided, we will organize our attributes as an aggregate of named
 ** key/value pairs of EGLint's. This way we won't have to track the
 ** index locations various attributes in a one-dimensional array.
 **/
```

```
struct {
 EGLint surface_type;
 EGLint red_size;
 EGLint green_size;
 EGLint blue_size;
 EGLint alpha_size;
 EGLint samples;
 EGLint config_id;
} egl_conf_attr = {
 .surface_type = EGL_WINDOW_BIT,   /* Ask for displayable and pbuffer surfaces */
 .red_size = EGL_DONT_CARE,        /* Minimum number of red bits per pixel */
 .green_size = EGL_DONT_CARE,      /* Minimum number of green bits per pixel */
 .blue_size = EGL_DONT_CARE,       /* Minimum number of blue bits per pixel */
 .alpha_size = EGL_DONT_CARE,      /* Minimum number of alpha bits per pixel */
 .samples = EGL_DONT_CARE,         /* Minimum number of samples per pixel */
 .config_id = EGL_DONT_CARE,       /* used to get a specific EGL config */
};

/* This function will convert EGL error codes into more meaningful messages. */
static void
egl_perror(const char *msg) {
    static const char *errmsg[] = {
        "function succeeded",
        "EGL is not initialized, or could not be initialized, for the specified display",
        "cannot access a requested resource",
        "failed to allocate resources for the requested operation",
        "an unrecognized attribute or attribute value was passed in an attribute list",
        "an EGLConfig argument does not name a valid EGLConfig",
        "an EGLContext argument does not name a valid EGLContext",
        "the current surface of the calling thread is no longer valid",
        "an EGLDisplay argument does not name a valid EGLDisplay",
        "arguments are inconsistent",
        "an EGLNativePixmapType argument does not refer to a valid native pixmap",
        "an EGLNativeWindowType argument does not refer to a valid native window",
        "one or more argument values are invalid",
        "an EGLSurface argument does not name a valid surface configured for rendering",
        "a power management event has occurred",
    };

    fprintf(stderr, "%s: %s\n", msg, errmsg[eglGetError() - EGL_SUCCESS]);
}

/* This function will parse the configuration string and/or select an appropriate
 * configuration based on configuration attributes.*/
EGLConfig choose_config(EGLDisplay egl_disp, const char* str)
{
 EGLConfig egl_conf = (EGLConfig)0;  /* the resulting EGL config */
 EGLConfig *egl_configs;             /* describes the color and ancillary buffers */
 EGLint egl_num_configs;             /* number of configs that match our attributes */
 EGLint val;                         /* an EGL integer value */
 EGLBoolean rc;                      /* the return value of EGL functions */
 const char *tok;                    /* a pointer that will traverse the string */
 EGLint i;                           /* variable used to loop on matching configs */

 /**
  ** We start by parsing the config string, which is a comma-separated list
  ** of specifiers. We don't have to use strtok because the syntax is quite
  ** simple. All we will need is strncmp and atoi. We start by skipping any
  ** whitespace or separators. The str argument might be null, indicating
  ** that we must find a config that matches the default criteria. In this
  ** case, we must skip any processing of the str and make sure that there
  ** is a default value for all configuration attributes (usually
  ** EGL_DONT_CARE.)
  **/

 if (str != NULL) {
  tok = str;
  while (*tok == ' ' || *tok == ',') {
   tok++;
  }

  /**
   ** Loop as long as there are tokens to be processed.
   **/

  while (*tok != '\0') {
   if (strncmp(tok, "rgba8888", strlen("rgba8888")) == 0) {
    egl_conf_attr.red_size = 8;
    egl_conf_attr.green_size = 8;
    egl_conf_attr.blue_size = 8;
    egl_conf_attr.alpha_size = 8;
    tok += strlen("rgba8888");
   } else if (strncmp(tok, "rgba5551", strlen("rgba5551")) == 0) {
    egl_conf_attr.red_size = 5;
    egl_conf_attr.green_size = 5;
    egl_conf_attr.blue_size = 5;
    egl_conf_attr.alpha_size = 1;
    tok += strlen("rgba5551");
   } else if (strncmp(tok, "rgba4444", strlen("rgba4444")) == 0) {
    egl_conf_attr.red_size = 4;
    egl_conf_attr.green_size = 4;
```

```
    egl_conf_attr.blue_size = 4;
    egl_conf_attr.alpha_size = 4;
    tok += strlen("rgba4444");
   } else if (strncmp(tok, "rgb565", strlen("rgb565")) == 0) {
    egl_conf_attr.red_size = 5;
    egl_conf_attr.green_size = 6;
    egl_conf_attr.blue_size = 5;
    egl_conf_attr.alpha_size = 0;
    tok += strlen("rgb565");
   } else if (isdigit(*tok)) {
    /**
     ** An integer value could either be an EGL configuration id or
     ** a multi-sampling count. An 'x' immediately after the integer
     ** indicates that it is a multi-sampling specifier.
     **/
    val = atoi(tok);
    while (isdigit(*(++tok)));
    if (*tok == 'x') {
     egl_conf_attr.samples = val;
     tok++;
    } else {
     /**
      ** Using the EGL_CONFIG_ID attribute allows us to get a
      ** configuration by its id without a typecast,
      ** i.e. egl_conf = (EGLConfig)val. Note that the EGL spec says
      ** that when the EGL_CONFIG_ID attribute is specified, all
      ** other attributes are ignored, so we don't have to shorten
      ** the list ourselves in this case.
      **/
     egl_conf_attr.config_id = val;
    }
   } else {
    /**
     ** Print a message on the console if we encounter something we
     ** don't know. This way, the user will know that he might not get
     ** the config he was expecting.
     **/
    fprintf(stderr, "Invalid configuration specifier: ");
    while (*tok != ' ' && *tok != ',' && *tok != '\0') {
     fputc(*tok++, stderr);
    }
    fputc('\n', stderr);
   }

   /**
    ** Skip any spaces and separators between this token and the next one.
    **/

   while (*tok == ' ' || *tok == ',') {
    tok++;
   }
  }
}

/* Use eglGetConfigs to retrieve EGL configurations */

rc = eglGetConfigs(egl_disp, NULL, 0, &egl_num_configs);
if (rc != EGL_TRUE) {
 egl_perror("eglGetConfigs");
 return egl_conf;
}
if (egl_num_configs == 0) {
 fprintf(stderr, "eglGetConfigs: could not find a configuration\n");
 return egl_conf;
}

/**
 ** Now the total number of configs has been stored in egl_num_configs.
 ** We will malloc enough memory to hold all the configs. We need this to
 ** traverse the configs to find a perfect match.
 **/

egl_configs = malloc(egl_num_configs * sizeof(*egl_configs));
if (egl_configs == NULL) {
 fprintf(stderr, "could not allocate memory for %d EGL configs\n", egl_num_configs);
 return egl_conf;
}

/**
 ** The second time we call eglGetConfigs, it is with an array of configs
 ** big enough to store all the results. The list of EGL configs is
 ** expected to be static, which means our list of matching configs should
 ** be static as well. As long as the call succeeds, we don't have to check
 ** for egl_num_configs again.
 **/

rc = eglGetConfigs(egl_disp, egl_configs,
 egl_num_configs, &egl_num_configs);
if (rc != EGL_TRUE) {
 egl_perror("eglGetConfigs");
 free(egl_configs);
```

```
 return egl_conf;
}

/**
 ** Now we just have to go through the list of configs and find one that
 ** has all the attributes we are looking for. Some attributes like the
 ** surface type or the renderable type are masks. All the others are just
 ** integers that we need to match unless we don't care about the value.
 **/

for (i = 0; i < egl_num_configs; i++) {
 /**
  ** Make sure the config id matches what we asked for.
  **/

 if (egl_conf_attr.config_id != EGL_DONT_CARE) {
  eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_CONFIG_ID, &val);
  if (val == egl_conf_attr.config_id) {
   egl_conf = egl_configs[i];
   break;
  } else {
   continue;
  }
 }

 /**
  ** Make sure the surface type matches what we asked for.
  **/

 eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_SURFACE_TYPE, &val);
 if ((val & egl_conf_attr.surface_type) != egl_conf_attr.surface_type) {
  continue;
 }

 /**
  ** Make sure the renderable type has the OpenGL ES bit.
  **/

 eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_RENDERABLE_TYPE, &val);
 if (!(val & EGL_OPENGL_ES_BIT)) {
  continue;
 }

 /**
  ** Make sure the number of red bits matches what we asked for.
  **/

 if (egl_conf_attr.red_size != EGL_DONT_CARE) {
  eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_RED_SIZE, &val);
  if (val != egl_conf_attr.red_size) {
   continue;
  }
 }

 /**
  ** Make sure the number of green bits matches what we asked for.
  **/

 if (egl_conf_attr.green_size != EGL_DONT_CARE) {
  eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_GREEN_SIZE, &val);
  if (val != egl_conf_attr.green_size) {
   continue;
  }
 }

 /**
  ** Make sure the number of blue bits matches what we asked for.
  **/

 if (egl_conf_attr.blue_size != EGL_DONT_CARE) {
  eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_BLUE_SIZE, &val);
  if (val != egl_conf_attr.blue_size) {
   continue;
  }
 }

 /**
  ** Make sure the number of alpha bits matches what we asked for.
  **/

 if (egl_conf_attr.alpha_size != EGL_DONT_CARE) {
  eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_ALPHA_SIZE, &val);
  if (val != egl_conf_attr.alpha_size) {
   continue;
  }
 }

 /**
  ** Make sure the number of samples matches what we asked for.
  **/
```

```
  if (egl_conf_attr.samples != EGL_DONT_CARE) {
   eglGetConfigAttrib(egl_disp, egl_configs[i], EGL_SAMPLES, &val);
   if (val != egl_conf_attr.samples) {
    continue;
   }
  }

  /**
   ** This config has the pixel format we asked for, so we can keep it
   ** and stop looking.
   **/

  egl_conf = egl_configs[i];
  break;
 }

 /**
  ** At this point, it is important to remember to free the array allocated
  ** to hold all configs before returning.
  **/

 free(egl_configs);

 if (egl_conf == (EGLConfig)0) {
  /**
   ** The value of egl_conf will be that of a matching config if one was
   ** found or (EGLConfig)0 if there were no exact matches. The calling
   ** function can decide to do what it wants with the result. For example,
   ** it could ask for a different config if no matches were found, or it can
   ** simply fail and report the error.
   **/

  fprintf(stderr, "eglChooseConfig: could not find a matching configuration\n");
 }

 return egl_conf;
}

int choose_format(EGLDisplay egl_disp, EGLConfig egl_conf)
{
 EGLint buffer_bit_depth, alpha_bit_depth;

 eglGetConfigAttrib(egl_disp, egl_conf, EGL_BUFFER_SIZE, &buffer_bit_depth);
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_ALPHA_SIZE, &alpha_bit_depth);
 switch (buffer_bit_depth) {
  case 32: {
   return SCREEN_FORMAT_RGBA8888;
  }
  case 24: {
   return SCREEN_FORMAT_RGB888;
  }
  case 16: {
   switch (alpha_bit_depth) {
    case 4: {
     return SCREEN_FORMAT_RGBA4444;
    }
    case 1: {
     return SCREEN_FORMAT_RGBA5551;
    }
    default: {
     return SCREEN_FORMAT_RGB565;
    }
   }
   break;
  }
  default: {
   return SCREEN_FORMAT_BYTE;
  }
 }
}

/**
 ** The function is used to initialize the OpenGL ES viewport and
 ** projection matrix. It also computes the position of vertices that will be
 ** used to do rendering. We compute the position of those vertices based on
 ** the window's dimensions instead of using scale and translation matrices.
 ** Because it is unlikely that the size will change very often, this is more
 ** efficient than applying transformations every time a frame is rendered.
 **/

static void resize(GLshort *points, GLint width, GLint height, GLint barwidth)
{
 /**
  ** The first four vertices take up 8 shorts. These vertices define a
  ** rectangle that goes from (0,0) to (barwidth,height). A translation
  ** matrix will be used to slide this rectangle across the viewport.
  **/

 points[0] = 0;
 points[1] = height;
 points[2] = barwidth;
```

```
    points[3] = height;
    points[4] = 0;
    points[5] = 0;
    points[6] = barwidth;
    points[7] = 0;

    /**
     ** The last six vertices take up 12 shorts. These vertices define two
     ** triangles that share a vertex. Because the OpenGL ES coordinate system
     ** starts at the bottom left instead of the top left corner, all y values
     ** need to be inverted. In other words, the hourglass needs to be
     ** translated up and down as the window height increases and decreases
     ** respectively.
     **/

    points[8] = 10;
    points[9] = height - 10;
    points[10] = 110;
    points[11] = height - 10;
    points[12] = 60;
    points[13] = height - 60;
    points[14] = 60;
    points[15] = height - 60;
    points[16] = 110;
    points[17] = height - 110;
    points[18] = 10;
    points[19] = height - 110;

    /* Update the viewport and projection matrix */
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrthof(0.0f, (GLfloat)width, 0.0f, (GLfloat)height, -1.0f, 1.0f);
    glMatrixMode(GL_MODELVIEW);
}

/**
 ** This is the entry point of our native application. This is where we will
 ** parse command line arguments, create our window, handle events, and do our
 ** rendering.
 **/

int main(int argc, char **argv)
{
    /**
     ** This is the size for an invisible exit button. We choose a value that's
     ** big enough to be useable with touchscreens and pointer devices.
     **/

    const int exit_area_size = 20;

    /**
     ** Make the sliding vertical blue bar 32 pixels wide. Any number would be
     ** fine here, as long as it is no larger than the width of the window.
     **/

    const int barwidth = 32;

    /**
     ** We will use the surface attributes to choose between single-buffered
     ** and double-buffered rendering. Again, to avoid having to keep track of
     ** indexes in a one-dimensional array of attribute/value pairs, we use
     ** an aggregate of named attribute/value pairs of EGLint's.
     **/

    struct {
        EGLint render_buffer[2];
        EGLint none;
    } egl_surf_attr = {
        .render_buffer = { EGL_RENDER_BUFFER, EGL_BACK_BUFFER }, /* Ask for double-buffering */
        .none = EGL_NONE                                         /* End of list */
    };

    screen_context_t screen_ctx;         /* connection to screen windowing system */
    screen_window_t screen_win;          /* native handle for our window */
    screen_event_t screen_ev;            /* handle used to pop events from our queue */
    EGLDisplay egl_disp;                 /* abstract display on which graphics are drawn */
    EGLConfig egl_conf;                  /* describes the color and ancillary buffers */
    EGLSurface egl_surf;                 /* refers to our window's rendering surface */
    EGLContext egl_ctx;                  /* a handle to a rendering context */
    int usage = SCREEN_USAGE_OPENGL_ES1; /* we will use OpenGL ES 1.X to do our rendering */
    int size[2] = { -1, -1 };            /* width and height of our window */
    int pos[2] = { 0, 0 };               /* x,y position of our window */
    int nbuffers = 2;                    /* number of buffers backing the window */
    int format;                          /* native visual type / screen format */
    int val;                             /* a generic variable used to set/get window properties */
    EGLint interval = 1;                 /* EGL swap interval */
    int verbose = EGL_FALSE;             /* EGL_TRUE if the verbose option was set */
    int vis = 1;                         /* boolean that indicates if our window is visible */
    int pause = 0;                       /* EGL_TRUE if rendering is frozen */
    const char *conf_str = NULL;         /* configuration string */
```

```
const char *tok;                    /* used to process command-line arguments */
int rval = EXIT_FAILURE;            /* application exits with value stored here */
int rc;                             /* return value from functions */
int i;                              /* loop/frame counter */
GLshort points[20];                 /* we'll store the vertices in this array */

/**
 ** We start by processing the command line arguments. The first argument
 ** is skipped because it contains the name of the program. Arguments
 ** follow the syntax -(option)=(value).
 **/

for (i = 1; i < argc; i++) {
 if (strncmp(argv[i], "-config=", strlen("-config=")) == 0) {
  /**
   ** The syntax of the EGL configuration option is
   ** -config=[option][,[option]...]. All we need is to do is pass
   ** the string after the '=' to choose_config, which will parse the
   ** options and find the right EGL config.
   **/

  conf_str = argv[i] + strlen("-config=");
 } else if (strncmp(argv[i], "-size=", strlen("-size=")) == 0) {
  /**
   ** The syntax of the size option is -size=(width)x(height).
   ** Because atoi stops processing at the first non-digit character,
   ** we can simply call atoi on the string after the '=' to get the
   ** width, and call atoi again on the string after the 'x' to get
   ** the height.
   **/

  tok = argv[i] + strlen("-size=");
  size[0] = atoi(tok);
  while (*tok >= '0' && *tok <= '9') {
   tok++;
  }
  size[1] = atoi(tok+1);
 } else if (strncmp(argv[i], "-pos=", strlen("-pos=")) == 0) {
  /**
   ** The syntax of the pos option is -pos=(x),(y).
   ** Because atoi stops processing at the first non-digit character,
   ** we can simply call atoi on the string after the '=' to get the
   ** x offset, and call atoi again on the string after the ',' to
   ** get the y offset.
   **/

  tok = argv[i] + strlen("-pos=");
  pos[0] = atoi(tok);
  while (*tok >= '0' && *tok <= '9') {
   tok++;
  }
  pos[1] = atoi(tok+1);
 } else if (strncmp(argv[i], "-interval=", strlen("-interval=")) == 0) {
  /**
   ** The syntax of the interval option is -interval=(number). All
   ** we need is to convert the number that starts after the '='.
   **/

  interval = atoi(argv[i] + strlen("-interval="));
 } else if (strcmp(argv[i], "-single-buffer") == 0) {
  /**
   ** The -single-buffer option on the command line will cause the
   ** rendering to be done to a single on-screen buffer. There are
   ** typically artifacts associated with single-buffered rendering
   ** caused by rendering to a visible surface.
   **/

  nbuffers = 1;
 } else if (strcmp(argv[i], "-double-buffer") == 0) {
  /**
   ** The -double-buffer option on the command line will cause the
   ** rendering to be on alternating back buffers. This eliminates
   ** artifacts associated with single-buffered rendering.
   **/

  nbuffers = 2;
 } else if (strncmp(argv[i], "-verbose", strlen("-verbose")) == 0) {
  /**
   ** The verbose option has no special syntax. It just has to be
   ** present on the command line to cause the verbose messages to
   ** be printed.
   **/

  verbose = EGL_TRUE;
 } else {
  /**
   ** Make sure we say something instead of silently ignoring a
   ** command line option.
   **/

  fprintf(stderr, "Invalid command-line option: %s\n", argv[i]);
```

```
 }
}

/**
 ** Before we can do any kind of rendering, we must establish a connection
 ** to a display. We don't have any particular preference, so we'll just
 ** ask for the default display, unless a display id is specified on the
 ** command line.
 **/

egl_disp = eglGetDisplay(EGL_DEFAULT_DISPLAY);
if (egl_disp == EGL_NO_DISPLAY) {
 egl_perror("eglGetDisplay");
 goto fail1;
}

/**
 ** Now we initialize EGL on the display. We can't do anything with this
 ** EGL display until EGL has been initialized. OpenGL ES 1.X is supported
 ** by all versions of EGL, so it is not necessary to check for the major
 ** and minor version numbers.
 **/

rc = eglInitialize(egl_disp, NULL, NULL);
if (rc != EGL_TRUE) {
 egl_perror("eglInitialize");
 goto fail2;
}

/**
 ** Choosing an EGL configuration can be a tedious process. Here, we call
 ** choose_config which will do all the necessary work. In this case, this
 ** includes parsing a configuration string and/or select an appropriate
 ** configuration based on some configuration attributes.
 **/

egl_conf = choose_config(egl_disp, conf_str);
if (egl_conf == (EGLConfig)0) {
 goto fail2;
}

/**
 ** Now we need to create an OpenGL ES rendering context. The context will
 ** keep track of the OpenGL ES 1.X state among other things. We don't have
 ** to specify the current rendering API with an eglBindApi call because
 ** OpenGL ES is the default rendering API. The third argument to
 ** eglCreateContext is another EGL rendering context that we wish to share
 ** data with. We pass EGL_NO_CONTEXT to indicate that we won't need any
 ** of the textures or vertex buffer objects created in another EGL render
 ** context. The last argument is an attribute list that can be used to
 ** specify an API version number. We would use it to override the
 ** EGL CONTEXT CLIENT VERSION default value of 1 to 2 if we were writing
 ** an OpenGL ES 2.X application.
 **/

egl_ctx = eglCreateContext(egl_disp, egl_conf, EGL_NO_CONTEXT, NULL);
if (egl_ctx == EGL_NO_CONTEXT) {
 egl_perror("eglCreateContext");
 goto fail2;
}

/**
 ** If the application was started with the -verbose command line argument,
 ** we will print a few information strings about the EGL configuration we
 ** end-up using. This might be useful if the -config option was not
 ** provided, or if the user doesn't know the particular details of a given
 ** pixel format. We will get the information by using the
 ** eglGetConfigAttrib with several interesting attribute names.
 **/

if (verbose) {
 printf("EGL_VENDOR = %s\n", eglQueryString(egl_disp, EGL_VENDOR));
 printf("EGL_VERSION = %s\n", eglQueryString(egl_disp, EGL_VERSION));
 printf("EGL_CLIENT_APIS = %s\n", eglQueryString(egl_disp, EGL_CLIENT_APIS));
 printf("EGL_EXTENSIONS = %s\n\n", eglQueryString(egl_disp, EGL_EXTENSIONS));

 i = -1;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_CONFIG_ID, &i);
 printf("EGL_CONFIG_ID = %d\n", i);

 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_RED_SIZE, &i);
 printf("EGL_RED_SIZE = %d\n", i);

 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_GREEN_SIZE, &i);
 printf("EGL_GREEN_SIZE = %d\n", i);

 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_BLUE_SIZE, &i);
 printf("EGL_BLUE_SIZE = %d\n", i);
```

```
 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_ALPHA_SIZE, &i);
 printf("EGL_ALPHA_SIZE = %d\n", i);

 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_DEPTH_SIZE, &i);
 printf("EGL_DEPTH_SIZE = %d\n", i);

 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_LEVEL, &i);
 printf("EGL_LEVEL = %d\n", i);

 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_NATIVE_RENDERABLE, &i);
 printf("EGL_NATIVE_RENDERABLE = %s\n", i ? "EGL_TRUE" : "EGL_FALSE");

 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_NATIVE_VISUAL_TYPE, &i);
 printf("EGL_NATIVE_VISUAL_TYPE = %d\n", i);

 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_RENDERABLE_TYPE, &i);
 printf("EGL_RENDERABLE_TYPE = 0x%04x\n", i);

 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_SURFACE_TYPE, &i);
 printf("EGL_SURFACE_TYPE = 0x%04x\n", i);

 i = 0;
 eglGetConfigAttrib(egl_disp, egl_conf, EGL_TRANSPARENT_TYPE, &i);
 if (i == EGL_TRANSPARENT_RGB) {
  printf("EGL_TRANSPARENT_TYPE = EGL_TRANSPARENT_RGB\n");

  i = 0;
  eglGetConfigAttrib(egl_disp, egl_conf, EGL_TRANSPARENT_RED_VALUE, &i);
  printf("EGL_TRANSPARENT_RED = 0x%02x\n", i);

  i = 0;
  eglGetConfigAttrib(egl_disp, egl_conf, EGL_TRANSPARENT_GREEN_VALUE, &i);
  printf("EGL_TRANSPARENT_GREEN = 0x%02x\n", i);

  i = 0;
  eglGetConfigAttrib(egl_disp, egl_conf, EGL_TRANSPARENT_BLUE_VALUE, &i);
  printf("EGL_TRANSPARENT_BLUE = 0x%02x\n\n", i);
 } else {
  printf("EGL_TRANSPARENT_TYPE = EGL_NONE\n\n");
 }
}

rc = screen_create_context(&screen_ctx, 0);
if (rc) {
 perror("screen_context_create");
 goto fail3;
}

rc = screen_create_window(&screen_win, screen_ctx);
if (rc) {
 perror("screen_create_window");
 goto fail4;
}

format = choose_format(egl_disp, egl_conf);
rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_FORMAT, &format);
if (rc) {
 perror("screen_set_window_property_iv(SCREEN_PROPERTY_FORMAT)");
 goto fail5;
}

rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &usage);
if (rc) {
 perror("screen_set_window_property_iv(SCREEN_PROPERTY_USAGE)");
 goto fail5;
}

rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SWAP_INTERVAL, &interval);
if (rc) {
 perror("screen_set_window_property_iv(SCREEN_PROPERTY_SWAP_INTERVAL)");
 goto fail5;
}

if (size[0] > 0 && size[1] > 0) {
 rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SIZE, size);
 if (rc) {
  perror("screen_set_window_property_iv(SCREEN_PROPERTY_SIZE)");
  goto fail5;
 }
} else {
 rc = screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_SIZE, size);
 if (rc) {
  perror("screen_get_window_property_iv(SCREEN_PROPERTY_SIZE)");
```

```
  goto fail5;
 }
}

if (pos[0] != 0 || pos[1] != 0) {
 rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_POSITION, pos);
 if (rc) {
  perror("screen_set_window_property_iv(SCREEN_PROPERTY_POSITION)");
  goto fail5;
 }
}

rc = screen_create_window_buffers(screen_win, nbuffers);
if (rc) {
 perror("screen_create_window_buffers");
 goto fail5;
}

rc = screen_create_event(&screen_ev);
if (rc) {
 perror("screen_create_event");
 goto fail5;
}

/**
 ** Now that we have created a native platform window we can use it to
 ** create an EGL on-screen rendering surface. We'll be able to use this
 ** surface as the target of our OpenGL ES 1.X rendering. We'll use the
 ** same EGL display and config to create the EGL surface as the ones we
 ** used to create our native window. The EGL config just needs to be
 ** compatible with the one used to create the window.
 **/

egl_surf = eglCreateWindowSurface(egl_disp, egl_conf,
 screen_win, (EGLint*)&egl_surf_attr);
if (egl_surf == EGL_NO_SURFACE) {
 egl_perror("eglCreateWindowSurface");
 goto fail6;
}

/**
 ** eglMakeCurrent binds ctx to the current rendering thread and to a draw
 ** and a read surface. In our case, we want to draw to our EGL surface and
 ** don't really care about where we read from. EGL does not allow
 ** specifying EGL_NO_SURFACE for the read surface only, so we will simply
 ** use egl_surf for both reading and writing. Once eglMakeCurrent
 ** completes successfully, all OpenGL ES 1.X calls will be executed on
 ** the context and surface provided as arguments.
 **/

rc = eglMakeCurrent(egl_disp, egl_surf, egl_surf, egl_ctx);
if (rc != EGL_TRUE) {
 egl_perror("eglMakeCurrent");
 goto fail7;
}

/**
 ** The eglSwapInterval function specifies the minimum number of video
 ** frame periods per buffer swap for the window associated with the
 ** current context. If the interval is 0, the application renders as
 ** fast as it can. Interval values of 1 or more limit the rendering
 ** to fractions of the display's refresh rate, i.e. 60, 30, 20, 15, etc
 ** fps in the case of a display with a refresh rate of 60 Hz.
 **/

rc = eglSwapInterval(egl_disp, interval);
if (rc != EGL_TRUE) {
 egl_perror("eglSwapInterval");
 goto fail8;
}

/**
 ** At this point, we can start doing OpenGL ES stuff. Our application is
 ** quite simple, so we'll just do the initialization here followed by
 ** some basic rendering in our application loop.
 **/

if (verbose) {
 printf("GL_VENDOR = %s\n", (char *)glGetString(GL_VENDOR));
 printf("GL_RENDERER = %s\n", (char *)glGetString(GL_RENDERER));
 printf("GL_VERSION = %s\n", (char *)glGetString(GL_VERSION));
 printf("GL_EXTENSIONS = %s\n", (char *)glGetString(GL_EXTENSIONS));
}

/* The resize function initializes the viewport and geometry */
resize(points, size[0], size[1], barwidth);

/* Set the clear color to yellow */
glClearColor(1.0f, 1.0f, 0.0f, 1.0f);

/* We will use one vertex array for all of our rendering */
```

```
glVertexPointer(2, GL_SHORT, 0, (const GLvoid *)points);
glEnableClientState(GL_VERTEX_ARRAY);

/**
 ** This is our main application loop. It keeps on running unless an error
 ** occurs or we receive a close event from the windowing system. The
 ** application loop consists of two parts. The first part processes any
 ** events that have been put in our queue. The second part does the
 ** rendering. When the window is visible, we don't wait if the event queue
 ** is empty and move on to the rendering part immediately. When the window
 ** is not visible we skip the rendering part.
 **/

i = 0;
while (1) {
 /**
  ** We start the loop by processing any events that might be in our
  ** queue. The only event that is of interest to us are the resize
  ** and close events. The timeout variable is set to 0 (no wait) or
  ** forever depending if the window is visible or invisible.
  **/

 while (!screen_get_event(screen_ctx, screen_ev, vis ? 0 : ~0)) {
  rc = screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &val);
  if (rc || val == SCREEN_EVENT_NONE) {
   break;
  }
  switch (val) {
   case SCREEN_EVENT_CLOSE:
    /**
     ** All we have to do when we receive the close event is
     ** exit the application loop. Because we have a loop
     ** within a loop, a simple break won't work. We'll just
     ** use a goto to take us out of here.
     **/
    goto end;
   case SCREEN_EVENT_PROPERTY:
    /**
     ** We are interested in visibility changes so we can pause
     ** or unpause the rendering.
     **/

    screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_NAME, &val);
    switch (val) {
     case SCREEN_PROPERTY_VISIBLE:
      /**
       ** The new visibility status is not included in the
       ** event, so we must get it ourselves.
       **/
      screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);
      break;
    }
    break;
   case SCREEN_EVENT_POINTER:
    /**
     ** To provide a way of gracefully terminating our application,
     ** we will exit if there is a pointer select event in the upper
     ** right corner of our window. This should happen if the mouse's
     ** left button is clicked or if a touch screen display is pressed.
     ** The event will come as a screen pointer event, with an (x,y)
     ** coordinate relative to the window's upper left corner and a
     ** select value. We have to verify ourselves that the coordinates
     ** of the pointer are in the upper right hand area.
     **/
    screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_BUTTONS, &val);
    if (val) {
     screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_POSITION, pos);
     if (pos[0] >= size[0] - exit_area_size &&
      pos[1] < exit_area_size) {
      goto end;
     }
    }
    break;
   case SCREEN_EVENT_KEYBOARD:
    screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_FLAGS, &val);
    if (val & KEY_DOWN) {
     screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_SYM, &val);
     switch (val) {
      case KEYCODE_ESCAPE:
       goto end;
      case KEYCODE_F:
       pause = !pause;
       break;
      default:
       break;
     }
    }
    break;
  }
 }
```

```
    /**
     ** The second part of the application loop is the rendering. We want
     ** to skip the rendering part if our window is not visible. This will
     ** leave the CPU and GPU to other applications and make the system a
     ** little bit more responsive while we are invisible.
     **/

    if (vis && !pause) {
     /* Start by clearing the window */
     glClear(GL_COLOR_BUFFER_BIT);

     /**
      ** We could use glLoadIdentity or glPushMatrix here. If we used
      ** glLoadIdentity, we would have to call glLoadIdentity again when
      ** we draw the hourglass. We assume that glPushMatrix,
      ** glTranslatef, glPopMatrix is more efficient than
      ** glLoadIdentity, glTranslatef, and glLoadIdentity.
      **/
     glPushMatrix();

     /* Use translation to animate the vertical bar */
     glTranslatef((GLfloat)(i++ % (size[0] - barwidth)), 0.0f, 0.0f);

     /* We want the animated vertical bar to be drawn in blue */
     glColor4f(0.0f, 0.0f, 1.0f, 1.0f);

     /* Render the vertical bar */
     glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

     /* We don't want the hourglass to be translated */
     glPopMatrix();

     /* We want the hourglass to be drawn in gray */
     glColor4f(0.62745f, 0.62745f, 0.62745f, 1.0f);

     /* Render the hourglass */
     glDrawArrays(GL_TRIANGLES, 4, 6);

     /**
      ** Posting of the new frame requires a call to eglSwapBuffers.
      ** For now, this is true even when using single buffering. If an
      ** event has occured that invalidates the surface we are currently
      ** using, eglSwapBuffers will return EGL_FALSE and set the error
      ** code to EGL_BAD_NATIVE_WINDOW. At this point, we could destroy
      ** our EGL surface, close our OpenKODE window, and start again.
      ** This application will simply exit when any errors occur.
      **/

     rc = eglSwapBuffers(egl_disp, egl_surf);
     if (rc != EGL_TRUE) {
      egl_perror("eglSwapBuffers");
      break;
     }
    }
   }

   /**
    ** If we made it here, it means everything ran successfully. We'll thus
    ** change the exit return value to indicate success.
    **/

  end:
   rval = EXIT_SUCCESS;

   /**
    ** Before we can destroy any of the resources we have created for this
    ** application, we must deactivate the rendering context that we were
    ** using and release the surfaces we were drawing to and reading from.
    ** This is done by calling eglMakeCurrent with EGL_NO_SURFACE and
    ** EGL_NO_CONTEXT for arguments. Note that the call to eglMakeCurrent
    ** will generate an error unless all arguments are EGL_NO_SURFACE and
    ** EGL_NO_CONTEXT, or all arguments are valid EGLSurface and EGLContext
    ** objects.
    **/

  fail8:
   eglMakeCurrent(egl_disp, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);

   /**
    ** Destroy the EGL surface if one was created.
    **/

  fail7:
   eglDestroySurface(egl_disp, egl_surf);

  fail6:
   screen_destroy_event(screen_ev);
  fail5:
   screen_destroy_window(screen_win);
  fail4:
   screen_destroy_context(screen_ctx);
```

```
 /**
  ** Destroy the EGL render context if one was created.
  **/

fail3:
 eglDestroyContext(egl_disp, egl_ctx);

 /**
  ** Terminate our connection to the EGL display. Since we are just about to
  ** exit, we can also release any resources that were allocated for this
  ** thread. This is done by calling eglReleaseThread. On most systems,
  ** those resources would probably be released automatically when the
  ** program exists.
  **/

fail2:
 eglTerminate(egl_disp);
 eglReleaseThread();

 /**
  ** Return with EXIT_SUCCESS or EXIT_FAILURE.
  **/

fail1:
 return rval;
}
```

Tutorial: Screenshots

Screen screenshots are pixels read from a source and then copied into a buffer. You can then manipulate the buffer as required; it can be simply written to a file or used in other windows or displays.

The Screen API reads pixels from the source and copies them into a provided buffer to capture the screenshot. The buffer can be either a pixmap or a window buffer, but must have the usage flag of type SCREEN_USAGE_NATIVE set. The choice of whether to use a pixmap buffer or a window buffer depends on the application of the screenshot after it is taken. For example, you may choose to use a pixmap buffer for your screenshot if you need to capture an image to be used in a different window or on a different display.

Window screenshot

The *screen_read_window()* function captures a screenshot of the window. There are no contraints on the context for this function call, but you must have used either the *screen_create_window()* function or the *screen_create_window_type()* function to create the window that's the target of this screenshot. When capturing screenshots of multiple unrelated windows, you will need to make a *screen_read_window()* function call per window.

Display screenshot

The *screen_read_display()* function captures a screenshot of the display. You will need to be working within a privileged context so that you have full access to the display properties of the system. You can create a privileged context by calling the function *screen_create_context()* with a context type of SCREEN_DISPLAY_MANAGER_CONTEXT. Your process must have an effective user ID of root to be able to create this context type. When capturing screenshots of multiple displays, you will need to make one *screen_read_display()* function call per display.

Capture a window screenshot

The following procedure describes how to use Screen to capture a screenshot of a single window. You can store the resulting screenshot in either a pixmap or window buffer for further manipulation. This particular procedure describes capturing the screenshot in a pixmap buffer and then writing the screenshot to a bitmap.

This sample application uses the components of a grey hourglass, a moving blue vertical bar, and a yellow background. It aims to demonstrate how to capture a screenshot using the Screen API.

Figure 13: Screenshot Application

You will learn to:

- create a pixmap and buffer to store your screenshot
- retrieve appropriate pixmap properties to prepare for screenshot
- take your screenshot
- write your screenshot to a bitmap file

Before you begin

Before proceeding, you are expected to have already created a Screen context and the window that will be the target of your screenshot.

In the following procedure, the created context will be referred to as *screenshot_ctx*.

The targeted window will be referred to as *screenshot_win*.

1. Create variables for the pixmap, the pixmap buffer, the pixmap buffer pointer, and the stride:

```
screen_pixmap_t screen_pix;
screen_buffer_t screenshot_buf;
char *screenshot_ptr = NULL;
int screenshot_stride = 0;
```

2. Create other variables necessary to support the Screen API calls and the writing of our screenshot to a bitmap.

   In this procedure, you will declare several integer variables to help in setting the pixmap and its properties. You will also need variables associated with the writing of the screenshot to bitmap. For this example, a set path and filename are used. Ensure that you have appropriate permissions to access the directory path of the file.

```
char header[54];
char *fname = "/accounts/1000/appdata/com.example.Tutorial_WindowApp."
              "testDev_l_WindowApp85f8001_/data/hourglass_window_screenshot.bmp";
int nbytes;
int fd;
int i;
int usage, format;
int size[2];
```

3. Create the pixmap for the screenshot and set the usage flag and format properties:

```
screen_create_pixmap(&screen_pix, screenshot_ctx);

usage = SCREEN_USAGE_READ | SCREEN_USAGE_NATIVE;
screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_USAGE, &usage);

format = SCREEN_FORMAT_RGBA8888;
```

```
screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_FORMAT, &format);
```

4. Set the buffer size of the pixmap for the screenshot:

   Set an appropriate buffer size for the pixmap. The pixmap buffer size doesn't have to necessarily match the size of the source. Scaling will be applied to make the screenshot fit into the buffer provided.

   ```
   size[0] = 200;
   size[1] = 200;
   screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_BUFFER_SIZE, size);
   ```

5. Create the pixmap buffer for the screenshot and get the buffer handle, the pointer, and the stride.

   Memory is allocated for your pixmap buffer; this is the buffer where the pixels from the source window will be copied to:

   ```
   screen_create_pixmap_buffer(screen_pix);
   screen_get_pixmap_property_pv(screen_pix, SCREEN_PROPERTY_RENDER_BUFFERS,
                                 (void**)&screenshot_buf);
   screen_get_buffer_property_pv(screenshot_buf, SCREEN_PROPERTY_POINTER,
                                 (void**)&screentshot_ptr);
   screen_get_buffer_property_iv(screenshot_buf, SCREEN_PROPERTY_STRIDE,
                                 &screenshot_stride);
   ```

6. Take the window screenshot:

   ```
   screen_read_window(screenshot_win, screenshot_buf, 0, NULL ,0);
   ```

   This function takes five arguments: the target of the screenshot, the pixmap buffer, the number of rectangles defining the area of capture, the array of integers representing rectangles of the area of capture, and the mutex flag. The arguments related to the area of capture are 0 and NULL because in this example you are capturing the target area in its entirety rather than a specific rectangular area. The last argument (which represents the mutex flag) should always be 0.

7. Create the bitmap file with appropriate permissions; prepare the header and write the buffer contents to the file. Afterwards, close the file:

   ```
   fd = creat(fname, S_IRUSR | S_IWUSR);

   nbytes = size[0] * size[1] * 4;
   write_bitmap_header(nbytes, fd, size);

   for (i = 0; i < size[1]; i++) {
    write(fd, screenshot_ptr + i * screenshot_stride, size[0] * 4);
   }

   close(fd);
   ```

   The value of *nbytes* represents the calculated size of the bitmap and is used in the header of the bitmap itself.

Although any instances created are destroyed when the application exits, it is best practice to destroy any window, pixmap and context instances that you created but no longer require.

In this example, you should destroy the pixmap that you created to perform the screenshot. After the pixmap buffer has been used to create the bitmap, the pixmap, and its buffer are no longer required. Therefore you should perform the appropriate cleanup.

```
screen_destroy_pixmap(screen_pix);
```

Complete sample: a window screenshot example

The complete code sample for a window screenshot is listed below.

In the following code sample, an hourglass is placed in the top left corner of an application window while a vertical bar sweeps from left to right across the screen. The ground window is of type `SCREEN_APPLICATION_WINDOW`, while the hourglass and the bar are implemented as windows of type `SCREEN_CHILD_WINDOW`.

This code sample performs a screenshot of the hourglass window upon a `MTOUCH` touch event. The screenshot of the hourglass window is written to a designated bitmap on the system.

```
#include <ctype.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <screen/screen.h>

const char *hg_id_string = "hourglass";

const char *bar_id_string = "bar";

const int barwidth = 32;

screen_window_t screen_bg_win = NULL;

screen_window_t screen_hg_win = NULL;

screen_window_t screen_bar_win = NULL;

/* Create the background window in this example */
screen_window_t create_bg_window(const char *group, int dims[2], screen_context_t screen_ctx)
{
 /* Start by creating the context, application window and window group. */
 screen_window_t screen_win;
 screen_create_window(&screen_win, screen_ctx);
 screen_create_window_group(screen_win, group);

 /* Set the visibility of this window to FALSE; we want to make all windows
  * invisible until all the windows have been created and are ready to be displayed.*/
 int vis = 0;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);

 /* Set the color of the background window. */
 int color = 0xffffff00;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_COLOR, &color);

 /* Screen API
  * requires all visible windows to have at least one buffer, so
  * here we will create the smallest possible buffer since you don't need to use this buffer
  * in this example.*/
 int rect[4] = { 0, 0, 1, 1 };
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_SIZE, dims);

 /* Move the source viewport to outside the bounds of the window buffer to allow the
  * windowing system to replace all areas outside of the buffer with the window color.*/
 int pos[2] = { -dims[0], -dims[1] };
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_POSITION, pos);

 /* Create and post the window buffer to make this window visible when we are ready. */
 screen_buffer_t screen_buf;
 screen_create_window_buffers(screen_win, 1);
```

```
 screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_buf);
 screen_post_window(screen_win, screen_buf, 1, rect, 0);

 return screen_win;
}

/* Create the bar window in this example.*/
void create_bar_window(const char *group, const char *id, int dims[2])
{
 /* Start by creating the another context. A separate context for each child window
  * emphasizes the steps required on how to deal with child windows created by other processes.*/
 screen_context_t screen_ctx;
 screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);

 /* Create a child window. */
 screen_window_t screen_win;
 screen_create_window_type(&screen_win, screen_ctx, SCREEN_CHILD_WINDOW);
 screen_join_window_group(screen_win, group);
 screen_set_window_property_cv(screen_win, SCREEN_PROPERTY_ID_STRING, strlen(id), id);

 /* Set the visibility to FALSE. */
 int vis = 0;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);

 /* Set the color of the bar window. */
 int color = 0xff0000ff;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_COLOR, &color);

 /* Screen API requires all visible windows to have at
  * least one buffer, so here we will create the smallest possible buffer
  * since you don't need to use this buffer in this example.*/
 int rect[4] = { 0, 0, 1, 1 };
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);

 /* Move the source viewport to outside the bounds of the window buffer to allow the
 windowing system to replace all areas outside of the buffer with the window color.*/

 int pos[2] = { -rect[2], -rect[3] };
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_POSITION, pos);

 /* Create and post the window buffer to make this window visible when we are ready. */
 screen_buffer_t screen_buf;
 screen_create_window_buffers(screen_win, 1);
 screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_buf);
 screen_post_window(screen_win, screen_buf, 1, rect, 0);
}

/* Create the hourglass window in this example. */
void create_hg_window(const char *group, const char *id, int dims[2])
{
 int i, j;

 /* Start by creating the another context. A separate context for each child window
 emphasizes the steps required on how to deal with child windows created by other processes.*/
 screen_context_t screen_ctx;
 screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);

 /* Create a child window. */
 screen_window_t screen_win;
 screen_create_window_type(&screen_win, screen_ctx, SCREEN_CHILD_WINDOW);
 screen_join_window_group(screen_win, group);
 screen_set_window_property_cv(screen_win, SCREEN_PROPERTY_ID_STRING, strlen(id), id);

 /* Set the static window property to indicate that the contents of this window buffer
 will not change and therefore posting will not be expected.*/
 int flag = 1;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_STATIC, &flag);

 /* Set the visibility to FALSE. */
 int vis = 0;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);

 /* Set the pixel format. The hourglass shape will have transparency, so we need
  * a pixel format with an alpha channel; here we choose RGBA8888.*/
 int format = SCREEN_FORMAT_RGBA8888;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_FORMAT, &format);

 /* Set usage flag. Usage flag must be set to write in order for us to draw to the
  * window buffer. */
 int usage = SCREEN_USAGE_WRITE;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &usage);

 /* Set the transparency mode.
  * By default, RGBA8888 formats will have the transparency mode set to
  * source over. The windowing system assumes that if an application chooses
  * rgba over rgbx, it's because it wants to do some blending. However,
  * it is good practice to set the transparency mode. */
 int transparency = SCREEN_TRANSPARENCY_SOURCE_OVER;
 screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_TRANSPARENCY, &transparency);

 /* Set the window buffer size for the hourglass. */
```

```
    int rect[4] = { 0, 0, 100, 100 };
    screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, rect+2);

    /* Create the window buffer and then get a handle to this buffer. */
    screen_buffer_t screen_buf;
    screen_create_window_buffers(screen_win, 1);
    screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_RENDER_BUFFERS, (void **)&screen_buf);

    /* Get the pointer to this buffer in order to fill the hourglass shape. */
    char *ptr = NULL;
    screen_get_buffer_property_pv(screen_buf, SCREEN_PROPERTY_POINTER, (void **)&ptr);

    /* Get the stride (the number of bytes between pixels on different rows) so that we can
     * use it to draw the hourglass shape. */
    int stride = 0;
    screen_get_buffer_property_iv(screen_buf, SCREEN_PROPERTY_STRIDE, &stride);

    /* Draw the hourglass shape. */
    for (i = 0; i < rect[3]; i++, ptr += stride) {
     for (j = 0; j < rect[2]; j++) {
      ptr[j*4] = 0xa0;
      ptr[j*4+1] = 0xa0;
      ptr[j*4+2] = 0xa0;
      ptr[j*4+3] = ((j >= i && j <= rect[3]-i) || (j <= i && j >= rect[3]-i)) ? 0xff : 0;
     }
    }

    /* Post the window. */
    screen_post_window(screen_win, screen_buf, 1, rect, 0);
}

void write_bitmap_header(int nbytes, int fd, const int size[])
{
    char header[54];

    /* Set standard bitmap header */
    header[0] = 'B';
    header[1] = 'M';
    header[2] = nbytes & 0xff;
    header[3] = (nbytes >> 8) & 0xff;
    header[4] = (nbytes >> 16) & 0xff;
    header[5] = (nbytes >> 24) & 0xff;
    header[6] = 0;
    header[7] = 0;
    header[8] = 0;
    header[9] = 0;
    header[10] = 54;
    header[11] = 0;
    header[12] = 0;
    header[13] = 0;
    header[14] = 40;
    header[15] = 0;
    header[16] = 0;
    header[17] = 0;
    header[18] = size[0] & 0xff;
    header[19] = (size[0] >> 8) & 0xff;
    header[20] = (size[0] >> 16) & 0xff;
    header[21] = (size[0] >> 24) & 0xff;
    header[22] = -size[1] & 0xff;
    header[23] = (-size[1] >> 8) & 0xff;
    header[24] = (-size[1] >> 16) & 0xff;
    header[25] = (-size[1] >> 24) & 0xff;
    header[26] = 1;
    header[27] = 0;
    header[28] = 32;
    header[29] = 0;
    header[30] = 0;
    header[31] = 0;
    header[32] = 0;
    header[33] = 0;
    header[34] = 0; /* image size*/
    header[35] = 0;
    header[36] = 0;
    header[37] = 0;
    header[38] = 0x9;
    header[39] = 0x88;
    header[40] = 0;
    header[41] = 0;
    header[42] = 0x9l;
    header[43] = 0x88;
    header[44] = 0;
    header[45] = 0;
    header[46] = 0;
    header[47] = 0;
    header[48] = 0;
    header[49] = 0;
    header[50] = 0;
    header[51] = 0;
    header[52] = 0;
    header[53] = 0;
```

```
 /* Write bitmap header to file */
 write(fd, header, sizeof(header));
}

void write_bitmap_file(const int size[], const char* screenshot_ptr, const int screenshot_stride)
{
 int nbytes;  /* number of bytes of the bimap */
 int fd   /* file descriptor */
 int i;   /* iterator to iterate over the screenshot buffer */
 char *fname = "/accounts/1000/appdata/com.example.Tutorial_WindowApp."
               "testDev_l_WindowApp85f8001_/data/hourglass_window_screenshot.bmp"; /* bitmap filename */

 /* Calculate the size of the bitmap */
 nbytes = size[0] * size[1] * 4;

 /* Open file*/
 fd = creat(fname, S_IRUSR | S_IWUSR);

  /* Write the standard bitmap header */
 write_bitmap_header(nbytes, fd, size);

 /* Write screenshot buffer contents to file */
 for (i = 0; i < size[1]; i++) {
  write(fd, screenshot_ptr + i * screenshot_stride, size[0] * 4);
 }

}

/* Take window screenshot. */
void take_window_screenshot(screen_window_t screenshot_win, screen_context_t screenshot_ctx)
{
 /* Variables for setting up taking a screenshot. */
 screen_pixmap_t screen_pix;
 screen_buffer_t screenshot_buf;

 char *screenshot_ptr = NULL;
 int screenshot_stride = 0;
 int usage, format;
 int size[2];

 /* Create pixmap. */
 screen_create_pixmap(&screen_pix, screenshot_ctx);

 /* Set Usage Flags. */
 usage = SCREEN_USAGE_READ | SCREEN_USAGE_NATIVE;
 screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_USAGE, &usage);

 /* Set format. */
 format = SCREEN_FORMAT_RGBA8888;
 screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_FORMAT, &format);

 /* Set pixmap buffer size */
 size[0] = 200;
 size[1] = 200;
 screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_BUFFER_SIZE, size);

 /* Create pixmap buffer and get handle to the buffer. */
 screen_create_pixmap_buffer(screen_pix);
 screen_get_pixmap_property_pv(screen_pix, SCREEN_PROPERTY_RENDER_BUFFERS, (void**)&screenshot_buf);

 /* Get a pointer to the buffer. */
 screen_get_buffer_property_pv(screenshot_buf, SCREEN_PROPERTY_POINTER, (void**)&screenshot_ptr);

 /* Get the stride. */
 screen_get_buffer_property_iv(screenshot_buf, SCREEN_PROPERTY_STRIDE, &screenshot_stride);

 /* Take the window screenshot. */
 screen_read_window(screenshot_win, screenshot_buf, 0, NULL ,0);

 /* Write the screenshot buffer to a bitmap file*/
 write_bitmap_file(size, screenshot_ptr, screenshot_stride);

 /* Perform necessary Screen API clean-up. */
 screen_destroy_pixmap(screen_pix);
}

int main(int argc, char **argv)
{
 int pos[2], size[2];
 int vis = 0;
 int type;

 /* Create the context to set up connection with the windowing system. */
 screen_context_t screen_ctx;
 screen_create_context(&screen_ctx, SCREEN_APPLICATION_CONTEXT);

 /* Specify the dimensions when creating our child windows. */

 /* Get all displays available for the context. */
 int count = 0;
 screen_get_context_property_iv(screen_ctx, SCREEN_PROPERTY_DISPLAY_COUNT, &count);
```

```
screen_display_t *screen_disps = calloc(count, sizeof(screen_display_t));
screen_get_context_property_pv(screen_ctx, SCREEN_PROPERTY_DISPLAYS, (void **)screen_disps);
screen_display_t screen_disp = screen_disps[0];
free(screen_disps);

/* Get the size of the display; we will use this as the dimensions for our windows. */
int dims[2] = { 0, 0 };
screen_get_display_property_iv(screen_disp, SCREEN_PROPERTY_SIZE, dims);

/* Construct a unique name for the window group. */
char str[16];
snprintf(str, sizeof(str), "%d", getpid());

/* Create the parent window; in this example the background window is the parent. */
screen_bg_win = create_bg_window(str, dims, screen_ctx);

/* Create the child windows. */
create_bar_window(str, bar_id_string, dims);
create_hg_window(str, hg_id_string, dims);

/* Create a screen event handle to be used to receive events from the windowing system. */
screen_event_t screen_ev;
screen_create_event(&screen_ev);

while (1) {
 do {
  /* Wait for next event. */
  screen_get_event(screen_ctx, screen_ev, vis ? 0 : ~0);
  screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &type);

  /* We are interested only in post, close and mtouch events in this example. */
  if (type == SCREEN_EVENT_POST) {

   /* Get the handle for the window for this post event. */
   screen_window_t screen_win;
   screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&screen_win);
   screen_get_window_property_cv(screen_win, SCREEN_PROPERTY_ID_STRING, sizeof(str), str);

   /* Determine which window is posting. */
   if (!screen_bar_win && !strcmp(str, bar_id_string)) {
    screen_bar_win = screen_win;
   } else if (!screen_hg_win && !strcmp(str, hg_id_string)) {
    screen_hg_win = screen_win;
   }

   /* Once the child windows have been created and posted, switch
    * all windows to be visible.*/
   if (screen_bar_win && screen_hg_win) {
    vis = 1;

    /* Set the screen size to full screen, except for the hourglass which will be 100x100
     * and positioned at 10, 10. */
    screen_get_window_property_iv(screen_hg_win, SCREEN_PROPERTY_BUFFER_SIZE, size);
    screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_SIZE, size);

    pos[0] = pos[1] = 10;
    screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_POSITION, pos);

    pos[0] = pos[1] = 0;
    screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_POSITION, pos);
    screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_POSITION, pos);

    size[0] = barwidth;
    size[1] = dims[1];
    screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_SIZE, size);

    size[0] = dims[0];
    screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_SIZE, size);

    int zorder = 0;
    screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_ZORDER, &zorder);
    zorder++;
    screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_ZORDER, &zorder);
    zorder++;
    screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_ZORDER, &zorder);

    /* Set all windows visible. */
    screen_set_window_property_iv(screen_bg_win, SCREEN_PROPERTY_VISIBLE, &vis);
    screen_set_window_property_iv(screen_hg_win, SCREEN_PROPERTY_VISIBLE, &vis);
    screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_VISIBLE, &vis);
    screen_flush_context(screen_ctx, SCREEN_WAIT_IDLE);
   }
  } else if (type == SCREEN_EVENT_CLOSE) {

   /* Handle the window that just posted the close event. */
   screen_window_t screen_win;
   screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&screen_win);

   /* Track the window that just closed. */
   if (screen_win == screen_bar_win) {
    screen_bar_win = NULL;
```

```
    } else if (screen_win == screen_hg_win) {
     screen_hg_win = NULL;
    }

    /* Destroy the window that just closed, so that resources that
     * were allocated locally can be freed. */
    screen_destroy_window(screen_win);

    /* Update visibility for that window. */
    if (!screen_bar_win || !screen_hg_win) {
     vis = 0;
    }
   }else if (type == SCREEN_EVENT_MTOUCH_TOUCH){
    /* Handle the mtouch event to take screenshot of hourglass window. */
    take_window_screenshot(screen_hg_win, screen_ctx);
   }
  } while (type != SCREEN_EVENT_NONE);

  /* Wrap the position of the bar window back at the origin of the buffer before
   * the bar goes off the edge. Also, to prevent the animation from moving the
   * bar too fast, we will call screen_flush_context with the appropriate flags.
   * This will limit the animation to the refresh rate of the display. */
  if (vis) {
   if (++pos[0] > dims[0] - barwidth) {
    pos[0] = 0;
   }
   screen_set_window_property_iv(screen_bar_win, SCREEN_PROPERTY_POSITION, pos);
   screen_flush_context(screen_ctx, SCREEN_WAIT_IDLE);
  }
 }

 /* Perform necessary cleanup. In this example we will rely on the windowing system
  * to release the resources for the bar and hourglass window contexts when the
  * process exits. */
 screen_destroy_event(screen_ev);
 screen_destroy_context(screen_ctx);

 return EXIT_SUCCESS;
}
```

Capture a display screenshot

The following procedure describes how to use Screen to capture a screenshot of a single display. You can store the resulting screenshot in either a pixmap or window buffer for further manipulation. This particular procedure describes capturing the screenshot in a pixmap buffer and then writing the screenshot to a bitmap.

This sample application uses the components of a grey hourglass, a moving blue vertical bar, and a yellow background. It aims to demonstrate how to capture a screenshot using the Screen API.

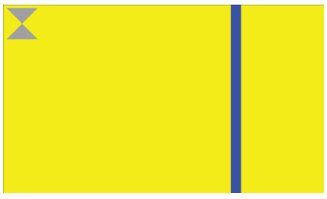


Figure 14: Screenshot Application

You will learn to:

- create a pixmap and buffer to store your screenshot
- retrieve appropriate pixmap properties to prepare for screenshot
- take your screenshot
- write your screenshot to a bitmap file

Before you begin

Before proceeding with the procedures to capture a screenshot, you are expected to have already created a privileged context using *screen_create_context()* with the context type of `SCREEN_DISPLAY_MANAGER_CONTEXT`. In order to be able to create this privileged context, remember that your process must have an effective user ID of root.

```
screen_context_t screen_ctx;
screen_create_context(&screenshot_ctx, SCREEN_DISPLAY_MANAGER_CONTEXT);
```

In the following procedure, the created context will be referred to as *screenshot_ctx*.

The targeted display will be referred to as *screenshot_disp*.

1. Create variables for the pixmap, the pixmap buffer, the pixmap buffer pointer, and the stride:

   ```
   screen_pixmap_t screen_pix;
   screen_buffer_t screenshot_buf;
   char *screenshot_ptr = NULL;
   int screenshot_stride = 0;
   ```

2. Create other variables necessary to support the Screen API calls and the writing of our screenshot to a bitmap.

   In this procedure, you will declare several integer variables to help in setting the pixmap and its properties. You will also need variables associated with the writing of the screenshot to bitmap. For this example, a set path and filename are used. Ensure that you have appropriate permissions to access the directory path of the file.

   ```
   char header[54];
   char *fname = "/accounts/1000/appdata/com.example.Tutorial_WindowApp."
                 "testDev_l_WindowApp85f8001_/data/hourglass_window_screenshot.bmp";
   int nbytes;
   int fd;
   int i;
   int usage, format;
   int size[2];
   ```

3. Create the pixmap for the screenshot and set the usage flag and format properties:

   ```
   screen_create_pixmap(&screen_pix, screenshot_ctx);

   usage = SCREEN_USAGE_READ | SCREEN_USAGE_NATIVE;
   screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_USAGE, &usage);

   format = SCREEN_FORMAT_RGBA8888;
   screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_FORMAT, &format);
   ```

4. Set the buffer size of the pixmap for the screenshot:

   Set an appropriate buffer size for the pixmap. The pixmap buffer size doesn't have to necessarily match the size of the source. Scaling will be applied to make the screenshot fit into the buffer provided.

   ```
   size[0] = 200;
   size[1] = 200;
   screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_BUFFER_SIZE, size);
   ```

5. Create the pixmap buffer for the screenshot and get the buffer handle, the pointer, and the stride.

   Memory is allocated for your pixmap buffer; this is the buffer where the pixels from the source window will be copied to:

   ```
   screen_create_pixmap_buffer(screen_pix);
   screen_get_pixmap_property_pv(screen_pix, SCREEN_PROPERTY_RENDER_BUFFERS,
                                 (void**)&screenshot_buf);
   screen_get_buffer_property_pv(screenshot_buf, SCREEN_PROPERTY_POINTER,
                                 (void**)&screentshot_ptr);
   screen_get_buffer_property_iv(screenshot_buf, SCREEN_PROPERTY_STRIDE,
                                 &screenshot_stride);
   ```

6. Take the display screenshot.

   ```
   screen_read_display(screenshot_disp, screenshot_buf, 0, NULL ,0);
   ```

   This function takes five arguments: the target of the screenshot, the pixmap buffer, the number of rectangles defining the area of capture, the array of integers representing rectangles of the area of capture, and the mutex flag. The arguments related to the area of capture are 0 and NULL because in this example you are capturing the target area in its entirety rather than a specific rectangular area. The last argument (which represents the mutex flag) should always be 0.

7. Create the bitmap file with appropriate permissions; prepare the header and write the buffer contents to the file. Afterwards, close the file:

   ```
   fd = creat(fname, S_IRUSR | S_IWUSR);

   nbytes = size[0] * size[1] * 4;
   write_bitmap_header(nbytes, fd, size);

   for (i = 0; i < size[1]; i++) {
    write(fd, screenshot_ptr + i * screenshot_stride, size[0] * 4);
   }

   close(fd);
   ```

   The value of *nbytes* represents the calculated size of the bitmap and is used in the header of the bitmap itself.

Although any instances created are destroyed when the application exits, it is best practice to destroy any window, pixmap and context instances that you created but no longer require.

In this example, you should destroy the pixmap that you created to perform the screenshot. After the pixmap buffer has been used to create the bitmap, the pixmap, and its buffer are no longer required. Therefore you should perform the appropriate cleanup.

```
screen_destroy_pixmap(screen_pix);
```

Complete sample: A display screenshot example

The complete code sample for a display screenshot is listed below.

This code sample performs a screenshot of the specified display and then the screenshot is written to a designated bitmap on the system.

```
#include <ctype.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <screen/screen.h>

void write_bitmap_header(int nbytes, int fd, const int size[])
{
 char header[54];

 /* Set standard bitmap header */
 header[0] = 'B';
 header[1] = 'M';
 header[2] = nbytes & 0xff;
 header[3] = (nbytes >> 8) & 0xff;
 header[4] = (nbytes >> 16) & 0xff;
 header[5] = (nbytes >> 24) & 0xff;
 header[6] = 0;
 header[7] = 0;
 header[8] = 0;
 header[9] = 0;
 header[10] = 54;
 header[11] = 0;
 header[12] = 0;
 header[13] = 0;
 header[14] = 40;
 header[15] = 0;
 header[16] = 0;
 header[17] = 0;
 header[18] = size[0] & 0xff;
 header[19] = (size[0] >> 8) & 0xff;
 header[20] = (size[0] >> 16) & 0xff;
 header[21] = (size[0] >> 24) & 0xff;
 header[22] = -size[1] & 0xff;
 header[23] = (-size[1] >> 8) & 0xff;
 header[24] = (-size[1] >> 16) & 0xff;
 header[25] = (-size[1] >> 24) & 0xff;
 header[26] = 1;
 header[27] = 0;
 header[28] = 32;
 header[29] = 0;
 header[30] = 0;
 header[31] = 0;
 header[32] = 0;
 header[33] = 0;
 header[34] = 0; /* image size*/
 header[35] = 0;
 header[36] = 0;
 header[37] = 0;
 header[38] = 0x9;
 header[39] = 0x88;
 header[40] = 0;
 header[41] = 0;
 header[42] = 0x9l;
 header[43] = 0x88;
 header[44] = 0;
 header[45] = 0;
 header[46] = 0;
 header[47] = 0;
 header[48] = 0;
 header[49] = 0;
 header[50] = 0;
 header[51] = 0;
 header[52] = 0;
 header[53] = 0;

 /* Write bitmap header to file */
 write(fd, header, sizeof(header));
}

void write_bitmap_file(const int size[], const char* screenshot_ptr, const int screenshot_stride)
{
 int nbytes;       /* number of bytes of the bimap */
 int fd;        /* file descriptor */
 int i;        /* iterator to iterate over the screenshot buffer */
```

```
 char *fname =  "screenshot.bmp"; /* bitmap filename */

 /* Calculate the size of the bitmap */
 nbytes = size[0] * size[1] * 4;

 /* Open file*/
 fd = creat(fname, S_IRUSR | S_IWUSR);

  /* Write the standard bitmap header */
 write_bitmap_header(nbytes, fd, size);

 /* Write screenshot buffer contents to file */
 for (i = 0; i < size[1]; i++) {
  write(fd, screenshot_ptr + i * screenshot_stride, size[0] * 4);
 }

}


int main(int argc, char **argv)
{
 screen_context_t screen_ctx;
 screen_display_t screen_disp = NULL;
 screen_display_t *screen_displays;
 screen_pixmap_t screen_pix;
 screen_buffer_t screen_buf;
 char *fname = "screenshot.bmp";
 char *disp = NULL;
 char *tok;
 char header[54];
 int size[2] = { 0, 0 };
 int val;
 void *pointer;
 int stride;
 int nbytes;
 int count, id, type;
 int i, fd, rc;

 /* Parse command-line input. */
 for (i = 1; i < argc; i++) {
  if (strncmp(argv[i], "-size=", strlen("-size=")) == 0) {
   tok = argv[i] + strlen("-size=");
   size[0] = atoi(tok);
   while (*tok >= '0' && *tok <= '9') {
    tok++;
   }
   size[1] = atoi(tok+1);
  } else if (strncmp(argv[i], "-display=", strlen("-display=")) == 0) {
   disp = argv[i] + strlen("-display=");
  } else if (strncmp(argv[i], "-file=", strlen("-file=")) == 0) {
   fname = argv[i] + strlen("-file=");
  } else {
   fprintf(stderr, "invalid command line option: %s\n", argv[i]);
  }
 }

 /* Create the privileged context so that the display properties can be accessed. */
 screen_create_context(&screen_ctx, SCREEN_DISPLAY_MANAGER_CONTEXT);

 /* Get the number of supported displays with this context. */
 count = 0;
 screen_get_context_property_iv(screen_ctx, SCREEN_PROPERTY_DISPLAY_COUNT,
                                &count);

 /* Get the displays for this context. */
 if (count > 0) {
  screen_displays = calloc(count, sizeof(screen_display_t));
  screen_get_context_property_pv(screen_ctx, SCREEN_PROPERTY_DISPLAYS,
                                 (void **)screen_displays);

  /* If no display was specified, use the first supported display available for this context. */
  if (!disp) {
   screen_disp = screen_displays[0];
  /* Otherwise, determine which display has been requested for the screen shot. */
  } else {
   if (isdigit(*disp)) {
    id = strtoul(disp, 0, NULL);
    for (i = 0; i < count; i++) {
     screen_get_display_property_iv(screen_displays[i], SCREEN_PROPERTY_ID,
                                    &val);
     if (val == id) {
      screen_disp = screen_displays[i];
      break;
     }
    }
   } else {
    if (!strcmp(disp, "internal")) {
     type = SCREEN_DISPLAY_TYPE_INTERNAL;
    } else if (!strcmp(disp, "rgb")) {
     type = SCREEN_DISPLAY_TYPE_COMPONENT_RGB;
    } else if (!strcmp(disp, "dvi")) {
```

```
        type = SCREEN_DISPLAY_TYPE_DVI;
       } else if (!strcmp(disp, "hdmi")) {
        type = SCREEN_DISPLAY_TYPE_HDMI;
       } else {
        type = SCREEN_DISPLAY_TYPE_OTHER;
       }
       for (i = 0; i < count; i++) {
        screen_get_display_property_iv(screen_displays[i], SCREEN_PROPERTY_TYPE,
                                       &val);
        if (val == type) {
         screen_disp = screen_displays[i];
         break;
        }
       }
      }
     }

     free(screen_displays);
    }

    if (!screen_disp) {
     fputs("no displays\n", stderr);
     return 1;
    }

    /* Create pixmap for the screen shot. */
    screen_create_pixmap(&screen_pix, screen_ctx);

    /* Set usage flag. */
    val = SCREEN_USAGE_READ | SCREEN_USAGE_NATIVE;
    screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_USAGE, &val);


    /* Set format. */
    val = SCREEN_FORMAT_RGBA8888;
    screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_FORMAT, &val);

    /* If size is not specified, get the size from the display. */
    if (size[0] <= 0 || size[1] <= 0) {
     screen_get_display_property_iv(screen_disp, SCREEN_PROPERTY_SIZE, size);
    }

    /* Set the pixmap buffer size. */
    screen_set_pixmap_property_iv(screen_pix, SCREEN_PROPERTY_BUFFER_SIZE, size);

    /* Create the pixmap buffer and get a handle to the buffer. */
    screen_create_pixmap_buffer(screen_pix);

    screen_get_pixmap_property_pv(screen_pix, SCREEN_PROPERTY_RENDER_BUFFERS,
                                  (void **)&screen_buf);

    /* Get the pointer to the buffer. */
    screen_get_buffer_property_pv(screen_buf, SCREEN_PROPERTY_POINTER, &pointer);

    /* Get the stride. */
    screen_get_buffer_property_iv(screen_buf, SCREEN_PROPERTY_STRIDE, &stride);

    /* Take the display screen shot. */
    screen_read_display(screen_disp, screen_buf, 0, NULL, 0);

     /* Write the screenshot buffer to a bitmap file*/
    write_bitmap_file(size, pointer, stride);

    /* Perform necessary clean-up. */
    screen_destroy_pixmap(screen_pix);
    screen_destroy_context(screen_ctx);

    return 0;
   }
```

Tutorial: Rendering text with FreeType and OpenGL ES

This is sample application that uses a native window to create an EGL on-screen rendering surface. Text is rendered on this surface by using the FreeType library with OpenGL.

This sample application aims to demonstrate how to integrate the use of the FreeType library, OpenGL ES 1.X, and Screen to render text.

Figure 15: Hello World Application

You will learn to:

- create a native context
- initialize and configure EGL for rendering
- create a native window and setting its properties
- calculate the DPI to use based on your display size and resolution
- load a font
- load background texture
- create a main application loop to:
  - process events in the native context
  - render text using OpenGL ES 1.X
- release resources

Using FreeType library and OpenGL ES to render text

The following walkthrough takes you through the process of writing a native application that uses the FreeType library and OpenGL ES for text rendering.

To use FreeType and OpenGL ES for your text rendering in a native application:

1. Create some basic variables you'll need for your application:

```
int rc;                         /* a return code */
screen_event_t screen_ev;       /* a screen event to handle */
screen_context_t screen_ctx;    /* a connection to the screen windowing system */
int vis = 1;                    /* an indicator if our window is visible */
int pause = 0;                  /* an indicator if rendering is frozen */
const int exit_area_size = 20;  /* a size of area on the window where a user can
```

```
                                 * contact (using a pointer) to exit this application */
int size[2] = { 0, 0 };         /* the width and height of your window; will default to
                                 * the size of display since the window property wasn't
                                 * explicitly set */
int val;                        /* a variable used to set/get window properties */
int pos[2] = { 0, 0 };          /* the x,y position of your pointer */
```

2. Create your native context.

```
rc = screen_create_context(&screen_ctx, 0);
```

3. Establish a connection to the EGL display.

   Before you can do any kind of rendering, you must establish a connection to a display.

   In this sample application, you will use the default display.

```
egl_disp = eglGetDisplay(EGL_DEFAULT_DISPLAY);
```

4. Initialize the EGL display.

   You will be able to do little with the EGL display until it's been initialized. The second and third arguments of *eglInitialize()* are both set to NULL because OpenGL ES 1.X is supported by all versions of EGL; therefore it isn't necessary to check for the major and minor version numbers.

```
rc = eglInitialize(egl_disp, NULL, NULL);
```

5. Choose an EGL configuration.

   First establish your EGL configuration attributes:

```
static EGLConfig egl_conf;      /* An aray of framebuffer configurations */
int num_configs;                /* The number of framebuffer configurations from eglChooseConfig()
 */

EGLint attrib_list[]= { EGL_RED_SIZE,        8,
                        EGL_GREEN_SIZE,      8,
                        EGL_BLUE_SIZE,       8,
                        EGL_BLUE_SIZE,       8,
                        EGL_SURFACE_TYPE,    EGL_WINDOW_BIT,
                        EGL_RENDERABLE_TYPE, EGL_OPENGL_ES_BIT,
                        EGL_NONE};
```

   Then you can use *eglChooseConfigs()* to choose your EGL configuration. The function *eglChooseConfigs()* is probably the most complicated function of EGL; there are many attributes that can be specified, each with its own matching rules, default value, and sorting order. It's easy to get confused with all the special rules ending up with the wrong configuration, or no configuration, without understanding why. Be aware of this fact when you are specifying your EGL configuration attributes.

```
rc = eglChooseConfig(egl_disp, attrib_list, &egl_conf, 1,
&num_configs))
```

6. Create an OpenGL ES rendering context.

   Now, create an OpenGL ES rendering context. Among other things, this context keeps track of the OpenGL ES state. You don't need to specify the current rendering

API with the *eglBindApi()* function because OpenGL ES is the default rendering API.

The third argument to *eglCreateContext()* is another EGL rendering context with which you wish to share data. Pass EGL_NO_CONTEXT to indicate that you won't need any of the textures or vertex buffer objects created in another EGL rendering context.

The last argument to *eglCreateContext()* is an attribute list that you can use to specify an API version number. You would use it to override the `EGL_CONTEXT_CLIENT_VERSION` value from `1` to `2` if you were writing an OpenGL ES 2.X application.

```
egl_ctx = eglCreateContext(egl_disp, egl_conf, EGL_NO_CONTEXT, NULL);
```

7. Create your native window.

```
rc = screen_create_window(&screen_win, screen_ctx);
```

8. Set your native window properties.
   a. Set your window format and usage.

```
int format = SCREEN_FORMAT_RGBX8888;
usage = SCREEN_USAGE_OPENGL_ES1 | SCREEN_USAGE_ROTATION;

rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_FORMAT, &format);
rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &usage);
```

   b. Set your window buffer size.

      In this case, you are going to set the buffer size of your window based on the orientation of the display.

```
int angle = atoi(getenv("ORIENTATION"));

screen_display_mode_t screen_mode;
rc = screen_get_display_property_pv(screen_disp, SCREEN_PROPERTY_MODE, (void**)&screen_mode);

int size[2];
rc = screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, size);

int buffer_size[2] = {size[0], size[1]};

if ((angle == 0) || (angle == 180)){
 if (((screen_mode.width > screen_mode.height) && (size[0] < size[1])) ||
  ((screen_mode.width < screen_mode.height) && (size[0] > size[1])))
  {
   buffer_size[1] = size[0];
   buffer_size[0] = size[1];
     }
} else if ((angle == 90) || (angle == 270)){
 if (((screen_mode.width > screen_mode.height) && (size[0] > size[1])) ||
  ((screen_mode.width < screen_mode.height && size[0] < size[1])))
  {
   buffer_size[1] = size[0];
   buffer_size[0] = size[1];
     }
}

rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, buffer_size);
```

   c. Set your window rotation.

```
rc = screen_set_window_property_iv(screen_win,
SCREEN_PROPERTY_ROTATION, &angle);
```

9. Create your window buffers for rendering.

```
rc = screen_create_window_buffers(screen_win, nbuffers);
```

10. Create the EGL on-screen rendering surface.

    Now that you've created a native platform window, you can use it to create an EGL on-screen rendering surface. You'll be able to use this surface as the target of your OpenGL ES rendering. You'll use the same EGL display and EGL configuration to create the EGL surface as you used to set the properties on your native window. The EGL configuration needs to be compatible with the one used to create the window.

```
egl_surf = eglCreateWindowSurface(egl_disp, egl_conf, screen_win, NULL);
```

11. Bind the EGL context to the current rendering thread and to a draw-and-read surface.

    In this application, you want to draw to the EGL surface and not really care about where you read from. Since EGL doesn't allow specifying EGL_NO_SURFACE for only the read surface, you will use *egl_surf* for both drawing and reading. Once *eglMakeCurrent()* completes successfully, all OpenGL ES calls will be executed on the context and the surface you provided as arguments.

```
rc = eglMakeCurrent(egl_disp, egl_surf, egl_surf, egl_ctx);
```

12. Set the EGL swap interval.

    The *eglSwapInterval()* function specifies the minimum number of video frame periods per buffer swap for the window associated with the current context. So, if the interval is 0, the application renders as fast as it can. Interval values of 1 or more limit the rendering to fractions of the display's refresh rate. (For example, 60, 30, 20, 15, etc. frames per second in the case of a display with a refresh rate of 60 Hz.)

```
rc = eglSwapInterval(egl_disp, interval);
```

13. Calculate the display resolution based on the display size.

```
int screen_phys_size[2] = { 0, 0 };

screen_get_display_property_iv(screen_disp, SCREEN_PROPERTY_PHYSICAL_SIZE, screen_phys_size);

/* If using a simulator, {0,0} is returned for physical size of the screen,
   so use 170 as the default dpi when this is the case. */
if ((screen_phys_size[0] == 0) && (screen_phys_size[1] == 0)){
        return 170;
} else{
        int screen_resolution[2] = { 0, 0 };
        screen_get_display_property_iv(screen_disp, SCREEN_PROPERTY_SIZE, screen_resolution);

        int diagonal_pixels = sqrt(screen_resolution[0] * screen_resolution[0]
                              + screen_resolution[1] * screen_resolution[1]);
        int diagonal_inches = 0.0393700787 * sqrt(screen_phys_size[0] * screen_phys_size[0]
                              + screen_phys_size[1] * screen_phys_size[1]);
        return (int)(diagonal_pixels / diagonal_inches);
}
```

14. Load your font.

Pick an appropriate font and ensure that you have the correct directory path to access that font.

In this example, the function *load_font()* is a simple utility function. It is written to facilitate the steps to load and ready the font for use by OpenGL.

```
font = load_font("/usr/fonts/font_repository/monotype/georgiab.ttf", 8, dpi);
```

15. Load your background texture.

    In this example, the function *load_texture()* is a simple utility function.

```
load_texture("app/native/HelloWorld_smaller_bubble.png", NULL, NULL, &tex_x, &tex_y, &background)
```

16. Initialize the Graphics Library for 2D rendering.

```
/* Query width and height of the window surface created by utility code */
eglQuerySurface(egl_disp, egl_surf, EGL_WIDTH, &surface_width);
eglQuerySurface(egl_disp, egl_surf, EGL_HEIGHT, &surface_height);

width = (float) surface_width;
height = (float) surface_height;

glViewport(0, 0, (int) width, (int) height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrthof(0.0f, width / height, 0.0f, 1.0f, -1.0f, 1.0f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

/* Set world coordinates to coincide with screen pixels */
glScalef(1.0f / height, 1.0f / height, 1.0f);

float text_width, text_height;
measure_text(font, "Hello world", &text_width, &text_height);
pos_x = (width - text_width) / 2;
pos_y = height / 2;

/* Setup background polygon */
vertices[0] = 0.0f;
vertices[1] = 0.0f;
vertices[2] = width;
vertices[3] = 0.0f;
vertices[4] = 0.0f;
vertices[5] = height;
vertices[6] = width;
vertices[7] = height;

tex_coord[0] = 0.0f;
tex_coord[1] = 0.0f;
tex_coord[2] = tex_x;
tex_coord[3] = 0.0f;
tex_coord[4] = 0.0f;
tex_coord[5] = tex_y;
tex_coord[6] = tex_x;
tex_coord[7] = tex_y;
```

17. Create a screen event.

    This screen event you create will be used to retrieve event information so that each event can be handled.

```
rc = screen_create_event(&screen_ev);
```

18. Create a main application loop that continues running until an explicit event to close the application (or a system error) occurs.

    This main application loop consists of two parts:

    The first part of the loop processes screen events.

```
while (!screen_get_event(screen_ctx, screen_ev, vis ? 0 : ~0))
{
 rc = screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &val);
 if (rc || val == SCREEN_EVENT_NONE)
 {
  break;
 }
 switch (val) {
    case SCREEN_EVENT_CLOSE:
```

```
    goto end;
    case SCREEN_EVENT_PROPERTY:
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_NAME, &val);
    switch (val) {
     case SCREEN_PROPERTY_VISIBLE:
      screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);
     break;
    }
    break;
    case SCREEN_EVENT_POINTER:
        screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_BUTTONS, &val);
     if (val) {
        screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_POSITION, pos);
     screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_SIZE, size);
     fprintf(stderr, "window width: %d, window height: %d\n", size[0], size[1]);
     fprintf(stderr, "pointer x: %d, pointer y: %d\n", pos[0], pos[1]);
     if (pos[0] >= size[0] - exit_area_size &&
         pos[1] < exit_area_size) {
      goto end;
     }
     }
     break;
 }
}
```

The second part of the loop performs the rendering.

Perform the rendering only if your window is visible. This will leave the CPU and GPU available to other applications and make the system more responsive while your window is invisible.

In this example, the function *render()* is a utility function that renders the background, sets the color to use for text rendering, renders the text onto the screen, and updates the screen (posts the new frame).

```
if (vis && !pause)
{
    rc = render();
}
```

19. Perform the appropriate cleanup.

```
/* Destroy the font */
if (font) {
 glDeleteTextures(1, &(font->font_texture));
    free(font);
}
/* Terminate EGL setup */
if (egl_disp != EGL_NO_DISPLAY) {
   eglMakeCurrent(egl_disp, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);
   if (egl_surf != EGL_NO_SURFACE) {
       eglDestroySurface(egl_disp, egl_surf);
       egl_surf = EGL_NO_SURFACE;
   }
   if (egl_ctx != EGL_NO_CONTEXT) {
       eglDestroyContext(egl_disp, egl_ctx);
       egl_ctx = EGL_NO_CONTEXT;
   }
   eglTerminate(egl_disp);
   egl_disp = EGL_NO_DISPLAY;
}
eglReleaseThread();

/* Clean up screen */
if (screen_win != NULL) {
    screen_destroy_window(screen_win);
    screen_win = NULL;
}
screen_destroy_event(screen_ev);
screen_destroy_context(screen_ctx);
```

Complete sample: Rendering text with FreeType and OpenGL ES

This code sample uses Screen with the FreeType library and OpenGL ES for text rendering.

Figure 16: Hello World Application

```
/*
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include <ctype.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/keycodes.h>
#include <screen/screen.h>
#include <ft2build.h>
#include FT_FREETYPE_H
#include <math.h>
#include <EGL/egl.h>
#include <GLES/gl.h>
#include "png.h"

static float width, height;
static GLuint background;
static GLfloat vertices[8];
static GLfloat tex_coord[8];
static float pos_x, pos_y;

struct font_t {
 unsigned int font_texture;
 float pt;
 float advance[128];
 float width[128];
 float height[128];
 float tex_x1[128];
 float tex_x2[128];
 float tex_y1[128];
 float tex_y2[128];
 float offset_x[128];
 float offset_y[128];
 int initialized;
};

typedef struct font_t font_t;

EGLDisplay egl_disp;
EGLSurface egl_surf;

static EGLConfig egl_conf;
static EGLContext egl_ctx;
static screen_window_t screen_win = NULL;
static screen_display_t screen_disp;
```

```
static int nbuffers = 2;
static int initialized = 0;
static font_t* font;

/* Utility function to calculate the dpi based on the display size */
int calculate_dpi() {
    int screen_phys_size[2] = { 0, 0 };

    screen_get_display_property_iv(screen_disp, SCREEN_PROPERTY_PHYSICAL_SIZE, screen_phys_size);

    /* If using a simulator, {0,0} is returned for physical size of the screen,
     so use 170 as the default dpi when this is the case. */
    if ((screen_phys_size[0] == 0) && (screen_phys_size[1] == 0)) {
        return 170;
    } else{
        int screen_resolution[2] = { 0, 0 };
        screen_get_display_property_iv(screen_disp, SCREEN_PROPERTY_SIZE, screen_resolution);

        int diagonal_pixels = sqrt(screen_resolution[0] * screen_resolution[0]
                                   + screen_resolution[1] * screen_resolution[1]);
        int diagonal_inches = 0.0393700787 * sqrt(screen_phys_size[0] * screen_phys_size[0]
                                                  + screen_phys_size[1] * screen_phys_size[1]);
        return (int)(diagonal_pixels / diagonal_inches);
    }
}

static inline int
nextp2(int x)
{
    int val = 1;
    while(val < x) val <<= 1;
    return val;
}


/* Utility function to load and ready the font for use by OpenGL */
font_t* load_font(const char* path, int point_size, int dpi) {
    FT_Library library;
    FT_Face face;
    int c;
    int i, j;
    font_t* font;

    if (!initialized) {
        fprintf(stderr, "EGL has not been initialized\n");
        return NULL;
    }

    if (!path){
        fprintf(stderr, "Invalid path to font file\n");
        return NULL;
    }

    if(FT_Init_FreeType(&library)) {
        fprintf(stderr, "Error loading Freetype library\n");
        return NULL;
    }
    if (FT_New_Face(library, path,0,&face)) {
        fprintf(stderr, "Error loading font %s\n", path);
        return NULL;
    }

    if(FT_Set_Char_Size ( face, point_size * 64, point_size * 64, dpi, dpi)) {
        fprintf(stderr, "Error initializing character parameters\n");
        return NULL;
    }

    font = (font_t*) malloc(sizeof(font_t));
    font->initialized = 0;

    glGenTextures(1, &(font->font_texture));

    /*Let each glyph reside in 32x32 section of the font texture */
    int segment_size_x = 0, segment_size_y = 0;
    int num_segments_x = 16;
    int num_segments_y = 8;

    FT_GlyphSlot slot;
    FT_Bitmap bmp;
    int glyph_width, glyph_height;

    /*First calculate the max width and height of a character in a passed font*/
    for(c = 0; c < 128; c++) {
        if(FT_Load_Char(face, c, FT_LOAD_RENDER)) {
            fprintf(stderr, "FT_Load_Char failed\n");
            free(font);
            return NULL;
        }

        slot = face->glyph;
        bmp = slot->bitmap;
```

```
            glyph_width = bmp.width;
            glyph_height = bmp.rows;

            if (glyph_width > segment_size_x) {
                segment_size_x = glyph_width;
            }

            if (glyph_height > segment_size_y) {
                segment_size_y = glyph_height;
            }
        }

        int font_tex_width = nextp2(num_segments_x * segment_size_x);
        int font_tex_height = nextp2(num_segments_y * segment_size_y);

        int bitmap_offset_x = 0, bitmap_offset_y = 0;

        GLubyte* font_texture_data = (GLubyte*) malloc(sizeof(GLubyte) * 2 * font_tex_width * font_tex_height);
        memset((void*)font_texture_data, 0, sizeof(GLubyte) * 2 * font_tex_width * font_tex_height);

        if (!font_texture_data) {
            fprintf(stderr, "Failed to allocate memory for font texture\n");
            free(font);
            return NULL;
        }

        /* Fill font texture bitmap with individual bmp data and record appropriate size,
         texture coordinates and offsets for every glyph */
        for(c = 0; c < 128; c++) {
            if(FT_Load_Char(face, c, FT_LOAD_RENDER)) {
                fprintf(stderr, "FT_Load_Char failed\n");
                free(font);
                return NULL;
            }

            slot = face->glyph;
            bmp = slot->bitmap;

            glyph_width = nextp2(bmp.width);
            glyph_height = nextp2(bmp.rows);

            div_t temp = div(c, num_segments_x);

            bitmap_offset_x = segment_size_x * temp.rem;
            bitmap_offset_y = segment_size_y * temp.quot;

            for (j = 0; j < glyph_height; j++) {
                for (i = 0; i < glyph_width; i++) {
                    font_texture_data[2 * ((bitmap_offset_x + i) + (j + bitmap_offset_y) * font_tex_width) + 0] =
                    font_texture_data[2 * ((bitmap_offset_x + i) + (j + bitmap_offset_y) * font_tex_width) + 1] =
                        (i >= bmp.width || j >= bmp.rows)? 0 : bmp.buffer[i + bmp.width * j];
                }
            }

            font->advance[c] = (float)(slot->advance.x >> 6);
            font->tex_x1[c] = (float)bitmap_offset_x / (float) font_tex_width;
            font->tex_x2[c] = (float)(bitmap_offset_x + bmp.width) / (float)font_tex_width;
            font->tex_y1[c] = (float)bitmap_offset_y / (float) font_tex_height;
            font->tex_y2[c] = (float)(bitmap_offset_y + bmp.rows) / (float)font_tex_height;
            font->width[c] = bmp.width;
            font->height[c] = bmp.rows;
            font->offset_x[c] = (float)slot->bitmap_left;
            font->offset_y[c] =  (float)((slot->metrics.horiBearingY-face->glyph->metrics.height) >> 6);
        }

        glBindTexture(GL_TEXTURE_2D, font->font_texture);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);

        glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE_ALPHA, font_tex_width, font_tex_height, 0, GL_LUMINANCE_ALPHA , GL_UNSIGNED_BYTE,

        int err = glGetError();

        free(font_texture_data);

        FT_Done_Face(face);
        FT_Done_FreeType(library);

        if (err != 0) {
            fprintf(stderr, "GL Error 0x%x", err);
            free(font);
            return NULL;
        }

        font->initialized = 1;
        return font;
    }

    /* Utility function to load background texture */
    int load_texture(const char* filename, int* width, int* height, float* tex_x, float* tex_y, unsigned int *tex) {
```

```
int i;
GLuint format;
png_byte header[8]; /* header for testing if it is a png */

if (!tex) {
    return EXIT_FAILURE;
}

/* Open file as binary */
FILE *fp = fopen(filename, "rb");
if (!fp) {
    return EXIT_FAILURE;
}

/* Read the header */
fread(header, 1, 8, fp);

/* Test if png */
int is_png = !png_sig_cmp(header, 0, 8);
if (!is_png) {
    fclose(fp);
    return EXIT_FAILURE;
}

/* Create png struct */
png_structp png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
if (!png_ptr) {
    fclose(fp);
    return EXIT_FAILURE;
}

/* Create png info struct */
png_infop info_ptr = png_create_info_struct(png_ptr);
if (!info_ptr) {
    png_destroy_read_struct(&png_ptr, (png_infopp) NULL, (png_infopp) NULL);
    fclose(fp);
    return EXIT_FAILURE;
}

/* Create png info struct */
png_infop end_info = png_create_info_struct(png_ptr);
if (!end_info) {
    png_destroy_read_struct(&png_ptr, &info_ptr, (png_infopp) NULL);
    fclose(fp);
    return EXIT_FAILURE;
}

/* Set up error handling (required without using custom error handlers above) */
if (setjmp(png_jmpbuf(png_ptr))) {
    png_destroy_read_struct(&png_ptr, &info_ptr, &end_info);
    fclose(fp);
    return EXIT_FAILURE;
}

/* Initialize png reading */
png_init_io(png_ptr, fp);

/* Let libpng know you already read the first 8 bytes */
png_set_sig_bytes(png_ptr, 8);

/* Read all the info up to the image data */
png_read_info(png_ptr, info_ptr);

/* Variables to pass to get info */
int bit_depth, color_type;
png_uint_32 image_width, image_height;

/* Get info about png */
png_get_IHDR(png_ptr, info_ptr, &image_width, &image_height, &bit_depth, &color_type, NULL, NULL, NULL);

switch (color_type)
{
    case PNG_COLOR_TYPE_RGBA:
        format = GL_RGBA;
        break;
    case PNG_COLOR_TYPE_RGB:
        format = GL_RGB;
        break;
    default:
        fprintf(stderr,"Unsupported PNG color type (%d) for texture: %s", (int)color_type, filename);
        fclose(fp);
        png_destroy_read_struct(&png_ptr, &info_ptr, &end_info);
        return NULL;
}

/* Update the png info struct. */
png_read_update_info(png_ptr, info_ptr);

/* Row size in bytes. */
int rowbytes = png_get_rowbytes(png_ptr, info_ptr);
```

```
    /* Allocate the image_data as a big block, to be given to opengl */
    png_byte *image_data = (png_byte*) malloc(sizeof(png_byte) * rowbytes * image_height);

    if (!image_data) {
        /* clean up memory and close file */
        png_destroy_read_struct(&png_ptr, &info_ptr, &end_info);
        fclose(fp);
        return EXIT_FAILURE;
    }

    /* Row_pointers is for pointing to image_data for reading the png with libpng */
    png_bytep *row_pointers = (png_bytep*) malloc(sizeof(png_bytep) * image_height);
    if (!row_pointers) {
        /* clean up memory and close stuff */
        png_destroy_read_struct(&png_ptr, &info_ptr, &end_info);
        free(image_data);
        fclose(fp);
        return EXIT_FAILURE;
    }

    /* Set the individual row_pointers to point at the correct offsets of image_data */
    for (i = 0; i < image_height; i++) {
        row_pointers[image_height - 1 - i] = image_data + i * rowbytes;
    }

    /* Read the png into image_data through row_pointers */
    png_read_image(png_ptr, row_pointers);

    int tex_width, tex_height;

    tex_width = nextp2(image_width);
    tex_height = nextp2(image_height);

    glGenTextures(1, tex);
    glBindTexture(GL_TEXTURE_2D, (*tex));
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    glTexImage2D(GL_TEXTURE_2D, 0, format, tex_width, tex_height, 0, format, GL_UNSIGNED_BYTE, NULL);
    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, image_width, image_height, format, GL_UNSIGNED_BYTE, image_data);

    GLint err = glGetError();

    /* Clean up memory and close file */
    png_destroy_read_struct(&png_ptr, &info_ptr, &end_info);
    free(image_data);
    free(row_pointers);
    fclose(fp);

    if (err == 0) {
        /* Return physical with and height of texture if pointers are not null */
        if(width) {
            *width = image_width;
        }
        if (height) {
            *height = image_height;
        }
        /* Return modified texture coordinates if pointers are not null */
        if(tex_x) {
            *tex_x = ((float) image_width - 0.5f) / ((float)tex_width);
        }
        if(tex_y) {
            *tex_y = ((float) image_height - 0.5f) / ((float)tex_height);
        }
        return EXIT_SUCCESS;
    } else {
        fprintf(stderr, "GL error %i \n", err);
        return EXIT_FAILURE;
    }
}

/* Utility function to perform EGL cleanup */
void egl_cleanup() {
    /* Typical EGL cleanup */
    if (egl_disp != EGL_NO_DISPLAY) {
        eglMakeCurrent(egl_disp, EGL_NO_SURFACE, EGL_NO_SURFACE, EGL_NO_CONTEXT);
        if (egl_surf != EGL_NO_SURFACE) {
            eglDestroySurface(egl_disp, egl_surf);
            egl_surf = EGL_NO_SURFACE;
        }
        if (egl_ctx != EGL_NO_CONTEXT) {
            eglDestroyContext(egl_disp, egl_ctx);
            egl_ctx = EGL_NO_CONTEXT;
        }
        if (screen_win != NULL) {
            screen_destroy_window(screen_win);
            screen_win = NULL;
```

```
        }
        eglTerminate(egl_disp);
        egl_disp = EGL_NO_DISPLAY;
    }
    eglReleaseThread();

    initialized = 0;
}

/* Utility function to initialize and configure a EGL rendering surface */
int init_egl(screen_context_t screen_ctx) {
    int usage;
    int format = SCREEN_FORMAT_RGBX8888;
    EGLint interval = 1;
    int rc, num_configs;

    EGLint attrib_list[]= { EGL_RED_SIZE,        8,
                            EGL_GREEN_SIZE,      8,
                            EGL_BLUE_SIZE,       8,
                            EGL_BLUE_SIZE,       8,
                            EGL_SURFACE_TYPE,    EGL_WINDOW_BIT,
                            EGL_RENDERABLE_TYPE, EGL_OPENGL_ES_BIT,
                            EGL_NONE};

    /* Assuming GL_ES_1 */
    usage = SCREEN_USAGE_OPENGL_ES1 | SCREEN_USAGE_ROTATION;

    /* Establish a connection to the default display */
    egl_disp = eglGetDisplay(EGL_DEFAULT_DISPLAY);
    if (egl_disp == EGL_NO_DISPLAY) {
     egl_cleanup();
        return EXIT_FAILURE;
    }

    /* Initialize EGL on the display */
    rc = eglInitialize(egl_disp, NULL, NULL);
    if (rc != EGL_TRUE) {
     egl_cleanup();
        return EXIT_FAILURE;
    }

    /* Calling eglBindAPI() to specify the current rendering API is not necessary
     * because OpenGL ES is the default rendering API.
    rc = eglBindAPI(EGL_OPENGL_ES_API);

    if (rc != EGL_TRUE) {
     egl_cleanup();
        return EXIT_FAILURE;
    }*/

    if(!eglChooseConfig(egl_disp, attrib_list, &egl_conf, 1, &num_configs)) {
     egl_cleanup();
        return EXIT_FAILURE;
    }

    egl_ctx = eglCreateContext(egl_disp, egl_conf, EGL_NO_CONTEXT, NULL);

    if (egl_ctx == EGL_NO_CONTEXT) {
     egl_cleanup();
        return EXIT_FAILURE;
    }

    rc = screen_create_window(&screen_win, screen_ctx);
    if (rc) {
        perror("screen_create_window");
        egl_cleanup();
        return EXIT_FAILURE;
    }

    rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_FORMAT, &format);
    if (rc) {
        perror("screen_set_window_property_iv(SCREEN_PROPERTY_FORMAT)");
        egl_cleanup();
        return EXIT_FAILURE;
    }

    rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &usage);
    if (rc) {
        perror("screen_set_window_property_iv(SCREEN_PROPERTY_USAGE)");
        egl_cleanup();
        return EXIT_FAILURE;
    }

    rc = screen_get_window_property_pv(screen_win, SCREEN_PROPERTY_DISPLAY, (void **)&screen_disp);
    if (rc) {
        perror("screen_get_window_property_pv");
        egl_cleanup();
        return EXIT_FAILURE;
    }

    int angle = atoi(getenv("ORIENTATION"));
```

```
 screen_display_mode_t screen_mode;
 rc = screen_get_display_property_pv(screen_disp, SCREEN_PROPERTY_MODE, (void**)&screen_mode);
 if (rc) {
  perror("screen_get_display_property_pv");
  egl_cleanup();
  return EXIT_FAILURE;
 }

 int size[2];
 rc = screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, size);
 if (rc) {
  perror("screen_get_window_property_iv");
  egl_cleanup();
  return EXIT_FAILURE;
 }

 int buffer_size[2] = {size[0], size[1]};

 if ((angle == 0) || (angle == 180)) {
  if (((screen_mode.width > screen_mode.height) && (size[0] < size[1])) ||
   ((screen_mode.width < screen_mode.height) && (size[0] > size[1]))) {
    buffer_size[1] = size[0];
    buffer_size[0] = size[1];
  }
 } else if ((angle == 90) || (angle == 270)){
  if (((screen_mode.width > screen_mode.height) && (size[0] > size[1])) ||
   ((screen_mode.width < screen_mode.height && size[0] < size[1]))) {
    buffer_size[1] = size[0];
    buffer_size[0] = size[1];
  }
 } else {
   fprintf(stderr, "Navigator returned an unexpected orientation angle.\n");
   egl_cleanup();
   return EXIT_FAILURE;
 }

 rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, buffer_size);
    if (rc) {
        perror("screen_set_window_property_iv");
        egl_cleanup();
        return EXIT_FAILURE;
    }

    rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_ROTATION, &angle);
    if (rc) {
        perror("screen_set_window_property_iv");
        egl_cleanup();
        return EXIT_FAILURE;
    }

    rc = screen_create_window_buffers(screen_win, nbuffers);
    if (rc) {
        perror("screen_create_window_buffers");
        egl_cleanup();
        return EXIT_FAILURE;
    }

    egl_surf = eglCreateWindowSurface(egl_disp, egl_conf, screen_win, NULL);
    if (egl_surf == EGL_NO_SURFACE) {
        egl_cleanup();
        return EXIT_FAILURE;
    }

    rc = eglMakeCurrent(egl_disp, egl_surf, egl_surf, egl_ctx);
    if (rc != EGL_TRUE) {
        egl_cleanup();
        return EXIT_FAILURE;
    }

    rc = eglSwapInterval(egl_disp, interval);
    if (rc != EGL_TRUE) {
        egl_cleanup();
        return EXIT_FAILURE;
    }

    initialized = 1;

    return EXIT_SUCCESS;
}

void measure_text(font_t* font, const char* msg, float* width, float* height) {
    int i, c;

    if (!msg) {
        return;
    }

    if (width) {
        /* Width of a text rectangle is a sum advances for every glyph in a string */
        *width = 0.0f;
```

```
        for(i = 0; i < strlen(msg); ++i) {
            c = msg[i];
            *width += font->advance[c];
        }
    }

    if (height) {
        /* Height of a text rectangle is a high of a tallest glyph in a string */
        *height = 0.0f;

        for(i = 0; i < strlen(msg); ++i) {
            c = msg[i];

            if (*height < font->height[c]) {
                *height = font->height[c];
            }
        }
    }
}

int init() {
 EGLint surface_width, surface_height;

 /* We are going to load MyriadPro-Bold as it looks a little better and scale it to
    fit out bubble nicely. */
 int dpi = calculate_dpi();

 /*font = load_font(
   "/usr/fonts/font_repository/adobe/MyriadPro-Bold.otf", 15, dpi); */
 font = load_font(
   "/usr/fonts/font_repository/monotype/georgiab.ttf", 8, dpi);
 if (!font) {
  return EXIT_FAILURE;
 }

 /* Load background texture */
 float tex_x, tex_y;
 if (EXIT_SUCCESS
   != load_texture("app/native/HelloWorld_smaller_bubble.png",
     NULL, NULL, &tex_x, &tex_y, &background)) {
  fprintf(stderr, "Unable to load background texture\n");
 }

 /* Query width and height of the window surface created by utility code */
 eglQuerySurface(egl_disp, egl_surf, EGL_WIDTH, &surface_width);
 eglQuerySurface(egl_disp, egl_surf, EGL_HEIGHT, &surface_height);

 EGLint err = eglGetError();
 if (err != 0x3000) {
  fprintf(stderr, "Unable to query EGL surface dimensions\n");
  return EXIT_FAILURE;
 }

 width = (float) surface_width;
 height = (float) surface_height;

 /* Initialize GL for 2D rendering */
 glViewport(0, 0, (int) width, (int) height);

 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();

 glOrthof(0.0f, width / height, 0.0f, 1.0f, -1.0f, 1.0f);

 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();

 /* Set world coordinates to coincide with screen pixels */
 glScalef(1.0f / height, 1.0f / height, 1.0f);

 float text_width, text_height;
 measure_text(font, "Hello world", &text_width, &text_height);
 pos_x = (width - text_width) / 2;
 pos_y = height / 2;

 /* Setup background polygon */
 vertices[0] = 0.0f;
 vertices[1] = 0.0f;
 vertices[2] = width;
 vertices[3] = 0.0f;
 vertices[4] = 0.0f;
 vertices[5] = height;
 vertices[6] = width;
 vertices[7] = height;

 tex_coord[0] = 0.0f;
 tex_coord[1] = 0.0f;
 tex_coord[2] = tex_x;
 tex_coord[3] = 0.0f;
 tex_coord[4] = 0.0f;
```

```
 tex_coord[5] = tex_y;
 tex_coord[6] = tex_x;
 tex_coord[7] = tex_y;

 return EXIT_SUCCESS;
}

void render_text(font_t* font, const char* msg, float x, float y) {
    int i, c;
    GLfloat *vertices;
    GLfloat *texture_coords;
    GLshort* indices;

    float pen_x = 0.0f;

    if (!font) {
        fprintf(stderr, "Font must not be null\n");
        return;
    }

    if (!font->initialized) {
        fprintf(stderr, "Font has not been loaded\n");
        return;
    }

    if (!msg) {
        return;
    }

    int texture_enabled;
    glGetIntegerv(GL_TEXTURE_2D, &texture_enabled);
    if (!texture_enabled) {
        glEnable(GL_TEXTURE_2D);
    }

    int blend_enabled;
    glGetIntegerv(GL_BLEND, &blend_enabled);
    if (!blend_enabled) {
        glEnable(GL_BLEND);
    }

    int gl_blend_src, gl_blend_dst;
    glGetIntegerv(GL_BLEND_SRC, &gl_blend_src);
    glGetIntegerv(GL_BLEND_DST, &gl_blend_dst);

    glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);

    int vertex_array_enabled;
    glGetIntegerv(GL_VERTEX_ARRAY, &vertex_array_enabled);
    if (!vertex_array_enabled) {
        glEnableClientState(GL_VERTEX_ARRAY);
    }

    int texture_array_enabled;
    glGetIntegerv(GL_TEXTURE_COORD_ARRAY, &texture_array_enabled);
    if (!texture_array_enabled) {
        glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    }

    vertices = (GLfloat*) malloc(sizeof(GLfloat) * 8 * strlen(msg));
    texture_coords = (GLfloat*) malloc(sizeof(GLfloat) * 8 * strlen(msg));

    indices = (GLshort*) malloc(sizeof(GLfloat) * 5 * strlen(msg));

    for(i = 0; i < strlen(msg); ++i) {
        c = msg[i];

        vertices[8 * i + 0] = x + pen_x + font->offset_x[c];
        vertices[8 * i + 1] = y + font->offset_y[c];
        vertices[8 * i + 2] = vertices[8 * i + 0] + font->width[c];
        vertices[8 * i + 3] = vertices[8 * i + 1];
        vertices[8 * i + 4] = vertices[8 * i + 0];
        vertices[8 * i + 5] = vertices[8 * i + 1] + font->height[c];
        vertices[8 * i + 6] = vertices[8 * i + 2];
        vertices[8 * i + 7] = vertices[8 * i + 5];

        texture_coords[8 * i + 0] = font->tex_x1[c];
        texture_coords[8 * i + 1] = font->tex_y2[c];
        texture_coords[8 * i + 2] = font->tex_x2[c];
        texture_coords[8 * i + 3] = font->tex_y2[c];
        texture_coords[8 * i + 4] = font->tex_x1[c];
        texture_coords[8 * i + 5] = font->tex_y1[c];
        texture_coords[8 * i + 6] = font->tex_x2[c];
        texture_coords[8 * i + 7] = font->tex_y1[c];

        indices[i * 6 + 0] = 4 * i + 0;
        indices[i * 6 + 1] = 4 * i + 1;
        indices[i * 6 + 2] = 4 * i + 2;
        indices[i * 6 + 3] = 4 * i + 2;
        indices[i * 6 + 4] = 4 * i + 1;
        indices[i * 6 + 5] = 4 * i + 3;
```

```
        /* Assume we are only working with typewriter fonts */
        pen_x += font->advance[c];
    }

    glVertexPointer(2, GL_FLOAT, 0, vertices);
    glTexCoordPointer(2, GL_FLOAT, 0, texture_coords);
    glBindTexture(GL_TEXTURE_2D, font->font_texture);

    glDrawElements(GL_TRIANGLES, 6 * strlen(msg), GL_UNSIGNED_SHORT, indices);

    if (!texture_array_enabled) {
        glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    }

    if (!vertex_array_enabled) {
        glDisableClientState(GL_VERTEX_ARRAY);
    }

    if (!texture_enabled) {
        glDisable(GL_TEXTURE_2D);
    }

    glBlendFunc(gl_blend_src, gl_blend_dst);

    if (!blend_enabled) {
        glDisable(GL_BLEND);
    }

    free(vertices);
    free(texture_coords);
    free(indices);
}

int render() {
 /* Typical rendering pass */
 glClear(GL_COLOR_BUFFER_BIT);

 /* Render background quad first */
 glEnable(GL_TEXTURE_2D);

 glEnableClientState(GL_VERTEX_ARRAY);
 glEnableClientState(GL_TEXTURE_COORD_ARRAY);

 glVertexPointer(2, GL_FLOAT, 0, vertices);
 glTexCoordPointer(2, GL_FLOAT, 0, tex_coord);
 glBindTexture(GL_TEXTURE_2D, background);

 glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
 glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

 glDisableClientState(GL_TEXTURE_COORD_ARRAY);
 glDisableClientState(GL_VERTEX_ARRAY);
 glDisable(GL_TEXTURE_2D);

 /* Set color to use for text rendering */
 glColor4f(0.35f, 0.35f, 0.35f, 1.0f);

 /* Render the text onto the screen */
 render_text(font, "Hello world", pos_x, pos_y);

 /* Update the screen; Posting of the new frame requires a call to eglSwapBuffers.
  * For now, this is true even when using single buffering. If an
  * event has occured that invalidates the surface we are currently
  * using, eglSwapBuffers() will return EGL_FALSE and set the error
  * code to EGL_BAD_NATIVE_WINDOW.
  */
 int rc = eglSwapBuffers(egl_disp, egl_surf);

 return rc;

}

int main(int argc, char **argv) {
 int rc;       /* a return code */
    screen_event_t screen_ev;  /* a screen event to handle */
    screen_context_t screen_ctx; /* a connection to the screen windowing system */
 int vis = 1;     /* an indicator if our window is visible */
 int pause = 0;     /* an indicator if rendering is frozen */

 const int exit_area_size = 20; /* a size of area on the window where a user can
          * contact (using a pointer) to exit this application */

 int size[2] = { 0, 0 };        /* the width and height of your window; will default to
          * the size of display since the window property wasn't
             * explicitly set */

 int val;      /* a variable used to set/get window properties */
 int pos[2] = { 0, 0 };          /* the x,y position of your pointer */

 /*Create a screen context that will be the connection to the windowing system;
```

```
 *this is used to create an EGL surface to receive libscreen events */
screen_create_context(&screen_ctx, 0);

/* Initialize EGL for rendering with GL ES 1.1;
 * this initialization includes initializing and configuring EGL as well as creating
 * a native window with the appropriate properties to be used as the EGL rendering
 * surface. */
if (EXIT_SUCCESS != init_egl(screen_ctx)) {
 fprintf(stderr, "Unable to initialize EGL\n");
 screen_destroy_context(screen_ctx);
 return 0;
}

/* Initialize application data;
 * this initialization includes loading the font and background and initializing
 * the viewport and geometry for your application. */
if (EXIT_SUCCESS != init()) {
 fprintf(stderr, "Unable to initialize app logic\n");
 egl_cleanup();
 screen_destroy_context(screen_ctx);
 return 0;
}

/* Create a screen event that will be used to retrieve events into so that these
 * events can be handled.*/
rc = screen_create_event(&screen_ev);
if (rc) {
 fprintf(stderr, "screen_create_event\n");
 egl_cleanup();
 screen_destroy_context(screen_ctx);
 return 0;
}

/* This is your main application loop. It keeps on running unless a close event is
 * received from the windowing system or an error occurs. The application loop consists
 * of two parts. The first part processes any events that have been put in your queue.
 * The second part does the rendering. When the window is visible, you don't wait if
 * the event queue is empty; you move on to the rendering part immediately.
 * When the window is not visible we skip the rendering part. */
while (1){
 /* The first part of the loop is to handle screen events.
  * We start the loop by processing any events that might be in our
  * queue. The only event that is of interest to us are the resize
  * and close events. The timeout variable is set to 0 (no wait) or
  * forever depending if the window is visible or not. */
 while (!screen_get_event(screen_ctx, screen_ev, vis ? 0 : ~0))
 {
  rc = screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &val);
  if (rc || val == SCREEN_EVENT_NONE)
  {
   break;
  }
  switch (val) {
   case SCREEN_EVENT_CLOSE:
     /* All we have to do when we receive the close event is
      * to exit the application loop. Because we have a loop
      * within a loop, a simple break won't work. We'll just
      * use a goto to take us out of here.*/
    goto end;
   case SCREEN_EVENT_PROPERTY:
     /* We are interested in visibility changes so we can pause
      * or unpause the rendering. */
    screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_NAME, &val);
    switch (val) {
     case SCREEN_PROPERTY_VISIBLE:
       /* The new visibility status is not included in the
        * event, so we must get it ourselves. */
      screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &vis);
      break;
    }
    break;
   case SCREEN_EVENT_POINTER:
     /* To provide a way of gracefully terminating our application,
      * we will exit if there is a pointer select event in the upper
      * right corner of our window. This should happen if the mouse's
      * left button is clicked or if a touch screen display is pressed.
      * The event will come as a screen pointer event, with an (x,y)
      * coordinate relative to the window's upper left corner and a
      * select value. We have to verify ourselves that the coordinates
      * of the pointer are in the upper right hand area. */
    screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_BUTTONS, &val);
    if (val) {
     screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_POSITION, pos);
     screen_get_window_property_iv(screen_win, SCREEN_PROPERTY_SIZE, size);
     fprintf(stderr, "window width: %d, window height: %d\n", size[0], size[1]);
     fprintf(stderr, "pointer x: %d, pointer y: %d\n", pos[0], pos[1]);
     if (pos[0] >= size[0] - exit_area_size &&
      pos[1] < exit_area_size) {
      goto end;
     }
    }
```

```
      break;
    }
   }

   /* The second part of the application loop is the rendering. You want
    * to skip the rendering part if your window is not visible. This will
    * leave the CPU and GPU to other applications and make the system a
    * little bit more responsive while you are invisible. */
   if (vis && !pause)
   {
    rc = render();
    if (rc != EGL_TRUE) break;
   }
  }

     end:
  /* Destroy the font */
  if (font) {
   glDeleteTextures(1, &(font->font_texture));
      free(font);
  }

  /* Terminate EGL setup */
  egl_cleanup();

  /* Clean up screen */
  screen_destroy_event(screen_ev);
  screen_destroy_context(screen_ctx);

  return EXIT_SUCCESS;
 }
```

Tutorial: Screen events

This is a sample application that injects a screen event into a specified display.

When you are working within the privileged contexts of either SCREEN_WINDOW_MANAGER_CONTEXT or SCREEN_INPUT_PROVIDER_CONTEXT, it's possible to inject a Screen event into the system. This sample application demonstrates how to use the Screen API to do so.

You will learn to:

- create a native context
- create a screen event and set the appropriate properties
- inject the screen event into a specified display
- release resources

Injecting a Screen event

The following walkthrough takes you through the process of injecting a screen event into a specified display.

1. Create some basic variables you'll need for your application:

```
screen_context_t screen_ctx;    /* a connection to the screen windowing system */
screen_display_t screen_disp;   /* a screen display */
screen_display_t *screen_dlist; /* a list of all available displays */
screen_event_t screen_ev;       /* a screen event to handle */
int ndisplays;                  /* number of available displays */
int val;                        /* a variable used to set/get window properties */
const char *display = "1";      /* the display type */
int rval = EXIT_FAILURE;        /* the application return code*/
int rc;                         /* a return code */
```

2. Create your native context with a privileged context that allows for the injection of Screen events. In this example, you will use the `SCREEN_INPUT_PROVIDER_CONTEXT` context.

```
rc = screen_create_context(&screen_ctx, SCREEN_INPUT_PROVIDER_CONTEXT);
```

3. Identify the display into which your Screen event is to be injected.

   In this example, the display used is the internal display, unless it is otherwise specified as a command-line argument.

```
rc = screen_get_context_property_iv(screen_ctx,
                                    SCREEN_PROPERTY_DISPLAY_COUNT,
                                    &ndisplays);
if (rc) {
    perror("screen_get_context_property_iv(SCREEN_PROPERTY_DISPLAY_COUNT)");
    goto fail2;
}

screen_dlist = calloc(ndisplays, sizeof(*screen_dlist));
if (screen_dlist == NULL) {
    fprintf(stderr, "could not allocate memory for display list\n");
    goto fail2;
}

rc = screen_get_context_property_pv(screen_ctx,
```

```
                                        SCREEN_PROPERTY_DISPLAYS,
                                        (void **)screen_dlist);
if (rc) {
    perror("screen_get_context_property_pv(SCREEN_PROPERTY_DISPLAYS)");
    free(screen_dlist);
    goto fail2;
}

if (isdigit(*display)) {
    j = atoi(display) - 1;
} else {
    int type = -1;
    if (strcmp(display, "internal") == 0) {
        type = SCREEN_DISPLAY_TYPE_INTERNAL;
    } else if (strcmp(display, "composite") == 0) {
        type = SCREEN_DISPLAY_TYPE_COMPOSITE;
    } else if (strcmp(display, "svideo") == 0) {
        type = SCREEN_DISPLAY_TYPE_SVIDEO;
    } else if (strcmp(display, "YPbPr") == 0) {
        type = SCREEN_DISPLAY_TYPE_COMPONENT_YPbPr;
    } else if (strcmp(display, "rgb") == 0) {
        type = SCREEN_DISPLAY_TYPE_COMPONENT_RGB;
    } else if (strcmp(display, "rgbhv") == 0) {
        type = SCREEN_DISPLAY_TYPE_COMPONENT_RGBHV;
    } else if (strcmp(display, "dvi") == 0) {
        type = SCREEN_DISPLAY_TYPE_DVI;
    } else if (strcmp(display, "hdmi") == 0) {
        type = SCREEN_DISPLAY_TYPE_HDMI;
    } else if (strcmp(display, "other") == 0) {
        type = SCREEN_DISPLAY_TYPE_OTHER;
    } else {
        fprintf(stderr, "unknown display type %s\n", display);
        free(screen_dlist);
        goto fail2;
    }
    for (j = 0; j < ndisplays; j++) {
        screen_get_display_property_iv(screen_dlist[j],
                                       SCREEN_PROPERTY_TYPE,
                                       &val);
        if (val == type) {
            break;
        }
    }
}

if (j >= ndisplays) {
    fprintf(stderr, "couldn't find display %s\n", display);
    free(screen_dlist);
    goto fail2;
}

screen_disp = screen_dlist[j];
free(screen_dlist);
```

4. Create a screen event.

   This is the screen event that you will be injecting into the display.

   ```
   rc = screen_create_event(&screen_ev);
   ```

5. Set the type of the event that you will be injecting.

   In this example, you will be injecting a keyboard event.

   ```
   val = SCREEN_EVENT_KEYBOARD;
   rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &val);
   ```

6. Set the value of the keyboard flags that are associated with this keyboard event.

   ```
   val = KEY_DOWN|KEY_SYM_VALID;
   rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_FLAGS, &val);
   ```

7. Set the value of the keyboard symbols that are associated with this keyboard event.

   In this example, the keyboard symbols are passed as command-line arguments.

   ```
   val = argv[i][j];
   rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_SYM, &val);
   ```

8. Inject your Screen event.

```
rc = screen_inject_event(screen_disp, screen_ev);
```

9. Release the resources.

```
screen_destroy_event(screen_ev);
screen_destroy_context(screen_ctx);
```

Complete sample: Injecting a Screen event

This code sample uses a privileged context to inject a Screen event into a specified display.

```
/*
 * $QNXLicenseC:
 * Copyright 2011, QNX Software Systems Limited. All Rights Reserved.
 *
 * This software is QNX Confidential Information subject to
 * confidentiality restrictions. DISCLOSURE OF THIS SOFTWARE
 * IS PROHIBITED UNLESS AUTHORIZED BY QNX SOFTWARE SYSTEMS IN
 * WRITING.
 *
 * You must obtain a written license from and pay applicable license
 * fees to QNX Software Systems Limited before you may reproduce, modify
 * or distribute this software, or any work that includes all or part
 * of this software. For more information visit
 * http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review
 * this entire file for other proprietary rights or license notices,
 * as well as the QNX Development Suite License Guide at
 * http://licensing.qnx.com/license-guide/ for other information.
 * $
 */

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/keycodes.h>
#include <time.h>
#include <screen/screen.h>

int main(int argc, char **argv)
{
 screen_context_t screen_ctx;
 screen_display_t screen_disp;
 screen_display_t *screen_dlist;
 screen_event_t screen_ev;
    int ndisplays;
 int val;
 const char *display = "1";
 int rval = EXIT_FAILURE;
 int rc;
 int i, j;

 for (i = 1; i < argc; i++) {
  if (strncmp(argv[i], "-display=", strlen("-display=")) == 0) {
   display = argv[i] + strlen("-display=");
  } else {
   break;
  }
 }

 rc = screen_create_context(&screen_ctx, SCREEN_INPUT_PROVIDER_CONTEXT);
 if (rc) {
  perror("screen_context_create");
  goto fail1;
 }

 rc = screen_get_context_property_iv(screen_ctx, SCREEN_PROPERTY_DISPLAY_COUNT, &ndisplays);
 if (rc) {
  perror("screen_get_context_property_iv(SCREEN_PROPERTY_DISPLAY_COUNT)");
  goto fail2;
 }

 screen_dlist = calloc(ndisplays, sizeof(*screen_dlist));
 if (screen_dlist == NULL) {
  fprintf(stderr, "could not allocate memory for display list\n");
```

```
 goto fail2;
}

rc = screen_get_context_property_pv(screen_ctx, SCREEN_PROPERTY_DISPLAYS, (void **)screen_dlist);
if (rc) {
 perror("screen_get_context_property_pv(SCREEN_PROPERTY_DISPLAYS)");
 free(screen_dlist);
 goto fail2;
}

if (isdigit(*display)) {
 j = atoi(display) - 1;
} else {
 int type = -1;
 if (strcmp(display, "internal") == 0) {
  type = SCREEN_DISPLAY_TYPE_INTERNAL;
 } else if (strcmp(display, "composite") == 0) {
  type = SCREEN_DISPLAY_TYPE_COMPOSITE;
 } else if (strcmp(display, "svideo") == 0) {
  type = SCREEN_DISPLAY_TYPE_SVIDEO;
 } else if (strcmp(display, "YPbPr") == 0) {
  type = SCREEN_DISPLAY_TYPE_COMPONENT_YPbPr;
 } else if (strcmp(display, "rgb") == 0) {
  type = SCREEN_DISPLAY_TYPE_COMPONENT_RGB;
 } else if (strcmp(display, "rgbhv") == 0) {
  type = SCREEN_DISPLAY_TYPE_COMPONENT_RGBHV;
 } else if (strcmp(display, "dvi") == 0) {
  type = SCREEN_DISPLAY_TYPE_DVI;
 } else if (strcmp(display, "hdmi") == 0) {
  type = SCREEN_DISPLAY_TYPE_HDMI;
 } else if (strcmp(display, "other") == 0) {
  type = SCREEN_DISPLAY_TYPE_OTHER;
 } else {
  fprintf(stderr, "unknown display type %s\n", display);
  free(screen_dlist);
  goto fail2;
 }
 for (j = 0; j < ndisplays; j++) {
  screen_get_display_property_iv(screen_dlist[j], SCREEN_PROPERTY_TYPE, &val);
  if (val == type) {
   break;
  }
 }
}

if (j >= ndisplays) {
 fprintf(stderr, "couldn't find display %s\n", display);
 free(screen_dlist);
 goto fail2;
}

screen_disp = screen_dlist[j];
free(screen_dlist);

rc = screen_create_event(&screen_ev);
if (rc) {
 perror("screen_create_event");
 goto fail2;
}

val = SCREEN_EVENT_KEYBOARD;
rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &val);
if (rc) {
 perror("screen_set_event_property_iv(SCREEN_PROPERTY_TYPE)");
 goto fail3;
}

val = KEY_DOWN|KEY_SYM_VALID;
rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_FLAGS, &val);
if (rc) {
 perror("screen_set_event_property_iv(SCREEN_PROPERTY_KEY_FLAGS)");
 goto fail3;
}

for (; i < argc; i++) {
 for (j = 0; argv[i][j]; j++) {
  val = argv[i][j];
  rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_SYM, &val);
  if (rc) {
   perror("screen_set_event_property_iv(SCREEN_PROPERTY_KEY_SYM)");
   goto fail3;
  }

  rc = screen_inject_event(screen_disp, screen_ev);
  if (rc) {
   perror("screen_inject_event");
   goto fail3;
  }
 }
}
rval = EXIT_SUCCESS;
```

```
fail3:
 screen_destroy_event(screen_ev);
fail2:
 screen_destroy_context(screen_ctx);
fail1:
 return rval;
}
```

Injecting a Screen `mtouch` event

The following walkthrough takes you through the process of injecting an Screen `mtouch` event into a specified display.

1. Create some basic variables you'll need for your application:

```
screen_context_t screen_ctx;    /* a connection to the screen windowing system */
screen_display_t screen_disp;   /* a screen display */
screen_display_t *screen_dlist; /* a list of all available displays */
screen_event_t screen_ev;       /* a screen event to handle */
int ndisplays;                  /* number of available displays */
int val;                        /* a variable used to set/get window properties */
const char *display = "1";      /* the display type */
int rval = EXIT_FAILURE;        /* the application return code*/
int rc;                         /* a return code */
```

2. Create your native context with a privileged context that allows for the injection of Screen events. In this example, you will use the `SCREEN_INPUT_PROVIDER_CONTEXT` context.

```
rc = screen_create_context(&screen_ctx, SCREEN_INPUT_PROVIDER_CONTEXT);
```

3. Identify the display into which your Screen event is to be injected.

   In this example, the display used is the internal display, unless it is otherwise specified as a command-line argument.

```
rc = screen_get_context_property_iv(screen_ctx,
                                    SCREEN_PROPERTY_DISPLAY_COUNT,
                                    &ndisplays);
if (rc) {
    perror("screen_get_context_property_iv(SCREEN_PROPERTY_DISPLAY_COUNT)");
    goto fail2;
}

screen_dlist = calloc(ndisplays, sizeof(*screen_dlist));
if (screen_dlist == NULL) {
    fprintf(stderr, "could not allocate memory for display list\n");
    goto fail2;
}

rc = screen_get_context_property_pv(screen_ctx,
                                    SCREEN_PROPERTY_DISPLAYS,
                                    (void **)screen_dlist);
if (rc) {
    perror("screen_get_context_property_pv(SCREEN_PROPERTY_DISPLAYS)");
    free(screen_dlist);
    goto fail2;
}

if (isdigit(*display)) {
    j = atoi(display) - 1;
} else {
    int type = -1;
    if (strcmp(display, "internal") == 0) {
        type = SCREEN_DISPLAY_TYPE_INTERNAL;
    } else if (strcmp(display, "composite") == 0) {
        type = SCREEN_DISPLAY_TYPE_COMPOSITE;
    } else if (strcmp(display, "svideo") == 0) {
        type = SCREEN_DISPLAY_TYPE_SVIDEO;
    } else if (strcmp(display, "YPbPr") == 0) {
        type = SCREEN_DISPLAY_TYPE_COMPONENT_YPbPr;
    } else if (strcmp(display, "rgb") == 0) {
        type = SCREEN_DISPLAY_TYPE_COMPONENT_RGB;
    } else if (strcmp(display, "rgbhv") == 0) {
```

```
            type = SCREEN_DISPLAY_TYPE_COMPONENT_RGBHV;
        } else if (strcmp(display, "dvi") == 0) {
            type = SCREEN_DISPLAY_TYPE_DVI;
        } else if (strcmp(display, "hdmi") == 0) {
            type = SCREEN_DISPLAY_TYPE_HDMI;
        } else if (strcmp(display, "other") == 0) {
            type = SCREEN_DISPLAY_TYPE_OTHER;
        } else {
            fprintf(stderr, "unknown display type %s\n", display);
            free(screen_dlist);
            goto fail2;
        }
        for (j = 0; j < ndisplays; j++) {
            screen_get_display_property_iv(screen_dlist[j],
                                           SCREEN_PROPERTY_TYPE,
                                           &val);
            if (val == type) {
                break;
            }
        }
    }

    if (j >= ndisplays) {
        fprintf(stderr, "couldn't find display %s\n", display);
        free(screen_dlist);
        goto fail2;
    }

    screen_disp = screen_dlist[j];
    free(screen_dlist);
```

4. Create a Screen event.

   This is the Screen `mtouch` event that you will be injecting into the display.

   ```
   rc = screen_create_event(&screen_ev);
   ```

5. Set the type of the event that you will be injecting.

   In this example, you will be injecting an `mtouch` event.

   ```
   val = SCREEN_EVENT_MTOUCH_TOUCH;
   rc = screen_set_event_property_iv(screen_ev,
                                     SCREEN_PROPERTY_TYPE,
                                     &val);
   ```

6. Set the value of each property that is associated with your `mtouch` event. The properties you set be used are used in the `mtouch_event` structure.

   a. Set the value of the device ID that is associated with this `mtouch` event.

      ```
      int device = 0;
      rc = screen_set
      _event_property_iv(screen_ev,
                         SCREEN_PROPERTY_DEVICE_INDEX,
                         &device);
      ```

   b. Set the value of the position of the touch point that is associated with this `mtouch` event.

      ```
      int pos[2] = {5, 5};
      rc = screen_set_event_property_iv(screen_ev,
                                        SCREEN_PROPERTY_POSITION,
                                        &pos);
      ```

   c. Set the value of the sequence ID that is associated with this `mtouch` event.

      The sequence ID is the sequence number of the mtouch event. You need to increment this number for each new `mtouch` event you send.

      ```
      int seq_id = 1;
      rc = screen_set_event_property_iv(screen_ev,
      ```

```
                                    SCREEN_PROPERTY_SEQUENCE_ID,
                                    &pos);
```

d. Set the value of the size of the touch area that is associated with this `mtouch` event.

The size, width and height in pixels, of the touch area.

```
int size[2] = { 5, 5};
rc = screen_set_event_property_iv(screen_ev,
                                    SCREEN_PROPERTY_SIZE,
                                    &size);
```

e. Set the value of the source position and source size.

In this example, we use the same position and size because the offset of the window and display are one and the same.

```
rc = screen_set_event_property_iv(screen_ev,
                                    SCREEN_PROPERTY_SOURCE_POSITION,
                                    &pos);
rc = screen_set_event_property_iv(screen_ev,
                                    SCREEN_PROPERTY_SOURCE_SIZE,
                                    &size);
```

f. Set the value of the touch orientation.

```
int touch_orientation = 1;
rc = screen_set_event_property_iv(screen_ev,
                                    SCREEN_PROPERTY_TOUCH_ORIENTATION,
                                    &touch_orientation);
```

g. Set the value of the touch point ID.

The touch point ID is the order of occurence of the `mtouch` event for multi-touch events; multiple `mtouch` events are used to describe one multi-touch event (e.g., a two-finger swipe).

```
int touchId = 1;
rc = screen_set_event_property_iv(screen_ev,
                                    SCREEN_PROPERTY_TOUCH_ID,
                                    &touchId);
```

h. Set the value of the touch pressure.

The amount of pressure on the touch point is a relative value between 0 and (2^{32}-1) where 0 represents the lightest pressure.

```
int touch_pressure = 1;
rc = screen_set_event_property_iv(screen_ev,
                                    SCREEN_PROPERTY_TOUCH_PRESSURE,
                                    &touchId);
```

7. Inject your Screen event.

```
rc = screen_inject_event(screen_disp, screen_ev);
```

8. Release the resources.

```
screen_destroy_event(screen_ev);
screen_destroy_context(screen_ctx);
```

Complete sample: Injecting a screen event

This code sample uses a privileged context to inject a Screen event into a specified display.

```
/*
 * $QNXLicenseC:
 * Copyright 2011, QNX Software Systems Limited. All Rights Reserved.
 *
 * This software is QNX Confidential Information subject to
 * confidentiality restrictions. DISCLOSURE OF THIS SOFTWARE
 * IS PROHIBITED UNLESS AUTHORIZED BY QNX SOFTWARE SYSTEMS IN
 * WRITING.
 *
 * You must obtain a written license from and pay applicable license
 * fees to QNX Software Systems Limited before you may reproduce, modify
 * or distribute this software, or any work that includes all or part
 * of this software. For more information visit
 * http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review
 * this entire file for other proprietary rights or license notices,
 * as well as the QNX Development Suite License Guide at
 * http://licensing.qnx.com/license-guide/ for other information.
 * $
 */

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <screen/screen.h>

int main(int argc, char **argv)
{
 screen_context_t screen_ctx;
 screen_display_t screen_disp;
 screen_display_t *screen_dlist;
 screen_event_t screen_ev;
    int ndisplays;
    int val;
 const char *display = "1";
 int rval = EXIT_FAILURE;
 int rc;
 int i, j;

 for (i = 1; i < argc; i++) {
  if (strncmp(argv[i], "-display=", strlen("-display=")) == 0) {
   display = argv[i] + strlen("-display=");
  } else {
   break;
  }
 }

 rc = screen_create_context(&screen_ctx, SCREEN_INPUT_PROVIDER_CONTEXT);
 if (rc) {
  perror("screen_context_create");
  goto fail1;
 }

 rc = screen_get_context_property_iv(screen_ctx, SCREEN_PROPERTY_DISPLAY_COUNT, &ndisplays);
 if (rc) {
  perror("screen_get_context_property_iv(SCREEN_PROPERTY_DISPLAY_COUNT)");
  goto fail2;
 }

 screen_dlist = calloc(ndisplays, sizeof(*screen_dlist));
 if (screen_dlist == NULL) {
  fprintf(stderr, "could not allocate memory for display list\n");
  goto fail2;
 }

 rc = screen_get_context_property_pv(screen_ctx, SCREEN_PROPERTY_DISPLAYS, (void **)screen_dlist);
 if (rc) {
  perror("screen_get_context_property_pv(SCREEN_PROPERTY_DISPLAYS)");
  free(screen_dlist);
  goto fail2;
 }

 if (isdigit(*display)) {
  int want_id = atoi(display);
  for (j = 0; j < ndisplays; ++j) {
   int actual_id = 0;  // invalid
   (void)screen_get_display_property_iv(screen_dlist[j],
     SCREEN_PROPERTY_ID, &actual_id);
   if (want_id == actual_id) {
```

```
   break;
  }
 }
} else {
 int type = -1;
 if (strcmp(display, "internal") == 0) {
  type = SCREEN_DISPLAY_TYPE_INTERNAL;
 } else if (strcmp(display, "composite") == 0) {
  type = SCREEN_DISPLAY_TYPE_COMPOSITE;
 } else if (strcmp(display, "svideo") == 0) {
  type = SCREEN_DISPLAY_TYPE_SVIDEO;
 } else if (strcmp(display, "YPbPr") == 0) {
  type = SCREEN_DISPLAY_TYPE_COMPONENT_YPbPr;
 } else if (strcmp(display, "rgb") == 0) {
  type = SCREEN_DISPLAY_TYPE_COMPONENT_RGB;
 } else if (strcmp(display, "rgbhv") == 0) {
  type = SCREEN_DISPLAY_TYPE_COMPONENT_RGBHV;
 } else if (strcmp(display, "dvi") == 0) {
  type = SCREEN_DISPLAY_TYPE_DVI;
 } else if (strcmp(display, "hdmi") == 0) {
  type = SCREEN_DISPLAY_TYPE_HDMI;
 } else if (strcmp(display, "other") == 0) {
  type = SCREEN_DISPLAY_TYPE_OTHER;
 } else {
  fprintf(stderr, "unknown display type %s\n", display);
  free(screen_dlist);
  goto fail2;
 }
 for (j = 0; j < ndisplays; j++) {
  screen_get_display_property_iv(screen_dlist[j], SCREEN_PROPERTY_TYPE, &val);
  if (val == type) {
   break;
  }
 }
}

if (j >= ndisplays) {
 fprintf(stderr, "couldn't find display %s\n", display);
 free(screen_dlist);
 goto fail2;
}

screen_disp = screen_dlist[j];
free(screen_dlist);

rc = screen_create_event(&screen_ev);
if (rc) {
 perror("screen_create_event");
 goto fail2;
}

int type = SCREEN_EVENT_MTOUCH_TOUCH;
rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &type);
if (rc) {
 perror("screen_set_event_property_iv(SCREEN_PROPERTY_TYPE)");
 goto fail3;
}

int device = 10;
rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_DEVICE_INDEX, &device);
if (rc) {
 perror("screen_set_event_property_iv(SCREEN_PROPERTY_DEVICE_INDEX)");
 goto fail3;
}

int pos[2] = { 50, 75 };
rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_POSITION, pos);
if (rc) {
 perror("screen_set_event_property_iv(SCREEN_PROPERTY_POSITION)");
 goto fail3;
}

int seq_id = 20;
rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_SEQUENCE_ID, &seq_id);
if (rc) {
 perror("screen_set_event_property_iv(SCREEN_PROPERTY_SEQUENCE_ID)");
 goto fail3;
}

int size[2] = { 5, 5};
rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_SIZE, size);
if (rc) {
 perror("screen_set_event_property_iv(SCREEN_PROPERTY_SIZE)");
 goto fail3;
}

rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_SOURCE_POSITION, pos);
if (rc) {
 perror("screen_set_event_property_iv(SCREEN_PROPERTY_SOURCE_POSITION)");
 goto fail3;
}
```

```
 rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_SOURCE_SIZE, &size);
 if (rc) {
  perror("screen_set_event_property_iv(SCREEN_PROPERTY_SOURCE_SIZE)");
  goto fail3;
 }

 int touch_orientation = 1;
 rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_TOUCH_ORIENTATION, &touch_orientation);
 if (rc) {
  perror("screen_set_event_property_iv(SCREEN_PROPERTY_TOUCH_ORIENTATION)");
  goto fail3;
 }

 int touchId = 3;
 rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_TOUCH_ID, &touchId);
 if (rc) {
  perror("screen_set_event_property_iv(SCREEN_PROPERTY_TOUCH_ID)");
  goto fail3;
 }

 int touch_pressure = 1;
 rc = screen_set_event_property_iv(screen_ev, SCREEN_PROPERTY_TOUCH_PRESSURE, &touch_pressure);
 if (rc) {
  perror("screen_set_event_property_iv(SCREEN_PROPERTY_TOUCH_PRESSURE)");
  goto fail3;
 }

 rc = screen_inject_event(screen_disp, screen_ev);
 if (rc) {
  perror("screen_inject_event");
  goto fail3;
 }

 rval = EXIT_SUCCESS;

fail3:
 screen_destroy_event(screen_ev);
fail2:
 screen_destroy_context(screen_ctx);
fail1:
 return rval;
}
```

Chapter 10
Screen Configuration

This section describes how to configure libraries, drivers, and Screen parameters using the configuration file, `graphics.conf`.

Your target needs access to the files and directories where Screen is installed (usually, under $***QNX_TARGET***). You will need the location of these files and directories when installing and configuring environment variables, configuration parameters, and drivers. In this document, we use *SCREEN-DIR* when we refer to this location.

Some of these steps are platform-dependent. You will need to use the appropriate target-specific configuration input for your platform. We use *TARGET-SPECIFIC* to indicate where platform-dependent configuration input is required.

Configure Screen

This section describes how to configure libraries, drivers, and Screen parameters using the configuration file, `graphics.conf`.

The `graphics.conf` file contains configuration information for Screen and is found under the following directory:

SCREEN-DIR`/usr/lib/graphics/`*TARGET-SPECIFIC*

The `graphics.conf` file is a free-form ASCII text file. It is parsed by Screen or clients. The file may contain extra tabs and blank lines for formatting purposes. Keywords in the file are case-sensitive. Comments may be placed anywhere within the file (except within quotes). Comments begin with the # character and end at the end of the line.

The file is essentially a set of configurable parameters along with the values that these parameters are being set to. The parameters are listed within specific defined sections of the configuration file.

The `graphics.conf` file includes the following main sections:

khronos

Specifies the libraries and parameters related to Khronos (GPU and WFD libraries). The libraries and parameters in this section apply to your EGL display and WFD driver. This section is denoted by the `begin khronos` and `end khronos` statements.

This `khronos` section consists of two sections:

- `egl display`
- `wfd device`

winmgr

Specifies the parameters related to the windowing system. This section is denoted by the `begin winmgr` and `end winmgr` statements. The parameters in this section include those that apply to:

- the windowing system globally
- the displays
- the class (e.g., framebuffers)
- the touch devices

This `winmgr` section consists of a combination of these four sections:

- `globals`

- `display`
- `class`
- `mtouch`

To specify a configuration, you need to follow the format below for the parameter you are configuring within the appropriate section of the configuration file:

`<parameter> = <value>`

Each configuration parameter is on its own separate line. If the parameter allows for multiple values for its configuration, then the values will follow the = after the parameter and will be separated by either a space or a comma. For example,

`<parameter> = <value1>,<value2>,<value3>`

or

`<parameter> = <value1> <value2> <value3>`

A typical `graphics.conf` file will look something like this:

```
begin khronos
  begin egl display 1
    egl-dlls = [IMG%s] libusc.so libsrv_um.so libpvr2d.so libIMGegl.so
    glesv1-dlls = libusc.so libsrv_um.so  libIMGegl.so libImgGLESv1_CM.so
    glesv2-dlls = libusc.so libsrv_um.so libusc.so libIMGegl.so libImgGLESv2.so
    gpu-dlls = libsrv_um.so libpvr2d.so pvrsrv.so
    gpu-string = SGX540rev120
    aperture = 200
  end egl display

  begin wfd device 1
    wfd-dlls = libomap4modes-panda.so libWFDomap4430.so
  end wfd device
end khronos

begin winmgr
  begin globals
    pointer-qsize = 1
    blit-config = pvr2d
    blits-logsize = 4096
    input-logsize = 8192
    requests-logsize = 65536
  end globals

  begin display hdmi
    formats = rgba8888 rgbx8888 nv12
    video-mode = 1280 x 720 @ 60
  end display

  begin class framebuffer
    display = hdmi
    pipeline = 3
    surface-size = 1280 x 720
    format = rgba8888
    usage = pvr2d
  end class

  begin mtouch
    driver = devi
    options = height=720,width=1280
  end mtouch
end winmgr
```

Configure `khronos` section

This section specifies the libraries and parameters related to Khronos (GPU and WFD libraries).

Your `graphics.conf` configuration file must include a Khronos section where your EGL display and WFD driver libraries and parameters are specified.

The Khronos section is identified as the section of the configuration file that is enclosed by `begin khronos` and `end khronos`.

```
begin khronos
  begin egl display 1
    ...
  end egl display

  begin wfd device 1
    ...
  end wfd device
end khronos
...
```

Within the Khronos section:

- libraries and parameters that are related to the EGL display are specified in the section starting with `begin egl display` *`display_id`* and ending with `end egl display`.
- libraries and parameters that are related to the WFD driver are specified in the section starting with `begin wfd device` *`device_id`* and ending with `end wfd device`.

Configure `egl display`

This section specifies the GPU and WFD libraries and parameters.

This section must begin with `begin egl display` *`display_id`* and end with `end egl display`.

Typically there is only one `egl display` section. Therefore, the *display ID* is conventionally set to *1*.

Below is an example of an `egl display` section of a `graphics.conf` file:

```
begin egl display 1
  egl-dlls = [IMG%s] libusc.so libsrv_um.so libpvr2d.so libIMGegl.soD
  glesv1-dlls = libusc.so libsrv_um.so  libIMGegl.so libImgGLESv1_CM.so
  glesv2-dlls = libusc.so libsrv_um.so libusc.so libIMGegl.so libImgGLESv2.so
  gpu-dlls = libsrv_um.so libpvr2d.so pvrsrv.so
  gpu-string = SGX540rev120
  aperture = 200
end egl display
```

This `egl display` section is not related, in any respect, to physical displays or to the `display` configuration section under the `winmgr` section.

Configuration parameters for `egl display`

The following are valid parameters that can be configured under the `egl display` section:

Parameter	Description	Type	Possible Value(s)
egl-dlls	The EGL libraries	string	• `libusc.so` • `libsrv_um.so` • `libpvr2d.so` • `libIMGegl.so`
glesv1-dlls	The OpenGL ES 1.X libraries	string	• `libusc.so` • `libsrv_um.so` • `libIMGegl.so` • `libImgGLESv1_CM.so`
glesv2-dlls	The OpenGL ES 2.X libraries	string	• `libusc.so` • `libsrv_um.so` • `libIMGegl.so` • `libImgGLESv2.so`
gpu-dlls	The GPU libraries	string	• `libsrv_um.so` • `libpvr2d.so` • `pvrsrv.so`
gpu-string	The SGX core. The *gpu-string* will determine core-specific behavior.	string	• `SGX535rev121` • `SGX530rev125` • `SGX540rev120`
aperture	The number of MB of GPU memory to allocate at startup.	integer	Range varies.

Most of these libraries and parameters will be provided in the default `graphics.conf` configuration file delivered with your platform.

Configure `wfd device`

This section specifies the OpenWF Display libraries and parameters.

This section must begin with `begin wfd device` *`device_id`* and end with `end wfd device`.

Typically there is only one `wfd device` section. Therefore, the *device ID* is conventionally set to *1*.

Below is an example of a `wfd device` section of a `graphics.conf` file.

```
begin wfd device 1
  wfd-dlls = libj5modes-evm.so libWFDjacinto5.so
  grpx0 = lcd
  grpx1 = hdmi
  grpx2 = hdmi
  video-layer0 = lcd
  video-layer1 = hdmi
end wfd device
```

The parameters that can be configured under the `wfd device` section are platform-specific. You will need to obtain the valid parameters for your platform by running the following on your platform:

`use libWFD`*`platform`*`.so`

where *platform* can be any one of the following:

- `vesabios`
- `jacinto5`
- `omap4430`
- `omap4460`
- `omap3730`
- `omap35xx`
- `imx6x`

The `use` command will display a message describing the specific parameters and libraries to configure for the particular platform.

Configure `winmgr` section

This section specifies the parameters related to the windowing manager system.

Your `graphics.conf` configuration file must include a `winmgr` section where your `global`, `display`, `class`, and `touch` parameters are specified.

The `winmgr` section is identified as the section of the configuration file that is enclosed by `begin winmgr` and `end winmgr`.

```
...
begin winmgr
  begin globals
  ...
  end globals

  begin display hdmi
  ...
  end display

  begin class framebuffer
  ...
  end class

  begin mtouch
  ...
```

```
  end mtouch
end winmgr
```

Within the `winmgr` section:

- parameters that apply globally to the windowing system are specified in the section starting with `begin globals` and ending with `end globals`; there can only be one `globals` section in your configuration file.
- parameters that are related to the display are specified in the section starting with `begin display` *`display_id`* and ending with `end display`; here may be multiple display sections to correspond to each physical display available and supported by the platform.
- parameters that are related to the class are specified in the section starting with `begin class` *`class_name`* and ending with `end class`; there may be multiple class sections to correspond to each class defined.
- parameters that are related to touch devices are specified in the section starting with `begin mtouch` and ending with `end mtouch`; there may be multiple mtouch sections to correspond to each touch device available and supported by the platform.

Configure `globals`

This section specifies the configuration applied globally to the windowing manager system.

This section must begin with `begin globals` and end with `end globals`.

There is only one `globals` section. The parameters that are set in this section will be applied to all pipelines.

Below is an example of a `globals` section of a `graphics.conf` file.

```
begin globals
  blit-config = pvr2d
  blits-logsize = 4096
end globals
```

Configuration parameters for `globals`

The following are valid parameters that can be configured under the `globals` section:

Parameter	Description	Type	Possible Value(s)
blit-config	The blitter used when your application explicitly calls the native blit API functions (*screen_blit()* and *screen_fill()*).  Only one blitter can be configured. The selection of valid blitters depends on your	string	• `sw` (default) • `pvr2d` • `bv-j5` • `gles2blt`

Parameter	Description	Type	Possible Value(s)
	platform; there may be more blitter values available for your particular platform. This parameter differs from the blitter specified in the *usage* parameter under your `display` section in `graphics.conf`. The blitter specified in the *usage* parameter is the blitter that Screen uses for composition.		
blits-logsize	The size (in bytes) of an internal working ring buffer.	integer	
idle-timeout	The minimum time (in seconds) between any activity before Screen will decide to generate an overdrive event.	long integer	
input	Human interface device (HID). This parameter can be configured with multiple values.  You must ensure that you have `io-hid -dusb` running before starting `screen` if you have a USB input device such as a keyboard, mouse, gamepad, or USB joystick. You must also have all of the following libraries in your ***LD_LIBRARY_PATH***: • `devh-usb.so` • `libusbdi.so.2` • `libhiddi.so.1`	string	• `keyboard` • `mouse` • `gamepad` • `joystick`
input-logsize	The size (in bytes) of an internal working ring buffer.	integer	
keymap	The location and name of the default keymap file. The keymap file specified is used when a new keyboard device is created. This is only applicable to HID keyboards. Although this keymap file can be specified using the Screen context property, SCREEN_PROPERTY_KEYMAP, the location of this file can only be specified using this parameter at time of configuration. This parameter is optional. If keymap is not set, then the default keymap at the default keymap file location will be used.	string	`/shared/keymap/en_US_101`

Parameter	Description	Type	Possible Value(s)
requests-logsize	The size (in bytes) of an internal ring buffer.	integer	
stack-size	The stack size (in units of 1024 bytes) that Screen is to use for its threads. *stack-size* must be configured appropriately for blitters/compositors that are using Mesa (e.g, gles2blt). The default stack size of 4 * 1024 bytes is insufficient for these types of blitters/compositors.	integer	`2048`

Configure `display` *display_id*

This section specifies the configuration applied to the physical displays supported by the platform.

This section must begin with `begin display` *`display_id`* and end with `end display`.

There can be multiple `display` sections within a configuration file. The number of `display` sections depends on the number of physical displays supported by the platform.

The *display_id* is used to identify the display to which the `display` section pertains. The *display_id* can be a number identifying the display or it can be the connection type of the display. If the *display_id* is an integer, Screen will apply the configuration parameters to the display corresponding to the integer specified as the *display_id*. Otherwise, if the configuration is a string that matches one of the valid display connection types, then Screen will apply the configuration parameters to the first available display whose connection type matches that specified as the *display_id*.

Below is an example of a `display` section of a `graphics.conf` file. In this example, the *display_id* is a connection type of *hdmi*; therefore, the configuration parameters will be applied to the first available display that supports *hdmi*.

```
begin display hdmi
  formats = rgba8888 rgbx8888 nv12
  video-mode = 1280 x 720 @ 60
end display
```

Configuration parameters for `display`

The following are valid parameters that can be configured under the `display` section:

Parameter	Description	Type	Possible Value(s)
background	The background color of the display. Use RGB color code (HEX value) to identify the color. (e.g., blue = 0xff)	unsigned long integer (hex)	• `0x00` (default: black)

Parameter	Description	Type	Possible Value(s)
cbabc	Content-based automatic brightness control; specifies the content type of the display	string	**`none`** The display content is not video, UI, or photo. **`video`** The display content is video. **`ui`** The display content is UI. **`photo`** The display content is photo.
cursor	The visibility of the cursor on the display	string	**`auto` (default)** The cursor will remain visible on the display until there is 10 seconds of cursor inactivity. After that, the cursor will be made invisible on the display. **`on`** The cursor will always remain visible on the display, regardless of cursor inactivity. **`off`** The cursor will always remain invisble on the display regardless of cursor activity.
formats	The pixel format(s) supported by the display. This parameter can be configured with multiple values.	string	• `byte` • `rgba4444` • `rgbx4444` • `rgba5551` • `rgbx5111`

Parameter	Description	Type	Possible Value(s)
			• `rgb565` • `rgb888` • `rgba8888` • `rgbx8888` • `yvu9` • `yuv420` • `nv12` • `yv12` • `uyvy` • `yuy2` • `v422` • `ayuv`
gamma	The gamma value of the WFD driver. The range for this gamma value is specific to the driver.	unsigned long integer	`0..255`
idle-timeout	The amount of time (in seconds) after which the display will enter an idle state.	string or unsigned long integer	• `off` • `0..ULONG_MAX`
mirroring	The mirror types	string	**`disabled`** Mirroring is disabled. **`normal`** Mirroring is enabled, and the aspect ratio of the image is 1:1. **`stretch`** Mirroring is enabled, and the image should fill the display while not preserving the aspect ratio. **`zoom`** Mirroring is enabled, and the image should fill the display while preserving the aspect

Parameter	Description	Type	Possible Value(s)
			ratio. Image content may be clipped. **`fill`** Mirroring is enabled, and the image should fill the display while preserving the aspect ratio. Image may be shown with black bars where applicable.
priority	The priority of the update thread (i.e., the thread that renders the framebuffer(s)).	integer	Default: `15`
rotation	The clockwise rotation of a display. Display rotation is absolute.	long integer	• `0` • `90` • `180` • `270`
rotation-mode	Your preference of display rotation mode. If the mode of rotation specified isn`t supported by your display controller, the default is to use blitter rotation.	string	**`blits`** Rotated blits with pipeline rotation where possible **`generic`** Generic modes can resize **`none`** No rotation is supported **`port`** Port rotation, framebuffer resizes **`pipeline`** Pipeline rotation, framebuffer resizes
splash	Indicates whether or not to post framebuffer on start-up. This parameter is	integer	**`0`(default)**

Parameter	Description	Type	Possible Value(s)
	considered only if the WFD driver supports it.		Indicates to post the framebuffer immediately (blank screen). **1** Indicates to not post the framebuffer until the application requests a post.
stack-size	The stack size (in units of 1024 bytes) that Screen is to use for its threads. *stack-size* must be configured appropriately for blitters/compositors that are using Mesa (e.g, gles2blt). The default stack size of 4 * 1024 bytes is insufficient for these types of blitters/compositors.	integer	`2048`
touch-adjustments	The x and y adjustments to be added to all touch events. This configuration must be in the form: *x-adjustment,y-adjustment*	unsigned long integer	
video-mode	The initial resolution and refresh rate for the display port. This configuration must be one that is reported by the WFD driver. If you set this to values that aren't supported, then the resolution and refresh rate will default to the first mode specified by the driver. The resolution and refresh rate must be in the form of: *width*x*height* *@[i]refresh* For example, `video-mode = 1280x720@60`. If *i* is indicated, then interlacing is set.	unsigned long integer	
protection-enable	Content protection for the window(s) on the display. Typcially this is set only if the application is interested in HDCP (High-Bandwidth Digital Content Protection).	string	• `true` • `false`

Configure `class`

This section specifies the default values for window properties.

This section must begin with `begin class` *`class_name`* and end with `end class`.

The `class` section is used to set default values for the window properties defined within the section; these properties are specified through parameters applied to framebuffers and application windows.

Framebuffers

The number of `class` sections that can be defined for framebuffers depends on the number of pipelines available. Generally, one framebuffer is defined per pipeline.

To specify that a `class` section is used to configure default values for a framebuffer, the follwing convention is used:

```
framebuffer<unique_string>
```

The *class_name* of the `class` section must start with the string *framebuffer* followed by any unique string to identify the class. For example, some valid *<class name>* strings are:

- *framebuffer*
- *framebuffer1*
- *framebufferA*
- etc.

Below is an example of using multiple `class` sections to specify default vaules for two framebuffers, each for a specific pipeline:

```
begin class framebuffer1
  display = 1
  pipeline = 1
  format = rgba8888
  usage = pvr2d
  id_string = fb1
end class

begin class framebuffer2
  display = 2
  pipeline = 2
  format = rgba8888
  usage = pvr2d
  id_string = fb2
end class
```

Application windows

There's no explicit limit on the number of `class` sections that can be defined for configuring application windows.

To specify that a `class` section is used to configure default values for an application window, the *class_name* of the `class` section must be a unique string. This string needs to match the window property, *SCREEN_PROPERTY_CLASS*, you set in your application.

Below is an example of using a `class` section to specify defaults for an application window:

```
begin class my_app_win
  visible = true
  surface-size = 640x480
  source-position = 0,0
  source-size = 640x480
  window-position = 0,0
  window-size = 640x480
  id_string = MY_APP_WINDOW
end class
```

If you use the above `class` section in your configuration file, then you can simply set the *SCREEN_PROPERTY_CLASS* window property in your application code. The setting of this property will trigger Screen to apply the configured values that are associated with that class to your application window.

For example, if you use the above `class` section in your configuration file, you can use this in your application code:

```
...
const char my_win_class = "my_app_win";
const int len = strlen(my_win_class);
screen_set_window_property_cv(screen_win, SCREEN_PROPERTY_CLASS, len, my_win_class);
...
```

instead of this:

```
...
const int visible = 1
const char id_string = "MY_APP_WINDOW";
const int len = strlen(id_string);
const int buffer_size[2] = { 640, 480};
const int src_pos[2] = { 0, 0};
const int src_size[2] = { 640, 480};
const int win_pos[2] = { 0, 0};
const int win_size[2] = { 640, 480};
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_VISIBLE, &visible);
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_BUFFER_SIZE, buffer_size);
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_POSITION, src_pos);
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SOURCE_SIZE, src_size);
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_POSITION, win_pos);
screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SIZE, win_size);
screen_set_window_property_cv(screen_win, SCREEN_PROPERTY_ID_STRING, len, id_string);
...
```

to set your window properties.

Configuration parameters for `class`

The following are valid parameters that can be configured under the `class` section; all parameters are valid for both framebuffers and application windows.

Parameter	Description	Type	Possible Value(s)
alpha-mode	Indication of method of alpha blending to use for alpha compositing	string	**`non-pre-multiplied`** **(default)**

Parameter	Description	Type	Possible Value(s)
			Uses straight RGBA pixel value. For example, (0, 1, 0, 0.5) is green with 50% opacity **`pre-multiplied`** Uses pre-multiplied RGBA pixel value. For example, a non-pre-multiplied value of (0, 1, 0, 0.5) can be intepreted as (0, 1,0) * 0.5, giving a pre-multiplied value of (0, 0.5, 0, 0.5), which 0.5 is 100% green intensity but with 50% opacity
brightness	The brightness of a window. This configuration depends on the hardware; it can be configured, but may or may not take effect as determined by your hardware.	integer	`-255..255`
cbabc	Content-based automatic brightness control; specifies the content type of a window. This configuration depends on the hardware; it can be configured, but may or may not take effect as determined by your hardware.	string	• `none` • `video` • `ui` • `photo`
clip-position	The x- and y- position of a clipped rectangular viewport within the window buffers. This configuration must be in the form of: *x-position,y-position*. For example, `clip-position = 100,100`.	integer	*`x-position, y-position`*
clip-size	The width and height (in pixels) of a clipped rectangular viewport within the window buffers. This configuration must be in the form of: *width*x*height* For example, `clip-size = 100x100`.	integer	*`width`*`x`*`height`*
color	The background color of a window. Use RGB color code (HEX value) to identify the color. (e.g., blue = 0xff).	unsigned long	• `0x00` (default: black)

Parameter	Description	Type	Possible Value(s)
		integer (hex)	
contrast	The contrast adjustment of a window. This configuration depends on the hardware; it can be configured, but may or may not take effect as determined by your hardware.	integer	`-128..127`
display	The connection type to the display or the display on which the window will be shown. If the configuration is an integer, screen will interpret the integer as the indication of the display on which the window will be shown. Otherwise, if the configuration is a string that matches one of the valid display connection types, then Screen will interpret the string as the configuration of the display connection type.	string or integer	• `internal` • `composite` • `svideo` • `YPbPr` • `rgb` • `rgbhv` • `dvi` • `hdmi` • `1..2`
format	The pixel format used by a window. Only one format is expected; this format must be supported by the display.	string	• `byte`(default) • `rgba4444` • `rgbx4444` • `rgba5551` • `rgbx5111` • `rgb565` • `rgb888` • `rgba8888` • `rgbx8888` • `yvu9` • `yuv420` • `nv12` • `yv12` • `uyvy` • `yuy2` • `v422` • `ayuv`
global-alpha	The global alpha mode to apply to a window.	integer	`0..225`
hue	The hue adjustment of a window.	integer	`-128..127`

Parameter	Description	Type	Possible Value(s)
id_string	A string to identify the contents of a window.	string	
order	The distance from the bottom that is used when ordering window groups among each other.	long integer	
pipeline	The pipeline identifier. This must refer to a pipeline defined in the `wfd device` section of the configuration file.	unsigned long integer	`1..3`
rotation	The clockwise rotation of a window. Window rotation is absolute.	long integer	• `0` • `90` • `180` • `270`
saturation	The saturation adjustment of a window.	long integer	`-128..127`
source-position	The x and y coordinates of the top-left corner of a rectangular region within the window buffer representing the source viewport of a window. This is the portion of the window buffer that is to be displayed. This configuration must be in the format of: *x-position,y-position*. For example, `source-position = 100,100`.	integer	*`x-position, y-position`*
source-size	The width and height (in pixels) of a region within the window buffer representing the source viewport of the window. This is the portion of the window buffer that is to be displayed. This configuration must be in the form of: *width*x*height*. For example, `source-size = 100x100`.	integer	
static	Indicates whether or not the contents of a window are expected to change	unsigned long integer	**`0`** Window content isn't static **`1`** Window content is static
surface-size	The width and height of the window buffer. This configuration must be in the form of: *width*x*height* For example, `surface-size = 100x100`.	integer	

Parameter	Description	Type	Possible Value(s)
interval	The minimum number of vsync periods between posts.	unsigned long integer	
usage	The intended usage for the window buffers; the resulting usage applied to the window buffers is the bitwise *OR* of all the valid entries configured with this parameter. You can configure this parameter using any of the following: • one or more valid usage flags (For complete listing of valid usage flags, refer to Screen usage flag types . • the blitter to use for composition onto a framebuffer (the default is `sw` if no blitter is specified) • one or more integers representing a valid usage bit If this configuration consists of a string that is not mapped to one of the valid usage flags, then Screen will interpret the string as the blitter to use for composition onto a framebuffer. Note that all framebuffers share the same blitter; therefore if you configure multiple `class` sections and define the *usage* with a blitter in each, then the blitter named in the *usage* of the last `class` section in your configuration file will be used. This parameter differs from the blitter specified in the *blit-config* parameter under your `globals` section in `graphics.conf`. The blitter specified in the *blit-config* parameter is the blitter that Screen uses when your application explicitly calls the native blit API functions (*screen_blit()* and *screen_fill()*). When multiple usage flags are configured, each flag in the configuration must be separated by a comma or space. You can combine the blitter name with the usage flags in the same configuration line as long as it is separated by a comma or a space.	string	**`gles1`** Used to indicate that OpenGL ES 1.X is used for rendering the buffer(s) associated with the window **`gles2`** Used to indicate that OpenGL ES 2.X is used for rendering the buffer(s) associated with the window **`native`** Used to indicate that buffer(s) associated with the window can be used for native API operations. For example, if your application explicitly calls (*screen_blit()* and/or *screen_fill()*), then this flag must be set. **`rotation`** Used to indicate that the buffer(s) associated with the window can be re-configured from landscape to portrait

Parameter	Description	Type	Possible Value(s)
	For example, the following are examples of valid entries for configuring *usage*: • usage = pvr2d • usage = sw native • usage = 2 4 8		orientation without having the need for reallocation. **sw** Used to indicate that the buffer(s) associated with the window can be read from and written to. **vg** Used to indicate that OpenVG is used for rendering the buffer(s) associated with the window
visible	Indicates whether or not a window is visible.	string	• `true` • `false`
window-position	The x and y positions of the window screen coordinates. Remember that the position of child and embedded windows are relative to the parent window. For example, if the position of the application window is (10, 10) and the position of the child window is (10, 10), then the position of the child window on the screen is actually (20, 20). This configuration must be in the form or: *x-position*,*y-position*. For example, `window-position = 10,10`.	integer	*`x-position,y-position`*
window-size	The width and height (in pixels) of a window. This configuration must be in the form of: *width*x*height*. For example, `window-size = 100x100`.	integer	*`width`*x*`height`*
buffer-count	The number of buffers that are to be created or attached to a window. Beyond configuration, the buffer count can't be set; it can only be queried by the application. There is no explicit limit for this number.	integer	3 (default)

Configure `mtouch`

This section specifies the configuration applied to the touch devices supported by the platform.

This section must begin with `begin mtouch` and end with `end mtouch`.

There can be multiple `mtouch` sections within a configuration file. The number of `mtouch` sections depends on the number of physical displays supported by the platform. If you have mutliple displays, you can configure the driver for each display; therefore you would have one `mtouch` section for each of these displays.

From within the `mtouch` section of the configuration file, filters that are related to the mtouch devices are specified in the section starting with `begin filter` and ending with `end filter`.

Below is an example of multiple `mtouch` sections of a `graphics.conf` file:

```
begin mtouch
      driver = devi
      options = height=720,width=1280
      scaling = /armle-v7/usr/lib/graphics/omap4430/scaling_omap4.conf
      display = 1
end mtouch

  begin mtouch
      driver = egalax
      options = height=800,width=480
      scaling = /armle-v7/usr/lib/graphics/omap4430/scaling_omap4.conf
      display = 2
end mtouch
```

Once `screen` is running, run the `calib-touch` utility with the `-display` option from either your startup script or from a shell to calibrate your touchscreens.

scaling.conf

The `scaling.conf` file is a free-form ASCII text file. It is parsed by Screen. The file may contain extra tabs and blank lines for formatting purposes. Keywords in the file are case-sensitive. Comments may be placed anywhere within the file (except within quotes). Comments begin with the # character and end at the end of the line.

You must define a mode for the touch coordinates using a `scaling.conf` configuration file; you can define only one mode per configuration. The mode recommended is the `scale` mode configured with the resolution of your display. For example, your entry in the scaling.conf file would be:

```
1280x720:mode=scale
```

By default, Screen expects scaling.conf file at this location:

```
/etc/system/config/scaling.conf
```

If your scaling.conf file is not located at the above location or is not named `scaling.conf`, you will need to indicate the correct path and name of your scaling configuration file using the *scaling* parameter in your `mtouch` configuration section.

Configuration parameters for `mtouch`

The following are valid parameters that can be configured under the `mtouch` section:

Parameter	Description	Type	Possible Value(s)
driver	The touch driver loaded by Screen. If you configure your driver to be `devi`, you must ensure that you have the following processes running before starting `screen`: • `devi-hid -R` *width*`,` *height* `touch` where *width* and *height* define the resolution of your display. (e.g., `devi-hid -R1280,720 touch`)) • `io-hid -dusb` (this is the case for configuring a multitouch display that uses USB for the control.) For more information about the `devi-hid` and `io-hid` utilities, see its entry in the OS *Utilities Reference*.  To be able to successfully run `io-hid` and `devi-hid`, you must ensure that you have the following libraries in your ***`LD_LIBRARY_PATH`*** environment variable: • `devh-usb.so` • `libusbdi.so.2` • `libhiddi.so.1` • `libmtouch-devi.so` Use the following command on your target to get more information on the options when running `devi-hid`: `use devi-hid` There is no such requirement if you are using the `egalax` driver, as this driver will communicate directly with the hardware.	string	• `devi` • `egalax`
options	The options configuration is driver-dependent. The string you configure is passed verbatim to the driver itself. It can be used to set the height	string	

Parameter	Description	Type	Possible Value(s)
	and width (in pixels) of a rectangular mtouch surface. Typcially, you set this to your display resolution (i.e., the configuration you used for the *video-mode* under the `display` section). This configuration must be in the form of: `height=`*display_height* `,width=`*display_width*. For example, `options = height=720,width=1280`.		
scaling	The full pathname of the server-side scaling configuration file that defines the scaling to apply to the touch coordinates. For an example of what a scaling configuration file must look like, see *Sample scaling.conf configuration file* . You must define a mode for the touch coordinates using a `scaling.conf` configuration file; you can define only one mode per configuration. The mode recommended is the `scale` mode configured with the resolution of your display. For example, your entry in the scaling.conf file would be: `1280x720:mode=scale`	string	`NULL`(default)
min_event_interval	The sampling rate (in microseconds) of the touch controller. You will not get two touch samples within a time less than that configured here. The effectiveness of this configuration is driver-dependent.	unsigned long integer	
display	The connection type to the display that is associated with the mtouch device. If the configuration is an integer, Screen will interpret the integer as the identifier of the display to be associated with the mtouch device; otherwise, if the configuration is a string that matches one of the valid display connection types, then Screen will associate the first display of that type with the mtouch device.	string or integer	• `internal` • `composite` • `svideo` • `YPbPr` • `rgb` • `rgbhv` • `dvi` • `hdmi` • `1..2`

Configuring `mtouch filter`

This section specifies the configuration applied to the filters of mtouch devices supported by the platform.

This section must begin with `begin filter` and end with `end filter`.

Below is an example of a `mtouch` section of a `graphics.conf` file.

```
begin filter
```

```
        type = 1
        options =
    end filter
```

Configuration parameters for `filter`

The following are valid parameters that can be configured under the `filter` section:

<table>
<tr><th>Parameter</th><th>Description</th><th>Type</th><th>Possible Value(s)</th></tr>
<tr><td>type</td><td>The series of filters that are applied to the touch coordinates.</td><td>integer</td><td><code>ballistic</code>
Minimizes the noise normally seen on successive touch events that follow a low-speed ballistic trajectory. The lower the speed of the ballistic movement (e.g., a stationary finger), the more gain reduction is applied to the speed of the movement. By doing so, the result is a solid touch with no noise.
<code>edge-swipe</code>
Detects the passing of a finger over the edge of the touch surface. This lets you get swipe gestures starting from outside the screen.
<code>kalman</code>
Minimizes the tracking noise of a moving finger on a given touch surface by use of the Kalman filter algorithm.
<code>bezel-touch</code>
<code>doa</code></td></tr>
<tr><td>options</td><td>Filter specific options. The available filter options depend on the filter type. The format for configuring filter options is: <code>options= <filter option>=<value></code> or, if there are multiple options to specify, then each option-value pair will be separated by a comma: <code>options= <filter option1>=<value>,<filter option2>=<value></code></td><td>string</td><td>Options are listed below per filter type:
• ballistic
<table><tr><th>Filter option</th><th>Description</th><th>Default</th></tr><tr><td><code>scale</code></td><td>FP scale factor</td><td>256</td></tr><tr><td><code>min_gain</code></td><td>Minimal gain</td><td>8</td></tr><tr><td><code>low_speed</code></td><td>Low speed threshold</td><td>32</td></tr></table>• edge_swipe_detect</td></tr>
</table>

<table>
<tr><th>Parameter</th><th>Description</th><th>Type</th><th>Possible Value(s)</th></tr>
<tr><td></td><td></td><td></td><td>
Table 1: Filter order
<table>
<tr><th>Filter option</th><th>Description</th><th>Default</th></tr>
<tr><td>filter_order</td><td>Filter Order</td><td>2</td></tr>
</table>
Table 2: Left edge (x=0)
<table>
<tr><th>Filter option</th><th>Description</th><th>Default</th></tr>
<tr><td>xl_enable</td><td>Apply detection</td><td>1</td></tr>
<tr><td>xl_bezel</td><td>Bezel width</td><td>50</td></tr>
<tr><td>xl_speed</td><td>Speed threshold</td><td>25</td></tr>
<tr><td>xl_jitter</td><td>Jitter control</td><td>0</td></tr>
<tr><td>xl_reject_physical_bezel</td><td>Reject physical bezel</td><td>0</td></tr>
</table>
Table 3: Right edge (x=max(raw_x))
<table>
<tr><th>Filter option</th><th>Description</th><th>Default</th></tr>
<tr><td>xh_enable</td><td>Apply detection</td><td>1</td></tr>
<tr><td>xh_bezel</td><td>Bezel width</td><td>50</td></tr>
<tr><td>xh_speed</td><td>Speed threshold</td><td>25</td></tr>
<tr><td>xh_jitter</td><td>Jitter control</td><td>0</td></tr>
<tr><td>xh_reject_physical_bezel</td><td>Reject physical bezel</td><td>0</td></tr>
</table>
Table 4: Top edge (y=0)
<table>
<tr><th>Filter option</th><th>Description</th><th>Default</th></tr>
<tr><td>yl_enable</td><td>Apply detection</td><td>1</td></tr>
<tr><td>yl_bezel</td><td>Bezel width</td><td>80</td></tr>
<tr><td>yl_speed</td><td>Speed threshold</td><td>25</td></tr>
<tr><td>yl_jitter</td><td>Jitter control</td><td>0</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Parameter</th><th>Description</th><th>Type</th><th>Possible Value(s)</th></tr>
<tr><td></td><td></td><td></td><td>
<table>
<tr><th>Filter option</th><th>Description</th><th>Default</th></tr>
<tr><td>yl_reject_physical_bezel</td><td>Reject physical bezel</td><td>0</td></tr>
</table>
Table 5: Bottom edge (y=max(raw_y))
<table>
<tr><th>Filter option</th><th>Description</th><th>Default</th></tr>
<tr><td>yh_enable</td><td>Apply detection</td><td>1</td></tr>
<tr><td>yh_bezel</td><td>Bezel width</td><td>50</td></tr>
<tr><td>yh_speed</td><td>Speed threshold</td><td>25</td></tr>
<tr><td>yh_jitter</td><td>Jitter control</td><td>0</td></tr>
<tr><td>yh_reject_physical_bezel</td><td>Reject physical bezel</td><td>0</td></tr>
</table>
<hr>

edge-swipe filter options nomenclature:

xl: x-low border, near x=0;
xh: x-high border, near x=maximal X touch coordinate;
yl: y-low border, near y=0;
yh: y-high border, near y=maximal Y touch coordinate;

<hr>

kalman

Table 6: Noise variance
<table>
<tr><th>Filter option</th><th>Description</th><th>Default</th></tr>
<tr><td>proc_noise_x_var</td><td>Process noise variance (X)</td><td>32</td></tr>
<tr><td>proc_noise_y_var</td><td>Process noise variance (Y)</td><td>32</td></tr>
<tr><td>meas_noise_x_var</td><td>Measurement noise variance(X)</td><td>100</td></tr>
<tr><td>meas_noise_y_var</td><td>Measurement noise variance (Y)</td><td>100</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Parameter</th><th>Description</th><th>Type</th><th>Possible Value(s)</th></tr>
<tr><td></td><td></td><td></td><td>
Table 7: Adaptive speed threshnold
<table>
<tr><th>Filter option</th><th>Description</th><th>Default</th></tr>
<tr><td><code>slot_threshold_1</code></td><td>Adaptive speed threshold 1</td><td>30</td></tr>
<tr><td><code>slot_threshold_2</code></td><td>Adaptive speed threshold 2</td><td>20</td></tr>
<tr><td><code>slot_threshold_3</code></td><td>Adaptive speed threshold 3</td><td>10</td></tr>
<tr><td><code>slot_threshold_4</code></td><td>Adaptive speed threshold 4</td><td>5</td></tr>
</table>
bezel_touch
<table>
<tr><th>Filter option</th><th>Description</th><th>Default</th></tr>
<tr><td><code>parrallel_threshold</code></td><td>Filter parallel threshold</td><td>140</td></tr>
<tr><td><code>perpendicular_threshold</code></td><td>Filter perpendicular threshold</td><td>70</td></tr>
</table>
</td></tr>
</table>

Apply your Screen configuration

The following procedure describes how to apply your Screen configuration to a platform that is running QNX Neutrino RTOS.

Prior to starting this procedure, ensure the following:

- Your target hardware is running QNX Neutrino RTOS.
- You can run a shell and commands such as `pidin`.
- If applicable, you have already installed the most recent compatible graphics patch for your platform and have verified that Screen applications can run successfully.
- You have modified `graphics.conf` with your desired configuration parameters.

To apply your Screen configuration:

1. Stop `screen` by using the following command:

   ```
   slay screen
   ```

2. Verify that your `screen` process has stopped. Run:

   ```
   pidin ar
   ```

and, verify that the `screen` process is not running.

3. Set the ***GRAPHICS_ROOT*** variable to include the appropriate graphics directory:

   ```
   export GRAPHICS_ROOT=SCREEN-DIR/usr/lib/graphics/TARGET-SPECIFIC
   ```

   For example, if you are using an OMAP3 board with the `screen` directory location mounted to `/`, you would need to set your ***GRAPHICS_ROOT***_ to be:

   ```
   export GRAPHICS_ROOT=/armle-v7/usr/lib/graphics/omap3
   ```

4. Set your ***LD_LIBRARY_PATH*** environment variable to include the directories of the shared libraries that Screen needs.

```
export LD_LIBRARY_PATH=SCREEN-DIR/usr/lib:SCREEN-DIR/lib:SCREEN-DIR/lib/dll:$GRAPHICS_ROOT:$LD_LIBRARY_PATH
```

   For example, if you are using VMware as the target, you would need to set your ***LD_LIBRARY_PATH*** to be:

   ```
   export LD_LIBRARY_PATH=/usr/lib:/lib:/lib/dll:$GRAPHICS_ROOT:$LD_LIBRARY_PATH
   ```

5. Optional: If you are configuring mtouch and using the *devi* driver, ensure that you have `io-hid` and `devi-hid` running. Run:
   `pidin ar`

   From the output of `pidin ar`, verify that `io-hid` and `devi-hid` are running.

   See an example display of the proceses that could be running on your platform:

```
     pid Arguments
       1 procnto-smp-instr -v
    4098 devc-seromap -e -p -F -b115200 -c48000000/16
0x48020000^2,106
    4099 slogger
    4100 pipe
    4101 i2c-omap35xx-omap4 -p 0x48070000 -i 88
    4102 i2c-omap35xx-omap4 -p 0x48072000 -i 89
    4103 i2c-omap35xx-omap4 -p 0x48060000 -i 93
    4104 i2c-omap35xx-omap4 -p 0x48350000 -i 94
    8201 io-audio -d omap4pdm
   16394 devb-mmcsd-blaze cam silent blk noatime,cache=8m
mmcsd
ioport=0x4809c100,ioport=0x4a056000,irq=115,dma=30,dma=61,dma=62
 dos exe=all
   16395 spi-master -u 0 -d omap4430
base=0x48098100,irq=97,sdma=0
   16396 spi-master -u 3 -d omap4430
base=0x480ba100,irq=80,sdma=0
   16397 io-usb -domap4430-mg ioport=0x4a0ab000,irq=124
-dehci-omap3 ioport=0x4a064c00,irq=109
   20494 io-pkt-v4 -ptcpip -dsmsc9500 mac=004460515624
   24591 dhcp.client
   32784 devc-pty
   32786 inetd
   36881 qconn
   36883 sh
   73748 fs-cifs 10.222.109.24:/test /t screen
  176149 io-hid -dusb
```

```
  180246 devi-hid -PrR1280 touch
  184343 pidin ar
```

To confirm that `io-hid` is running with `devh-usb.so`, you can run the command:

```
pidin -p io-hid mem
```

```
pid tid  name                 prio STATE           code
data   stack
176149   1 t/bin/io-hid       10r SIGWAITINFO      36K
124K   24K(516K)*
176149   2 t/bin/io-hid       21r RECEIVE          36K
124K   4096(12K)
176149   3 t/bin/io-hid       10r RECEIVE          36K
124K   4096(20K)
176149   5 t/bin/io-hid       10r REPLY            36K
124K   4096(20K)
176149   6 t/bin/io-hid       10r RECEIVE          36K
124K   4096(20K)
            libc.so.3         @ 1000000            460K
 16K
            libhiddi.so.1     @78000000             32K
4096
            devh-usb.so       @78009000             16K
4096
            libusbdi.so.2     @7800e000             40K
8192
```

6. Restart `screen` by using the following command:

   SCREEN-DIR`/sbin/screen`

7. Verify that there were no warnings generated from your new configuration by using the following command:

   ```
   sloginfo
   ```

Troubleshooting

Here are some common problems you might encounter after applying your configuration.

Why is `screen` (or my Screen application) not running?

It's possible that there was an invalid configuration parameter set or that some of the libraries required for your configuration could not be found. There are two ways to help find out why `screen` or your Screen application is not running:

1. Ensure that `screen` is no longer running on your platform by checking the processes that are running; use the following command:

   ```
   pidin ar
   ```

   From the output of `pidin ar`, look for the `screen` process.

```
# pidin ar
     pid Arguments
       1 procnto-smp-instr -v
    4098 devc-seromap -e -p -F -b115200 -c48000000/16
0x48020000^2,106
    4099 slogger
    4100 pipe
    4101 i2c-omap35xx-omap4 -p 0x48070000 -i 88
    4102 i2c-omap35xx-omap4 -p 0x48072000 -i 89
    4103 i2c-omap35xx-omap4 -p 0x48060000 -i 93
    4104 i2c-omap35xx-omap4 -p 0x48350000 -i 94
    8201 io-audio -d omap4pdm
  16394 devb-mmcsd-blaze cam silent blk noatime,cache=8m
 mmcsd
ioport=0x4809c100,ioport=0x4a056000,irq=115,dma=30,dma=61,dma=62
 dos exe=all
   16395 spi-master -u 0 -d omap4430
base=0x48098100,irq=97,sdma=0
   16396 spi-master -u 3 -d omap4430
base=0x480ba100,irq=80,sdma=0
   16397 io-usb -domap4430-mg ioport=0x4a0ab000,irq=124
 -dehci-omap3 ioport=0x4a064c00,irq=109
   20494 io-pkt-v4 -ptcpip -dsmsc9500 mac=004460515624
   24591 dhcp.client
   32784 devc-pty
   32786 inetd
   36881 qconn
   36883 sh
   57364 fs-cifs 10.222.109.24:/test /t screen
 5799960 /t/650SDP/B999/daily/nto/armle-v7/sbin/screen
 6815765 pidin ar
```

2. If `screen` is running, slay the `screen` process by using the following command:

   ```
   slay screen
   ```

   Use `pidin ar` again to confirm that the screen process has shut down. It is possible that you will have to slay `screen` more than once before the process shuts down.

3. Set up your environment for debugging; you can do either one or both of the following:

   - **Option 1:** Set the ***LD_DEBUG*** environment variable to use debugging symbols:

     ```
     export LD_DEBUG=libs
     ```

     This will display information on missing libraries, if any, when you are running `screen` or your Screen application.
   - **Option 2:** Use `sloginfo` by first clearing the logs with the command: `sloginfo -c` and then starting `sloginfo` with the `-w` option: `sloginfo -w &`. The `-w` option will display logs from `sloginfo` as events arrive.

4. Restart `screen` by using the following command:

   SCREEN-DIR`/sbin/screen`

5. If you had trouble running your Screen application, try running it again; you should be able to see if the problem involves missing libraries.

Why do I get only a solid colored screen when I run my OpenGL ES 2.x application on my i.MX 6 board?

It's likely that, due to a known issue, you don't have `libGLSLC.so` in your ***`LD_LIBRARY_PATH`*** environment variable.

If running *SCREEN-DIR*`/usr/bin/gles1-gears` works, but running *SCREEN-DIR*`/usr/bin/gles2-gears` doesn't, then chances are you'll need to do the following:

1. Stop `screen` by using the following command:

   ```
   slay screen
   ```

2. Copy `libGLSLC.so` from

   SCREEN-DIR`/usr/lib/graphics/`*TARGET-SPECIFIC*

   to

   SCREEN-DIR`/usr/lib/`

   For example, for an i.MX 6 board with the `screen` directory location mounted to `/`, the command would be:

   ```
   cp /usr/lib/graphics/imx6/libGLSLC.so /usr/lib/
   ```

3. Restart `screen` by using the following command:

   SCREEN-DIR`/sbin/screen`

4. Try running your OpenGL ES 2.x application again.

Why is `screen` (or my Screen application) not displaying what I'm expecting?

There can be many reasons why what you see on the display isn't what you're expecting. It's possible that some of your configuration values aren't compatible. The best way to confirm that you're using the correct configuration values, or that Screen is using the values you are expecting, is to look at the files under `/dev/screen/`.

Under `/dev/screen/` on your target, you can use the following files to help you see what values Screen is applying to your application at the time you `cat` the file.

`/dev/screen/0/dpy-`*`display_id`*

The configuration for each of your displays (e.g., `/dev/screen/0/dpy-1`).

`/dev/screen/0/win-`*`window_id`* **`/win-`*`window_id`***

The configuration for each of your framebuffers (e.g., `/dev/screen/0/win-0/win-0`).

`/dev/screen/0/win-`*`window_id`* **`/win-`*`window_id`*`-bmp_id.bmp`**

The bitmap of what's drawn into your framebuffer at the moment when you touch this file (e.g., `/dev/screen/1564694/win-1/win-1-1.bmp`, `/dev/screen/1564694/win-1/win-1-2.bmp`, etc.).

`/dev/screen/`*`pid`* **`/win-`*`window_id`*** **`/win-`*`window_id`***

The configuration for each of your windows (e.g., `/dev/screen/1564694/win-1/win-1`).

`/dev/screen/`*`pid`* **`/win-`*`window_id`*** **`/win-`*`window_id`*`-bmp_id.bmp`**

The bitmap of what is drawn into your window buffers at the moment when you touch these files (e.g., `/dev/screen/1564694/win-1/win-1-1.bmp`, `/dev/screen/1564694/win-1/win-1-2.bmp`, etc.).

Here are examples of what these files may look like:

- *`/dev/screen/0/dpy-1`* (p. 172)
- *`/dev/screen/0/win-0/win-0`* (p. 174)
- *`/dev/screen/1564694/win-1/win-1`* (p. 175)

You can capture all relevant information to your current Screen configuration by doing the following from your target:

1. Make a copy of your /dev/screen directory into your file system; for example:

   ```
   cp -r /dev/screen /dev-screen-copy
   ```

2. Tar the copied directory to capture the status of your system:

   ```
   tar czvf screen-log.tgz /dev-screen-copy
   ```

Sample `/dev/screen/0/dpy-1` file

This file lists the display configurations that Screen is using.

The `dpy-1` file in `/dev/screen/0/` shows the configurations of the display that Screen is currently using.

From your target, you can see the display configurations that Screen is currently using by looking at the `dpy-`*`display_id`* file in `/dev/screen/0/`. The values you see from this file are the values that are currently used at the moment when you `cat` the file.

```
# cat /dev/screen/0/dpy-1
```

```
device id = 1
port id = 1
port type = HDMI
WFD_QNX_bchs extension = 0
WFD_QNX_vsync extension = 1
WFD_QNX_port_mode_info = 1
WFD_QNX_cbabc = 1
active = 1
device_id = LLG-4615-0-2010-1
video mode = 1280 x 720 @ 60 (progressive)
native resolution = 1280 x 1024
physical size = 360 x 290
detachable = 1
protection enabled = off
rotation support = WFD_ROTATION_SUPPORT_NONE
rotation mode = blits
rotation = 0, 0
fill port area = 0
writeback = 0
destination = WFD_INVALID_HANDLE
gamma range = [1 .. 1]
gamma = 1
aspect ratio = 0:0
refresh = 0
formats = rgba8888 rgbx8888 nv12
power mode = SCREEN_POWER_MODE_ON
mirror mode = disabled
viewport = (0,0 0x0)
cbabc mode = WFD_PORT_CBABC_MODE_NONE_QNX
updates = 0
counter = 1
priority = 15
rebuild = 1
splash = 0
background color = ff000000
transparent color = 00000000
capture buffer = (none)
frame buffer = win-0
rotation buffer = (none)
cursor = disabled
cursor updates = 0
window manager = (none)
composition module = pvr2d (loaded)
effect module = (null) (not loaded)
keep awakes = 0
focus = (none)
scene = (empty)
bypass = 0
dirty = (none)
windows = (none)
background = (0,0;1280,720)
holes = (none)
update mutex = 0x1615cc
update condvar = 0x1615d4
plane 0 {
```

```
  pipeline id = 1
  order = 2
  scale range = [0.125 .. 7]
  rotation support = WFD_ROTATION_SUPPORT_NONE
  usage = SCREEN_USAGE_DISPLAY
  format = SCREEN_FORMAT_RGBX8888
  cursor = off
  source = (none)
}
plane 1 {
  pipeline id = 4
  order = 3
  scale range = [1 .. 1]
  rotation support = WFD_ROTATION_SUPPORT_NONE
  usage = (none)
  format = SCREEN_FORMAT_RGBA8888
  cursor = off
  source = (none)
}
metrics.total.attach = 1
metrics.total.power = 0
metrics.total.idle = 0
metrics.total.events = 0
metrics.delta.attach = 0
metrics.delta.power = 0
metrics.delta.idle = 0
metrics.delta.events = 0
```

Sample `/dev/screen/0/win-0/win-0` file

This file lists the values of the framebuffer parameters that Screen is using.

The `win-0` file in `/dev/screen/0/` shows the configurations of the framebuffers that Screen is currently using.

From your target, you can see the configurations of the framebuffer that Screen is currently using by looking at the `win-0` file in `/dev/screen/0/win-0`. The values you see from this file are the values that are currently used at the moment when you `cat` the file.

For example, you can use this command from your shell:

```
# cat /dev/screen/0/win-0/win-0
```

to see the framebuffer parameters currently used:

```
type = SCREEN_ROOT_WINDOW
autonomous = 1
status = WIN_STATUS_REALIZED
replaced = 0
references = 1
id string =
display id = 1
plane id = 4
parent = (none)
children = (none)
window above = (none)
window below = (none)
alternate window = (none)
```

```
insert id = 0
reclip = 0
updates = 0
locked = 0
valid = 0x00000008
class = framebuffer
flags = WIN_FLAG_VISIBLE WIN_FLAG_FLOATING
buffer size = 1280x720
format = SCREEN_FORMAT_RGBA8888
usage = SCREEN_USAGE_DISPLAY
order = 0
sensitivity = SCREEN_SENSITIVITY_TEST
swap interval = 1
holes = (none)
regions = (none)
flip = 0
mirror = 0
source viewport = (0,0 1280x720)
source clip rectangle = (0,0;1280,720)
clipped source viewport = (0,0;1280,720 1280x720)
destination rectangle = (0,0 1280x720)
destination clip rectangle = (0,0;1280,720)
clipped destination rectangle = (0,0;1280,720 1280x720)
rotation = 0
clipped rotation = 0
transparency = SCREEN_TRANSPARENCY_SOURCE_OVER
global alpha = 255 -> 255
brightness = 0
contrast = 0
hue = 0
saturation = 0
scale quality = 0
idle mode = normal
page viewport = (none)
protection_enable = off
cbabc mode = SCREEN_CBABC_MODE_NONE
rcvids = (none)
metrics.total.updates.count = 1
metrics.total.updates.pixels = 921600 pixels
metrics.total.updates.reads = 0 bytes
metrics.total.updates.writes = 3 Mbytes
metrics.delta.updates.count = 1
metrics.delta.updates.pixels = 921600 pixels
metrics.delta.updates.reads = 0 bytes
metrics.delta.updates.writes = 3 Mbytes
```

Sample `/dev/screen/<pid>/win-1/win-1` file

This file lists the values of the window parameters that Screen is using for the process specified by the process ID.

The `win-`*`window_id`* file in `/dev/screen/`*`pid/`* shows the values of the window parameters that Screen is currently using.

From your target, you can see the values of the window parameters that Screen is currently using by looking at the `win-`*`window_id`* file in `/dev/screen/`*`pid`*`/win-`*`window_id`*. The values you see from this file are the values that are currently used at the moment when you `cat` the file.

For example, for process 1564694 with only one application window, you can use this command from your shell:

```
# cat /dev/screen/1564694/win-1/win-1
```

to see the window parameters currently used:

```
type = SCREEN_APPLICATION_WINDOW
autonomous = 0
status = WIN_STATUS_VISIBLE
replaced = 0
references = 1
id string = gles1-gears
display id = 1
plane id = 4
parent = (none)
children = (none)
window above = (none)
window below = (none)
alternate window = (none)
insert id = 1
reclip = 0
updates = 0
locked = 0
valid = 0x0000001c
class =
flags = WIN_FLAG_VISIBLE WIN_FLAG_FLOATING
buffer size = 1280x720
format = SCREEN_FORMAT_RGBA8888
usage = SCREEN_USAGE_OPENGL_ES1
order = 0
sensitivity = SCREEN_SENSITIVITY_TEST
swap interval = 1
holes = (none)
regions = (0,0;1280,720)
flip = 0
mirror = 0
source viewport = (0,0 1280x720)
source clip rectangle = (0,0;1280,720)
clipped source viewport = (0,0;1280,720 1280x720)
destination rectangle = (0,0 1280x720)
destination clip rectangle = (0,0;1280,720)
clipped destination rectangle = (0,0;1280,720 1280x720)
rotation = 0
clipped rotation = 0
transparency = SCREEN_TRANSPARENCY_NONE
global alpha = 255 -> 255
brightness = 0
contrast = 0
hue = 0
saturation = 0
scale quality = 0
idle mode = normal
page viewport = (none)
protection_enable = off
cbabc mode = SCREEN_CBABC_MODE_NONE
rcvids = 65563
metrics.total.objcnt = 0
metrics.total.apicnt = 0
metrics.total.drawcnt = 0
metrics.total.tricnt = 0
metrics.total.vtxcnt = 0
metrics.total.teximg = 0
metrics.total.subdata = 0
metrics.total.events = 0
metrics.total.blits.count = 0
metrics.total.blits.pixels = 0 pixels
```

```
metrics.total.blits.reads = 0 bytes
metrics.total.blits.writes = 0 bytes
metrics.total.posts.count = 2675
metrics.total.posts.pixels = 2465 Mpixels
metrics.total.updates.count = 2674
metrics.total.updates.pixels = 2464 Mpixels
metrics.total.updates.reads = 9857 Mbytes
metrics.delta.objcnt = 0
metrics.delta.apicnt = 0 (0 / post)
metrics.delta.drawcnt = 0 (0 / post)
metrics.delta.tricnt = 0 (0 / post)
metrics.delta.vtxcnt = 0 (0 / post)
metrics.delta.teximg = 0
metrics.delta.subdata = 0
metrics.delta.events = 0
metrics.delta.blits.count = 0
metrics.delta.blits.pixels = 0 pixels
metrics.delta.blits.reads = 0 bytes
metrics.delta.blits.writes = 0 bytes
metrics.delta.posts.count = 2675
metrics.delta.posts.pixels = 2465 Mpixels
metrics.delta.updates.count = 2674
metrics.delta.updates.pixels = 2464 Mpixels
metrics.delta.updates.reads = 9857 Mbytes
```

Chapter 11
Screen Library Reference

Two important aspects to note when using Screen are Function safety and Function execution types.

Function safety

Function safety refers to the situations under which it is safe to use the Screen API functions.

Function execution types

Function execution refers to the execution timeliness of the Screen API functions.

Function safety

Function safety refers to whether or not it's safe to use the Screen API functions in certain situations.

Screen API functions are thread-safe and will behave as documented even in a multithreaded environement. However, Screen API functions are neither interrupt-safe nor signal-safe. Don't use Screen API functions in an interrupt handler or a signal handler.

Classification:

Screen API

Safety	Value
Interrupt handler	No
Signal handler	No
Thread	Yes

Function execution types

Function execution timeliness essentially refers to whether or not the operations performed by Screen are immediate or delayed.

Each Screen API function can be categorized by its execution type:

Immediate execution

Immediate execution describes API functions where the command from the function call is executed immediately and isn't queued for batch processing.

Flushing execution

Flushing execution describes API functions where the command from the function call is queued for batch processing, and then the command buffer is immediately flushed to process all queued commands—previously queued ones included.

Delayed execution

Delayed execution describes API functions where the command from the function call is queued for batch processing.

Apply execution

Apply execution describes API functions where the command from the function call is added to the queue for batch processing, and then this queue is immediately flushed to process all queued commands—previously queued ones included. The display is redrawn if necessary.

Most of the Screen API functions aren't executed immediately. Instead, the commands resulting from the function calls are queued at the API for batch processing later in time. As your application makes multiple API function calls, commands accumulate in a command buffer that's associated with a context. These commands are batch-processed either when the command buffer is full, or when an API function of the type flushing execution is called. By batch-processing these commands, a large number of commands can be submitted in one atomic operation and the communication between the client and the Composition Manager is reduced to fewer, larger messages.

However, it's important to note that functions that are executed immediately may additionally flush a set of queued commands. Furthermore, a function that flushes any queued commands may or may not necessarily cause a redraw of the display.

You should understand the exact execution type of each API function. Knowing how and when these functions will be executed in the scope of your Screen application will be fundamental to your development.

Apply execution

Apply execution describes API functions where the command from the function call is added to the queue for batch processing, and then this queue is immediately flushed to process all queued commands—previously queued ones included. The display is redrawn if necessary.

Apply execution functions have essentially the same behavior as those of flushing execution, with the exception that apply execution functions cause the contents of the display to be updated when applicable.

For example, *screen_flush_context()* is of type apply execution. This API function adds the flush command to the batch-processing command queue. Then the function proceeds to flush this queue, and in doing so, communicates with the Composition Manager. The contents of the display will also be updated.

The return value from apply execution functions indicates whether or not all commands flushed from the batch-processing queue have been executed successfully. A successful return value indicates that all queued commands were processed and executed without errors. An unsuccessful return value indiates an error in either the execution of a previously queued command, or the flushing command (the command from the call of the apply execution function).

Delayed execution

Delayed execution describes API functions where the command from the function call is queued for batch processing.

Delayed execution functions queue the command so that it can be batch-processed later. The command remains on the command buffer until it's flushed and processed by a flushing execution function. A less frequent reason for flushing the command buffer is if it is full and can't accomodate the delayed execution function command. In this case, the command buffer is flushed and any previously queued commands are processed. Once the command buffer is empty, the command from the delayed execution function is queued to be batch-processed later.

For example, *screen_set_context_property_cv()* is of type delayed execution. This API function simply queues the command for setting the specified context property. The value of the property *isn't* actually set until a flushing exeuction function is called, or until a delayed execution function is called when the command buffer is full.

The return value from delayed execution functions indicates whether or not the command was successfully queued for batch-processing later.

Flushing execution

Flushing execution describes API functions where the command from the function call is queued for batch processing, and then the command buffer is immediately flushed to process all queued commands—previously queued ones included.

This behavior (adding commands to the command buffer and then immediately flushing it) typically happens because the execution of the API function depends on commands previously queued on the command buffer. In addition, the API function requires immediate communication with the Composition Manager.

For example, *screen_get_context_property_cv()* is of type flushing execution. This API function queues the command for retrieving the specified context property. Then the function proceeds to flush the command buffer, and after doing so, communicates with the Composition Manager. This function needs to flush the command buffer because the value of the property being retrieved may depend on a previously queued command, such as setting the value of the property. Therefore, the set command for the property should be processed first, before retrieving the value.

The return value from flushing execution functions indicates whether or not all commands flushed from the batch-processing queue have been executed successfully. A successful return value indicates that all queued commands were processed and executed without errors. An unsuccessful return value indiates an error in either the execution of a previously queued command, or the flushing command (the command from the call of the flushing execution function).

The Composition Manager doesn't stop processing batched commands when it detects an error. All commands queued for batch-processing will be processed until the batch-processing command queue is empty.

Immediate execution

Immediate execution describes API functions where the command from the function call is executed immediately and isn't queued for batch processing.

Some API functions, although categorized as immediate exeuction, may block for some period of time. The execution of the command is immediate, but the API function may need to communicate with the Composition Manager. This means that the client application will be blocked until this communication is complete and the required command to execute the API function is executed.

For example, both *screen_create_context()* and *screen_get_event()* are immediate execution types. Both API functions need to communicate with the Composition Manager. The *screen_create_context()* function normally returns in a timely manner when a connection to the services is established. Conversely, *screen_get_event()* may

block for long periods of time if the event queue is empty and a large or infinite timeout is specified.

Immediate execution functions neither cause the contents on a display to change nor flush any queued commands. Any previously queued commands for batch processing remain on the client side after you call immediate exeuction API functions —even if the immediate execution requires communication with the Composition Manager.

The return value from immediate execution functions indicates whether or not the execution of the API function was successful.

Function types

The tables in this section list the type of each function, where the type indicates the expected timeliness of each function call.

Blits

Function	Function type
screen_blit() (p. 258)	Delayed Execution
screen_fill() (p. 260)	Delayed Execution
screen_flush_blits() (p. 261)	Flushing Execution

Buffers

Function	Function type
screen_create_buffer() (p. 264)	Immediate Execution
screen_destroy_buffer() (p. 265)	Immediate Execution
screen_get_buffer_property_cv() (p. 265)	Immediate Execution
screen_get_buffer_property_iv() (p. 267)	Immediate Execution
screen_get_buffer_property_llv() (p. 268)	Immediate Execution
screen_get_buffer_property_pv() (p. 269)	Immediate Execution
screen_set_buffer_property_cv() (p. 270)	Immediate Execution
screen_set_buffer_property_iv() (p. 271)	Immediate Execution
screen_set_buffer_property_llv() (p. 272)	Immediate Execution
screen_set_buffer_property_pv() (p. 273)	Immediate Execution

Contexts

Function	Function type
screen_create_context() (p. 279)	Immediate Execution
screen_destroy_context() (p. 280)	Apply Execution
screen_flush_context() (p. 280)	Apply Execution
screen_get_context_property_cv() (p. 281)	Flushing Execution
screen_get_context_property_iv() (p. 282)	Flushing Execution
screen_get_context_property_llv() (p. 284)	Flushing Execution
screen_get_context_property_pv() (p. 285)	Flushing Execution
screen_notify() (p. 286)	Immediate Execution
screen_set_context_property_cv() (p. 287)	Delayed Execution
screen_set_context_property_iv() (p. 288)	Delayed Execution
screen_set_context_property_llv() (p. 289)	Delayed Execution
screen_set_context_property_pv() (p. 290)	Delayed Execution

Devices

Function	Function type
screen_create_device_type() (p. 300)	Immediate Execution
screen_destroy_device() (p. 301)	Flushing Execution
screen_get_device_property_cv() (p. 302)	Flushing Execution
screen_get_device_property_iv() (p. 303)	Flushing Execution
screen_get_device_property_llv() (p. 304)	Flushing Execution
screen_get_device_property_pv() (p. 305)	Flushing Execution
screen_set_device_property_cv() (p. 306)	Delayed Execution
screen_set_device_property_iv() (p. 308)	Delayed Execution
screen_set_device_property_llv() (p. 309)	Delayed Execution
screen_set_device_property_pv() (p. 310)	Delayed Execution

Displays

Function	Function type
screen_get_display_property_cv() (p. 319)	Flushing Execution
screen_get_display_property_iv() (p. 320)	Flushing Execution
screen_get_display_property_llv() (p. 322)	Flushing Execution
screen_get_display_property_pv() (p. 323)	Flushing Execution
screen_set_display_property_cv() (p. 325)	Delayed Execution
screen_set_display_property_iv() (p. 326)	Delayed Execution
screen_set_display_property_llv() (p. 327)	Delayed Execution
screen_set_display_property_pv() (p. 328)	Delayed Execution
screen_get_display_modes() (p. 318)	Flushing Execution
screen_read_display() (p. 324)	Immediate Execution
screen_share_display_buffers() (p. 329)	Flushing Execution
screen_wait_vsync() (p. 330)	Immediate Execution

Events

Function	Function type
screen_create_event() (p. 337)	Immediate Execution
screen_destroy_event() (p. 338)	Immediate Execution
screen_get_event() (p. 339)	Immediate Execution
screen_get_event_property_cv() (p. 340)	Immediate Execution
screen_get_event_property_iv() (p. 341)	Immediate Execution
screen_get_event_property_llv() (p. 344)	Immediate Execution
screen_get_event_property_pv() (p. 345)	Immediate Execution
screen_inject_event() (p. 347)	Immediate Execution
screen_send_event() (p. 348)	Immediate Execution
screen_set_event_property_cv() (p. 349)	Immediate Execution
screen_set_event_property_iv() (p. 350)	Immediate Execution
screen_set_event_property_llv() (p. 353)	Immediate Execution
screen_set_event_property_pv() (p. 354)	Immediate Execution

Groups

Function	Function type
screen_create_group() (p. 357)	Immediate Execution
screen_destroy_group() (p. 358)	Flushing Execution
screen_get_group_property_cv() (p. 359)	Flushing Execution
screen_get_group_property_iv() (p. 360)	Flushing Execution
screen_get_group_property_llv() (p. 361)	Flushing Execution
screen_get_group_property_pv() (p. 362)	Flushing Execution
screen_set_group_property_cv() (p. 364)	Delayed Execution
screen_set_group_property_iv() (p. 365)	Delayed Execution
screen_set_group_property_llv() (p. 366)	Delayed Execution
screen_set_group_property_pv() (p. 367)	Delayed Execution

Pixmaps

Function	Function type
screen_attach_pixmap_buffer() (p. 371)	Flushing Execution
screen_create_pixmap() (p. 372)	Immediate Execution
screen_create_pixmap_buffer() (p. 373)	Flushing Execution
screen_destroy_pixmap() (p. 373)	Flushing Execution
screen_destroy_pixmap_buffer() (p. 374)	Flushing Execution
screen_get_pixmap_property_cv() (p. 375)	Flushing Execution
screen_get_pixmap_property_iv() (p. 376)	Flushing Execution
screen_get_pixmap_property_llv() (p. 377)	Flushing Execution
screen_get_pixmap_property_pv() (p. 378)	Flushing Execution
screen_join_pixmap_group() (p. 379)	Delayed Execution
screen_leave_pixmap_group() (p. 380)	Delayed Execution
screen_ref_pixmap() (p. 381)	Immediate Execution
screen_set_pixmap_property_cv() (p. 382)	Delayed Execution
screen_set_pixmap_property_iv() (p. 383)	Delayed Execution
screen_set_pixmap_property_llv() (p. 384)	Delayed Execution

Function	Function type
screen_set_pixmap_property_pv() (p. 385)	Delayed Execution
screen_unref_pixmap() (p. 386)	Immediate Execution

Windows

Function	Function type
screen_attach_window_buffers() (p. 396)	Flushing Execution
screen_create_window() (p. 397)	Immediate Execution
screen_create_window_type() (p. 400)	Immediate Execution
screen_create_window_buffers() (p. 398)	Flushing Execution
screen_create_window_group() (p. 399)	Delayed Execution
screen_destroy_window() (p. 401)	Flushing Execution
screen_destroy_window_buffers() (p. 402)	Flushing Execution
screen_discard_window_regions() (p. 403)	Delayed Execution
screen_get_window_property_cv() (p. 404)	Flushing Execution
screen_get_window_property_iv() (p. 405)	Flushing Execution
screen_get_window_property_llv() (p. 407)	Flushing Execution
screen_get_window_property_pv() (p. 408)	Flushing Execution
screen_join_window_group() (p. 410)	Delayed Execution
screen_leave_window_group() (p. 411)	Delayed Execution
screen_post_window() (p. 411)	Apply Execution
screen_read_window() (p. 414)	Apply Execution
screen_ref_window() (p. 415)	Immediate Execution
screen_set_window_property_cv() (p. 416)	Delayed Execution
screen_set_window_property_iv() (p. 417)	Delayed Execution
screen_set_window_property_llv() (p. 419)	Delayed Execution
screen_set_window_property_pv() (p. 420)	Delayed Execution
screen_share_window_buffers() (p. 421)	Flushing Execution
screen_unref_window() (p. 422)	Immediate Execution
screen_wait_post() (p. 423)	Immediate Execution

General (screen.h)

Constants and Datatypes that are available for multiple Screen API objects.

Definitions in *screen.h*

Preprocessor macro definitions for the screen.h header file in the library.

Definitions:

`#define SCREEN_MODE_PREFERRED_INDEX (-1)`

Defines the mode preferred index.

Used as a convenience value to pass when setting SCREEN_PROPERTY_MODE to fall back to the default video mode without having to first query all the modes supported by the display to find the one with SCREEN_MODE_PREFERRED set in `flags`.

Library:

`libscreen`

_screen_mode

A structure to contain values related to Screen display mode.

Synopsis:

```
typedef struct _screen_mode {
    _Uint32t width ;
    _Uint32t height ;
    _Uint32t refresh ;
    _Uint32t interlaced ;
    _Uint32t aspect_ratio [2];
    _Uint32t flags ;
    _Uint32t index ;
    _Uint32t reserved [6];
}screen_display_mode_t;
```

Data:

_Uint32t width

Width of display.

_Uint32t height

Height of display.

_Uint32t refresh

Refresh of display.

_Uint32t interlaced

Interlace mode of display.

_Uint32t aspect_ratio[2]

Aspect ratio of display.

_Uint32t flags

Mutext flags of display.

_Uint32t index

Index of display.

_Uint32t reserved[6]

Reserved bits.

Library:

```
libscreen
```

Screen content mode types

Types of content modes.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_CBABC_MODE_NONE = 0x7671
      SCREEN_CBABC_MODE_VIDEO = 0x7672
      SCREEN_CBABC_MODE_UI = 0x7673
      SCREEN_CBABC_MODE_PHOTO = 0x7674
};
```

Data:

SCREEN_CBABC_MODE_NONE

The window content is not video, UI or photo.

SCREEN_CBABC_MODE_VIDEO

The window content is video.

SCREEN_CBABC_MODE_UI

The window content is UI.

SCREEN_CBABC_MODE_PHOTO

The window content is photo.

Library:

`libscreen`

Description:

The CBABC (content-based automatic brightness control) refers to the brightness control that is based on content, not ambient light. However, this enumeration is used mainly to describe the content type of the window, rather than the brightness control. If not set, the type will default to the mode of the display framebuffer.

Screen Alpha Blending Modes

Types of available alpha blending modes.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_NON_PRE_MULTIPLIED_ALPHA = 0
      SCREEN_PRE_MULTIPLIED_ALPHA = 1
};
```

Data:

SCREEN_NON_PRE_MULTIPLIED_ALPHA

The non pre-multiplied alpha content.

This is the default. In this case, the source blending is done using the equation:

c(r,g,b) = s(r,g,b) * s(a) + d(r,g,b) * (1 - s(a))

SCREEN_PRE_MULTIPLIED_ALPHA

The pre-multiplied alpha content.

In this case, the source blending is done using the equation:

c(r,g,b) = s(r,g,b) + d(r,g,b) * (1 - s(a))

Library:

libscreen

Description:

Screen color space types

The window/pixmap supported color space types.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_COLOR_SPACE_UNCORRECTED = 0x0
      SCREEN_COLOR_SPACE_SRGB = 0x1
      SCREEN_COLOR_SPACE_LRGB = 0x2
      SCREEN_COLOR_SPACE_BT601 = 0x3
      SCREEN_COLOR_SPACE_BT601_FULL = 0x4
      SCREEN_COLOR_SPACE_BT709 = 0x5
      SCREEN_COLOR_SPACE_BT709_FULL = 0x6
};
```

Data:

SCREEN_COLOR_SPACE_UNCORRECTED

The default.

SCREEN_COLOR_SPACE_SRGB

SCREEN_COLOR_SPACE_LRGB

Linear RGB.

SCREEN_COLOR_SPACE_BT601

SCREEN_COLOR_SPACE_BT601_FULL

SCREEN_COLOR_SPACE_BT709

SCREEN_COLOR_SPACE_BT709_FULL

Library:

libscreen

Description:

Screen flushing types

Types of flushing options.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_WAIT_IDLE = (1 << 0)
      SCREEN_PROTECTED = (1 << 1)
};
```

Data:

SCREEN_WAIT_IDLE

Indicates that the function will block until the operation has completed.

SCREEN_PROTECTED

Library:

```
libscreen
```

Description:

Screen idle mode types

Types of idle modes.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_IDLE_MODE_NORMAL = 0
      SCREEN_IDLE_MODE_KEEP_AWAKE = 1
};
```

Data:

SCREEN_IDLE_MODE_NORMAL

The default idle mode; the display is allowed to go idle after the period of time indicated by SCREEN_PROPERTY_IDLE_TIMEOUT and potentially turn off.

SCREEN_IDLE_MODE_KEEP_AWAKE

The idle mode which will prevent the display from going idle after a period of no input - such as video playback.

By default, the display will go idle after the period of time indicated by `SCREEN_PROPERTY_IDLE_TIMEOUT`.

Library:

`libscreen`

Description:

Screen mirror types

Types of mirrors.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_MIRROR_DISABLED = 0
      SCREEN_MIRROR_NORMAL = 1
      SCREEN_MIRROR_STRETCH = 2
      SCREEN_MIRROR_ZOOM = 3
      SCREEN_MIRROR_FILL = 4
};
```

Data:

SCREEN_MIRROR_DISABLED

Mirroring is disabled.

SCREEN_MIRROR_NORMAL

Mirroring is enabled and that the aspect-ratio of the image is 1:1.

SCREEN_MIRROR_STRETCH

Mirroring is enabled and that the image should fill the screen while not preserving the aspect-ratio.

SCREEN_MIRROR_ZOOM

Mirroring is enabled and that the image should fill the screen while preserving the aspect-ratio.

Image content may be clipped.

SCREEN_MIRROR_FILL

Mirroring is enabled and that the image should fill the screen while preserving the aspect-ratio.

Image may be shown with black bars where applicable.

Library:

libscreen

Description:

Screen mouse button types

Types of mouse buttons.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_LEFT_MOUSE_BUTTON = (1 << 0)
      SCREEN_MIDDLE_MOUSE_BUTTON = (1 << 1)
      SCREEN_RIGHT_MOUSE_BUTTON = (1 << 2)
};
```

Data:

SCREEN_LEFT_MOUSE_BUTTON

SCREEN_MIDDLE_MOUSE_BUTTON

SCREEN_RIGHT_MOUSE_BUTTON

Library:

libscreen

Description:

Screen object types

Types of Screen API objects.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_OBJECT_TYPE_CONTEXT = 0
      SCREEN_OBJECT_TYPE_GROUP = 1
      SCREEN_OBJECT_TYPE_DISPLAY = 2
      SCREEN_OBJECT_TYPE_DEVICE = 3
      SCREEN_OBJECT_TYPE_PIXMAP = 4
      SCREEN_OBJECT_TYPE_WINDOW = 8
};
```

Data:

SCREEN_OBJECT_TYPE_CONTEXT

SCREEN_OBJECT_TYPE_GROUP

SCREEN_OBJECT_TYPE_DISPLAY

SCREEN_OBJECT_TYPE_DEVICE

SCREEN_OBJECT_TYPE_PIXMAP

SCREEN_OBJECT_TYPE_WINDOW

Library:

```
libscreen
```

Description:

Screen pixel format types

Types of supported pixel formats.

Synopsis:

```
#include <screen/screen.h>
```

```
enum {
      SCREEN_FORMAT_BYTE = 1
      SCREEN_FORMAT_RGBA4444 = 2
      SCREEN_FORMAT_RGBX4444 = 3
      SCREEN_FORMAT_RGBA5551 = 4
      SCREEN_FORMAT_RGBX5551 = 5
      SCREEN_FORMAT_RGB565 = 6
      SCREEN_FORMAT_RGB888 = 7
      SCREEN_FORMAT_RGBA8888 = 8
      SCREEN_FORMAT_RGBX8888 = 9
      SCREEN_FORMAT_YVU9 = 10
      SCREEN_FORMAT_YUV420 = 11
      SCREEN_FORMAT_NV12 = 12
      SCREEN_FORMAT_YV12 = 13
      SCREEN_FORMAT_UYVY = 14
      SCREEN_FORMAT_YUY2 = 15
      SCREEN_FORMAT_YVYU = 16
      SCREEN_FORMAT_V422 = 17
      SCREEN_FORMAT_AYUV = 18
      SCREEN_FORMAT_NFORMATS
};
```

Data:

SCREEN_FORMAT_BYTE

SCREEN_FORMAT_RGBA4444

16 bits per pixel (4 bits per channel) RGB with alpha channel

SCREEN_FORMAT_RGBX4444

16 bits per pixel (4 bits per channel) RGB with alpha channel disregarded

SCREEN_FORMAT_RGBA5551

16 bits per pixel, 2 bytes containing R, G, and B values (5 bits per channel with single-bit alpha channel)

SCREEN_FORMAT_RGBX5551

16 bits per pixel, 2 bytes containing R, G, and B values (5 bits per channel with single-bit alpha channel disregarded)

SCREEN_FORMAT_RGB565

16 bits per pixel; uses five bits for red, six bits for green and five bits for blue.

This pixel format represents each pixel in the following order (high byte to low byte): RRRR RGGG GGGB BBBB

SCREEN_FORMAT_RGB888

24 bits per pixel (8 bits per channel) RGB

SCREEN_FORMAT_RGBA8888

32 bits per pixel (8 bits per channel) RGB with alpha channel

SCREEN_FORMAT_RGBX8888

32 bits per pixel (8 bits per channel) RGB with alpha channel disregarded

SCREEN_FORMAT_YVU9

9 bits per pixel planar YUV format.

8-bit Y plane and 8-bit 4x4 subsampled U and V planes. Registered by Intel.

SCREEN_FORMAT_YUV420

Standard NTSC TV transmission format.

SCREEN_FORMAT_NV12

12 bits per pixel planar YUV format.

8-bit Y plane and 2x2 subsampled, interleaved U and V planes.

SCREEN_FORMAT_YV12

12 bits per pixel planar YUV format.

8-bit Y plane and 8-bit 2x2 subsampled U and V planes.

SCREEN_FORMAT_UYVY

16 bits per pixel packed YUV format.

YUV 4:2:2 - Y sampled at every pixel, U and V sampled at every second pixel horizontally on each line. A macropixel contains 2 pixels in 1 uint32.

SCREEN_FORMAT_YUY2

16 bits per pixel packed YUV format.

YUV 4:2:2 - as in UYVY, but with different component ordering within the uint32 macropixel.

SCREEN_FORMAT_YVYU

16 bits per pixel packed YUV format.

YUV 4:2:2 - as in UYVY, but with different component ordering within the uint32 macropixel.

SCREEN_FORMAT_V422

Packed YUV format.

Inverted version of UYVY.

SCREEN_FORMAT_AYUV

Packed YUV format.

Combined YUV and alpha

SCREEN_FORMAT_NFORMATS

Library:

`libscreen`

Description:

Formats with an alpha channel will have source alpha enabled automatically. Applications that want the Screen library to disregard the alpha channel can choose a pixel format with an X.

Screen power mode types

Types of power modes.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_POWER_MODE_OFF = 0x7680
      SCREEN_POWER_MODE_SUSPEND = 0x7681
      SCREEN_POWER_MODE_LIMITED_USE = 0x7682
      SCREEN_POWER_MODE_ON = 0x7683
};
```

Data:

SCREEN_POWER_MODE_OFF

The power mode in an inactive state.

SCREEN_POWER_MODE_SUSPEND

The power mode in a state of being partially off; the display or device is no longer active.

The power usage in this state can be greater than in SCREEN_POWER_MODE_OFF, but will allow for a faster transition to active state.

SCREEN_POWER_MODE_LIMITED_USE

The power mode in a state of reduced power; the display or device is active, but may be slower to update than if it was in SCREEN_POWER_MODE_ON.

SCREEN_POWER_MODE_ON

The power mode in an active state.

Library:

`libscreen`

Description:

Screen property types

Types of properties that are associated with Screen API objects.

Synopsis:

```
#include <screen/screen.h>

 enum {
      SCREEN_PROPERTY_ALPHA_MODE = 1
      SCREEN_PROPERTY_GAMMA = 2
      SCREEN_PROPERTY_BRIGHTNESS = 3
      SCREEN_PROPERTY_BUFFER_COUNT = 4
      SCREEN_PROPERTY_BUFFER_SIZE = 5
      SCREEN_PROPERTY_BUTTONS = 6
      SCREEN_PROPERTY_CLASS = 7
      SCREEN_PROPERTY_COLOR_SPACE = 8
      SCREEN_PROPERTY_CONTRAST = 9
      SCREEN_PROPERTY_DEVICE = 10
      SCREEN_PROPERTY_DEVICE_INDEX = 10
      SCREEN_PROPERTY_DISPLAY = 11
      SCREEN_PROPERTY_EGL_HANDLE = 12
      SCREEN_PROPERTY_FLIP = 13
      SCREEN_PROPERTY_FORMAT = 14
      SCREEN_PROPERTY_FRONT_BUFFER = 15
```

```
SCREEN_PROPERTY_GLOBAL_ALPHA = 16
SCREEN_PROPERTY_PIPELINE = 17
SCREEN_PROPERTY_GROUP = 18
SCREEN_PROPERTY_HUE = 19
SCREEN_PROPERTY_ID_STRING = 20
SCREEN_PROPERTY_INPUT_VALUE = 21
SCREEN_PROPERTY_INTERLACED = 22
SCREEN_PROPERTY_JOG_COUNT = 23
SCREEN_PROPERTY_KEY_CAP = 24
SCREEN_PROPERTY_KEY_FLAGS = 25
SCREEN_PROPERTY_KEY_MODIFIERS = 26
SCREEN_PROPERTY_KEY_SCAN = 27
SCREEN_PROPERTY_KEY_SYM = 28
SCREEN_PROPERTY_MIRROR = 29
SCREEN_PROPERTY_NAME = 30
SCREEN_PROPERTY_OWNER_PID = 31
SCREEN_PROPERTY_PHYSICALLY_CONTIGUOUS = 32
SCREEN_PROPERTY_PLANAR_OFFSETS = 33
SCREEN_PROPERTY_POINTER = 34
SCREEN_PROPERTY_POSITION = 35
SCREEN_PROPERTY_PROTECTED = 36
SCREEN_PROPERTY_RENDER_BUFFERS = 37
SCREEN_PROPERTY_ROTATION = 38
SCREEN_PROPERTY_SATURATION = 39
SCREEN_PROPERTY_SIZE = 40
SCREEN_PROPERTY_SOURCE_POSITION = 41
SCREEN_PROPERTY_SOURCE_SIZE = 42
SCREEN_PROPERTY_STATIC = 43
SCREEN_PROPERTY_STRIDE = 44
SCREEN_PROPERTY_SWAP_INTERVAL = 45
SCREEN_PROPERTY_TRANSPARENCY = 46
SCREEN_PROPERTY_TYPE = 47
SCREEN_PROPERTY_USAGE = 48
SCREEN_PROPERTY_USER_DATA = 49
SCREEN_PROPERTY_USER_HANDLE = 50
SCREEN_PROPERTY_VISIBLE = 51
SCREEN_PROPERTY_WINDOW = 52
SCREEN_PROPERTY_RENDER_BUFFER_COUNT = 53
SCREEN_PROPERTY_ZORDER = 54
SCREEN_PROPERTY_PHYSICAL_ADDRESS = 55
SCREEN_PROPERTY_SCALE_QUALITY = 56
SCREEN_PROPERTY_SENSITIVITY = 57
SCREEN_PROPERTY_MIRROR_MODE = 58
SCREEN_PROPERTY_DISPLAY_COUNT = 59
SCREEN_PROPERTY_DISPLAYS = 60
SCREEN_PROPERTY_CBABC_MODE = 61
SCREEN_PROPERTY_EFFECT = 62
SCREEN_PROPERTY_FLOATING = 63
SCREEN_PROPERTY_ATTACHED = 64
SCREEN_PROPERTY_DETACHABLE = 65
SCREEN_PROPERTY_NATIVE_RESOLUTION = 66
SCREEN_PROPERTY_PROTECTION_ENABLE = 67
SCREEN_PROPERTY_SOURCE_CLIP_POSITION = 68
SCREEN_PROPERTY_PHYSICAL_SIZE = 69
SCREEN_PROPERTY_FORMAT_COUNT = 70
SCREEN_PROPERTY_FORMATS = 71
SCREEN_PROPERTY_SOURCE_CLIP_SIZE = 72
SCREEN_PROPERTY_TOUCH_ID = 73
SCREEN_PROPERTY_VIEWPORT_POSITION = 74
SCREEN_PROPERTY_VIEWPORT_SIZE = 75
SCREEN_PROPERTY_TOUCH_ORIENTATION = 76
SCREEN_PROPERTY_TOUCH_PRESSURE = 77
SCREEN_PROPERTY_TIMESTAMP = 78
SCREEN_PROPERTY_SEQUENCE_ID = 79
```

```
        SCREEN_PROPERTY_IDLE_MODE = 80
        SCREEN_PROPERTY_IDLE_STATE = 81
        SCREEN_PROPERTY_KEEP_AWAKES = 82
        SCREEN_PROPERTY_IDLE_TIMEOUT = 83
        SCREEN_PROPERTY_KEYBOARD_FOCUS = 84
        SCREEN_PROPERTY_MTOUCH_FOCUS = 85
        SCREEN_PROPERTY_POINTER_FOCUS = 86
        SCREEN_PROPERTY_ID = 87
        SCREEN_PROPERTY_POWER_MODE = 88
        SCREEN_PROPERTY_MODE_COUNT = 89
        SCREEN_PROPERTY_MODE = 90
        SCREEN_PROPERTY_CLIP_POSITION = 91
        SCREEN_PROPERTY_CLIP_SIZE = 92
        SCREEN_PROPERTY_COLOR = 93
        SCREEN_PROPERTY_MOUSE_WHEEL = 94
        SCREEN_PROPERTY_CONTEXT = 95
        SCREEN_PROPERTY_DEBUG = 96
        SCREEN_PROPERTY_ALTERNATE_WINDOW = 97
        SCREEN_PROPERTY_DEVICE_COUNT = 98
        SCREEN_PROPERTY_BUFFER_POOL = 99
        SCREEN_PROPERTY_OBJECT_TYPE = 100
        SCREEN_PROPERTY_DEVICES = 101
        SCREEN_PROPERTY_KEYMAP_PAGE = 102
        SCREEN_PROPERTY_SELF_LAYOUT = 103
        SCREEN_PROPERTY_GROUP_COUNT = 104
        SCREEN_PROPERTY_GROUPS = 105
        SCREEN_PROPERTY_PIXMAP_COUNT = 106
        SCREEN_PROPERTY_PIXMAPS = 107
        SCREEN_PROPERTY_WINDOW_COUNT = 108
        SCREEN_PROPERTY_WINDOWS = 109
        SCREEN_PROPERTY_KEYMAP = 110
        SCREEN_PROPERTY_MOUSE_HORIZONTAL_WHEEL = 111
        SCREEN_PROPERTY_TOUCH_TYPE = 112
        SCREEN_PROPERTY_NATIVE_IMAGE = 113
        SCREEN_PROPERTY_SCALE_FACTOR = 114
        SCREEN_PROPERTY_DPI = 115
        SCREEN_PROPERTY_METRIC_COUNT = 116
        SCREEN_PROPERTY_METRICS = 117
        SCREEN_PROPERTY_BUTTON_COUNT = 118
        SCREEN_PROPERTY_VENDOR = 119
        SCREEN_PROPERTY_PRODUCT = 120
        SCREEN_PROPERTY_BRUSH_CLIP_POSITION = 121
        SCREEN_PROPERTY_BRUSH_CLIP_SIZE = 122
        SCREEN_PROPERTY_ANALOG0 = 123
        SCREEN_PROPERTY_ANALOG1 = 124
        SCREEN_PROPERTY_BRUSH = 125
        SCREEN_PROPERTY_TRANSFORM = 127
        SCREEN_PROPERTY_TECHNOLOGY = 129
        SCREEN_PROPERTY_REFERENCE_COLOR = 138
};
```

Data:

SCREEN_PROPERTY_ALPHA_MODE

A single integer that defines how alpha should be interpreted.

The alpha mode must be of type *Screen alpha mode types* (p. 191). When retrieving or setting this property type, ensure that you have sufficient storage

for one integer. The following API objects have this property and share this same definition:

- pixmap
- window
    - In the configuration file, , the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following are valid settings that can be used for this property in graphics.conf:
        - alpha-mode = pre-multipled
        - alpha-mode = non-pre-multipled

SCREEN_PROPERTY_GAMMA

A single integer that indicates the gamma value of the current display; a property of a display object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer.

SCREEN_PROPERTY_BRIGHTNESS

A single integer between [-255..255] that is used to adjust the brightness of a window; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- brightness = [brightness -255..255]

SCREEN_PROPERTY_BUFFER_COUNT

A single integer that indicates the number of buffers that were created or attached to the window; a property of a window object.

When retrieving this property type, ensure that you have sufficient storage for one integer. Also note that this property is local to the window object. This means that a query for this property will not trigger a flush of the command buffer despite that screen_get_window_property_iv() is of type

flushing execution. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value (beyond initialization, this property can only queried and not set. The following is the usage for setting this property in graphics.conf:

- buffer-count = [number of buffers]

SCREEN_PROPERTY_BUFFER_SIZE

A pair of integers containing the width and height, in pixels, of the buffer.

When retrieving or setting this property type, ensure that you provide sufficient storage for two integers. The following API objects have this property and share this same definition:

- buffer
- pixmap
- window
    - In the configuration file, , the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:
        - surface-size [width] x [height]

SCREEN_PROPERTY_BUTTONS

A single integer which is a bitmask indicating which buttons are pressed; a property of an event object or device object.

Note that D-pad, A, B, X, Y, Start, Select, Left, and Right are all considered buttons on gamepads. Currently, there is no button-map on gamepads equivalent to keymaps for keyboards. When retrieving this property type, ensure that you provide sufficient storage for one integer. The SCREEN_PROPERTY_BUTTONS property is applicable to the following events and device types:

- SCREEN_EVENT_GAMEPAD
- SCREEN_EVENT_JOYSTICK
- SCREEN_EVENT_MTOUCH_TOUCH
- SCREEN_EVENT_MTOUCH_MOVE
- SCREEN_EVENT_MTOUCH_RELEASE

- SCREEN_EVENT_POINTER: In the case of a pointer, SCREEN_PROPERTY_BUTTONS must be a combination of type *Screen mouse button types* (p. 195).

SCREEN_PROPERTY_CLASS

The name of a class as defined in the configuration file, graphics.conf; a property of a window object.

The class specifies a set of window property and value pairs which will be applied to the window as initial or or default values. When retrieving or setting this property type, ensure that you have sufficient storage for a character buffer.

SCREEN_PROPERTY_COLOR_SPACE

A single integer that indicates the color space of a buffer.

The color space must be of type *Screen color space types* (p. 192). The default value is SCREEN_COLOR_SPACE_UNCORRECTED. When retrieving or setting this property type, ensure that you have sufficient storage for one integer. The following API objects have this property and share this same definition:

- pixmap
- window

SCREEN_PROPERTY_CONTRAST

A single integer between [-128..127] that is used to adjust the contrast of a window; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- contrast = [window contrast -128..127]

SCREEN_PROPERTY_DEVICE

A single integer representing the object handle for the input device that the event came from; a property of an event object.

When retrieving this property type, ensure that you provide sufficient storage for one integer. The SCREEN_PROPERTY_DEVICE property is applicable to the following events:

- SCREEN_EVENT_GAMEPAD
- SCREEN_EVENT_INPUT
- SCREEN_EVENT_JOG
- SCREEN_EVENT_JOYSTICK
- SCREEN_EVENT_KEYBOARD
- SCREEN_EVENT_MTOUCH_TOUCH
- SCREEN_EVENT_MTOUCH_MOVE
- SCREEN_EVENT_MTOUCH_RELEASE
- SCREEN_EVENT_POINTER
- SCREEN_EVENT_DEVICE

SCREEN_PROPERTY_DEVICE_INDEX

Deprecated:

This property has been deprecated.

Use SCREEN_PROPERTY_DEVICE instead.

SCREEN_PROPERTY_DISPLAY

A display handle.

When retrieving or setting this property type, ensure that you have sufficient storage for one `void` pointer. The following API objects have this property, each with its own variant of the definition:

- device: The display that is the focus for the specified input device. A value of NULL indicates that the input device is focused on the default display.
- event:
  - In the case of the SCREEN_EVENT_PROPERTY event, SCREEN_PROPERTY_DISPLAY is the property of either a device or a window, depending on the recipient object of the event.
  - In the case of the SCREEN_EVENT_DISPLAY event, SCREEN_PROPERTY_DISPLAY is the handle of the new external display that has been detected.

- In the case of the SCREEN_EVENT_IDLE event, SCREEN_PROPERTY_DISPLAY is the handle of the display in which a window entered an idle state.

- window: The display that the specified window will be shown on if the window is visible. A value of NULL indicates that the window will be shown on the default display. Note that setting SCREEN_PROPERTY_DISPLAY invalidates the pipeline. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have its property initialized to this value. The following are valid settings that can be used for this property in graphics.conf:
  - display = internal
  - display = composite
  - display = svideo
  - display = YPbPr
  - display = rgb
  - display = rgbhv
  - display = dvi
  - display = hdmi
  - display = [display id]

SCREEN_PROPERTY_EGL_HANDLE

A handle to the EGL driver; a property of a buffer object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one `void` pointer.

SCREEN_PROPERTY_FLIP

A single integer that indicates whether or not the window contents are flipped; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_FORMAT

A single integer that indicates the pixel format of the buffer.

The format must be of type *Screen pixel format types* (p. 196). When retrieving or setting this property type, ensure that you provide sufficient storage for

one integer. The following API objects have this property and share this same definition:

- buffer
- pixmap
- window
    - When you set a format with alpha (e.g., SCREEN_FORMAT_RGBA4444), the SCREEN_PROPERTY_TRANSPARENCY property is set to SCREEN_TRANSPARENCY_SOURCE_OVER as a convenience. If this is not your intention, then we recommend that you use a pixel format type that disregards the alpha channel (e.g. SCREEN_FORMAT_RGBX4444).
    - In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have its property initialized to this value. The following are valid settings that can be used for this property in graphics.conf:
        - format = byte
        - format = rgba4444
        - format = rgbx4444
        - format = rgba5551
        - format = rgbx5551
        - format = rgb565
        - format = rgb888
        - format = rgba8888
        - format = rgbx8888
        - format = yvu9
        - format = nv12
        - format = yv12
        - format = uyvy
        - format = yuy2
        - format = yvyu
        - format = v422
        - format = ayuv
        - format = [pixel format type 0..16]

SCREEN_PROPERTY_FRONT_BUFFER

A handle to the last buffer of the window to have been posted; a property of a window object.

When retrieving this property type, ensure that you have sufficient storage for one `void` pointer.

SCREEN_PROPERTY_GLOBAL_ALPHA

A single integer that indicates the global alpha value to be applied to the window; a property of a window object.

This value must be between `0` and `255`. When retrieving or setting this property type, ensure that you have sufficient storage for one integer. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- global-alpha = [global alpha 0..255]

SCREEN_PROPERTY_PIPELINE

A single integer that contains the pipeline ID; a property of a window object.

Screen uses, as much as possible, hardware layering (pipelines) for composition. You must determine the pipelines that are on your system and then you choose the pipeline on which you want to display a window.

Pipeline ordering and the z-ordering of windows on a layer are not related to each other. If your application assigns pipelines manually, it must ensure that the z-order values makes sense with regard to the pipeline order of the target hardware. Pipeline ordering takes precedence over z-ordering operations in Screen. Screen does not control the ordering of hardware pipelines. It always arranges windows in the z-order specified by the application.

If you assign a framebuffer to the top layer in a graphics configuration on a non-composited window (which does not have the correct z-order set), your application cannot display a new window (no matter its z-order) above the framebuffer. The same constraint applies if you assign a framebuffer to the bottom layer of a graphics configuration. In this case, your application cannot display a window below the framebuffer.

Using the `SCREEN_USAGE_OVERLAY` flag in `SCREEN_PROPERTY_USAGE` is recommended. Otherwise, Screen may ignore `SCREEN_PROPERTY_PIPELINE`.

In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- pipeline = [pipeline id]

SCREEN_PROPERTY_GROUP

The group that the API object is associated with.

When retrieving or setting this property type, ensure that you have sufficient storage according to the definition of the property for the specific API object. The following API objects have this property, each with its own variant of this definition(s):

- event: The window group that is associated with the event. SCREEN_PROPERTY_GROUP is applicable for the following events:
  - SCREEN_EVENT_IDLE The pointer to a group of type screen_group_t that has changed to idle state.
  - SCREEN_EVENT_PROPERTY The pointer to a group of type screen_group_t that has had a property changed.
  - SCREEN_EVENT_CREATE The name of the group that the window has joined. Typically this property would be relevant only to a SCREEN_EVENT_CREATE event for a child or embedded window. For an application window, the SCREEN_EVENT_CREATE is forwarded to the windowing system immediately and will have no group associated with it. However, a SCREEN_EVENT_CREATE event for a child or embedded window does not really exist unless it is associated with a parent (window group). Therefore, the SCREEN_EVENT_CREATE event for a child or embedded window has a group association.
- pixmap:
  - The name of the group that the pixmap is associated with when SCREEN_PROPERTY_GROUP is used with screen_get_pixmap_property_cv(). When retrieving this property type, ensure that you have sufficient storage for a character buffer.
  - The pointer to a group of type screen_group_t, that the pixmap is associated with when SCREEN_PROPERTY_GROUP is used with screen_get_pixmap_property_pv(). When retrieving this property type, ensure that you have sufficient storage for a structure of type screen_group_t.
- window
  - The name of the group that the window has created or parented when SCREEN_PROPERTY_GROUP is used with screen_get_window_property_cv(). When retrieving this property type,

ensure that you have sufficient storage for a character buffer. If the window has not created or parented a group, then this property refers to the group that the window has joined.

- The pointer to a group of type screen_group_t, that the window has created or parented when SCREEN_PROPERTY_GROUP is used with screen_get_window_property_pv(). When retrieving this property type, ensure that you have sufficient storage for a for a structure of type screen_group_t. If the window has not created or parented a group, then this property refers to the group that the window has joined.

SCREEN_PROPERTY_HUE

A single integer between [-128..127] that is used to adjust the hue of a window; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- hue = [global alpha -128..127]

SCREEN_PROPERTY_ID_STRING

A string that can be used by window manager or parent to identify the contents of the specified API object.

When retrieving or setting this property type, ensure that you provide a character buffer. The following API objects have this property and share this same definition:

- device
- display (SCREEN_PROPERTY_ID_STRING can only be retrieved and not set for a display object)
- pixmap
- window
    - In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:
        - id_string = [string]

SCREEN_PROPERTY_INPUT_VALUE

A single integer that indicates the input value associated with the specific event; a property of an event object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer. SCREEN_PROPERTY_INPUT_VALUE is applicable only to a SCREEN_EVENT_INPUT event.

SCREEN_PROPERTY_INTERLACED

A single integer that indicates whether or not the buffer contains interlaced fields instead of progressive data; a property of a buffer object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer.

SCREEN_PROPERTY_JOG_COUNT

A single integer that indicates the jog count associated with the specific event; a property of an event object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer. SCREEN_PROPERTY_JOG_COUNT is applicable only to a SCREEN_EVENT_JOG event.

SCREEN_PROPERTY_KEY_CAP

A single integer that indicates the keyboard cap associated with the specific event; a property of an event object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer. SCREEN_PROPERTY_KEY_CAP is applicable only to a SCREEN_EVENT_KEYBOARD event.

SCREEN_PROPERTY_KEY_FLAGS

A single integer that indicates the keyboard flags associated with the specific event; a property of an event object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer. SCREEN_PROPERTY_KEY_FLAGS is applicable only only to a SCREEN_EVENT_KEYBOARD event.

SCREEN_PROPERTY_KEY_MODIFIERS

A single integer that indicates the keyboard modifiers associated with the specific API object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer. The following API objects have this property and share this same definition:

- display(SCREEN_PROPERTY_KEY_MODIFIERS can only be retrieved and not set for a display object)
- device(SCREEN_PROPERTY_KEY_MODIFIERS can only be retrieved and not set for a device object)
- event: This is only applicable for the following events:
  - SCREEN_EVENT_MTOUCH_TOUCH
  - SCREEN_EVENT_MTOUCH_MOVE
  - SCREEN_EVENT_MTOUCH_RELEASE
  - SCREEN_EVENT_POINTER
  - SCREEN_EVENT_KEYBOARD
  - SCREEN_EVENT_GAMEPAD
  - SCREEN_EVENT_JOYSTICK

SCREEN_PROPERTY_KEY_SCAN

A single integer that indicates the keyboard scan associated with the specific event; a property of an event object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer. SCREEN_PROPERTY_KEY_SCAN is applicable only to a SCREEN_EVENT_KEYBOARD event.

SCREEN_PROPERTY_KEY_SYM

A single integer that indicates the keyboard symbols associated with the specific event; a property of an event object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer. SCREEN_PROPERTY_KEY_SYM is applicable only to a SCREEN_EVENT_KEYBOARD event.

SCREEN_PROPERTY_MIRROR

A single integer (0 or 1) that indicates whether or not contents of the API object are mirrored (flipped horizontally); a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_NAME

A single integer containing the name of the window group.

When retrieving or setting this property type, ensure you provide sufficient storage a character buffer. The following API objects have this property, and share this same definition:

- event (Applicable only to a SCREEN_EVENT_PROPERTY event)
- group

SCREEN_PROPERTY_OWNER_PID

A single integer that indicates the process id of the process responsible for creating the window; a property of a window object.

This property can be used by window managers to identify windows. When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_PHYSICALLY_CONTIGUOUS

A single integer that indicates whether or not the buffer is physically contiguous; a property of a buffer object.

When retrieving or setting this property type, ensure you provide sufficient storage for one integer.

SCREEN_PROPERTY_PLANAR_OFFSETS

Three integers that provide the offset from the base address for each of the Y, U and V components of planar YUV formats; a property of a buffer object.

When retrieving or setting this property type, ensure that you have sufficient storage for three integers.

SCREEN_PROPERTY_POINTER

A pointer that can be used by software renderers to read from and/or write to the buffer; a property of a buffer object.

When this property is used, ensure that you provide sufficient storage space for one `void` pointer. The buffer must have been realized with a usage containing `SCREEN_USAGE_READ` and/or `SCREEN_USAGE_WRITE` for this property to be a valid pointer.

SCREEN_PROPERTY_POSITION

Integers that define position of the screen coordinates of the related API object.

The following API objects have this property, each with its own variant of this definition:

- event: The x and y values for the contact point of the mtouch or pointer. When retrieving or setting this property type, ensure that you have sufficient storage for two integers. This is only applicable for the following events:
  - `SCREEN_EVENT_MTOUCH_TOUCH`
  - `SCREEN_EVENT_MTOUCH_MOVE`
  - `SCREEN_EVENT_MTOUCH_RELEASE`
  - `SCREEN_EVENT_POINTER`
- window: The x and y positions of the window screen coordinates. Remember that the position of child and embedded windows are relative to the parent window. For example, if the position of the application window is {10, 10} and the position of the child window is {10, 10}, then the position of the child window on the screen is actually {20, 20}. When retrieving or setting this property type, ensure that you have sufficient storage for two integers. In the configuration file, , the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:
  - window-position = [x-position] , [y-position]

SCREEN_PROPERTY_PROTECTED

A single integer that specifies whether or not there is protection for the buffer; a property of a buffer object.

The content of the buffer will not be displayed unless there is a secure link present. Operations on the buffer such as reading from, writing to, or mapping a region of the buffer to a different address space will be prohibited. Note that setting protection on a buffer does not invoke a request for authentication. Typically, the window that owns the buffer will have its window property, SCREEN_PROPERTY_PROTECTION_ENABLE, set. The request for authentication will be made when the window is posted and its SCREEN_PROPERTY_VISIBLE property indicates that the window is visible. When retrieving or setting this property type, ensure that you provide sufficient storage for one integer.

SCREEN_PROPERTY_RENDER_BUFFERS

A handle to the buffer or buffers available for rendering.

When retrieving this property type, ensure that you provide sufficient storage according to the API object type. The following API objects have this property, each with its own variant of this definition:

- pixmap: Only one buffer is allowed for a pixmap object. When retrieving SCREEN_PROPERTY_RENDER_BUFFERS for a pixmap object, ensure that you have sufficient storage for one `void` pointer.
- window: Multiple buffers may be available for rendering for a window object. When retrieving SCREEN_PROPERTY_RENDER_BUFFERS for a window, ensure that you have sufficient storage for one `void` pointer for each available buffer. Use the window property SCREEN_PROPERTY_RENDER_BUFFER_COUNT to determine the number of buffers that are available for rendering.

SCREEN_PROPERTY_ROTATION

A single integer that defines the current rotation of the API object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer. The following API objects have this property, each with its own variant of this definition:

- display: The current rotation of the display. The rotation value is one of: `0`, `90`, `180`, `270` degrees clockwise. It's used for the positioning and sizing of the display. Changing the display rotation, does not implicitly change any window properties.
- window: The current rotation of the window. Window rotation is absolute. In the configuration file, graphics.conf, the value of this property can be

set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- rotation = [rotation 0, 90, 180, 270]

SCREEN_PROPERTY_SATURATION

A single integer between [-128..127] that is used to adjust the saturation of a window; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have its property initialized to this value. The following is the usage for setting this property in graphics.conf:

- saturation = [saturation -128..127]

SCREEN_PROPERTY_SIZE

The size of the associated API object.

When retrieving or setting this property type, ensure that you provide sufficient storage according to the API object. The following API objects have this property, each with its own variant of this definition:

- buffer: A single integer that indicates the size, in bytes, of the buffer. When retrieving or setting this property type, ensure that you have sufficient storage for one integer.
- display: A pair of integers that define the width and height, in pixels, of the current video resolution. When retrieving this property type, ensure that you have sufficient storage for two integers. Note that the display size changes with the display rotation. For example, if the video mode is 1024x768 and the rotation is `0` degrees, the display size will indicate 1024x768. When the display rotation is set to `90` degrees, the display size will become 768x1024. (`SCREEN_PROPERTY_SIZE` can only be retrieved and not set for a display object)
- event: A pair of integers that define the width and height, in pixels, of the touch or contact area. This is only applicable for the following events:
  - `SCREEN_EVENT_MTOUCH_TOUCH`
  - `SCREEN_EVENT_MTOUCH_MOVE`
  - `SCREEN_EVENT_MTOUCH_RELEASE`

- window: A pair of integers that define the width and height, in pixels, of the window. When retrieving this property type, ensure that you have sufficient storage for two integers. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:
  - window-size = [width] x [height]

SCREEN_PROPERTY_SOURCE_POSITION

A pair of integers that define the x-and y- position of a source viewport within the window buffers.

When retrieving or setting this property type, ensure that you have sufficient storage for two integers. The following API objects have this property, each with its own variant of this defnition:

- event: This is only applicable for the following events:
  - `SCREEN_EVENT_MTOUCH_TOUCH`
  - `SCREEN_EVENT_MTOUCH_MOVE`
  - `SCREEN_EVENT_MTOUCH_RELEASE`
  - `SCREEN_EVENT_POINTER`
- window: The x and y coordinates of the top left corner of a rectangular region within the window buffer representing the source viewport of the window. This is the portion of the window buffer that is to be displayed. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:
  - source-position = [x-position], [y-position]

SCREEN_PROPERTY_SOURCE_SIZE

A pair of integers that define the width and height, pixels, of a source viewport within the window buffers.

When retrieving or setting this property type, ensure that you have sufficient storage for two integers. The following API objects have this property, each with its own variant of this definition:

- event: This is only applicable for the following events:

- SCREEN_EVENT_MTOUCH_TOUCH
- SCREEN_EVENT_MTOUCH_MOVE
- SCREEN_EVENT_MTOUCH_RELEASE

- window: The width and height of a rectangular region within the window buffer representing the source viewport of the window. This is the portion of the window buffer that is to be displayed. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:
  - source-size = [width] x [height]

SCREEN_PROPERTY_STATIC

A single integer that indicates whether or not the contents of a window are expected to change; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- static = [static 0,1]

SCREEN_PROPERTY_STRIDE

A single integer that indicates the number of bytes between the same pixels on adjacent rows; a property of a buffer object.

When retrieving or setting this property type, ensure that you provide sufficient storage for one integer.

SCREEN_PROPERTY_SWAP_INTERVAL

A single integer that specifies the minimum number of vsync periods between posts; a property of a window object, When retrieving or setting this property type, ensure that you have sufficient storage for one integer.

In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- interval = [swap interval]

SCREEN_PROPERTY_TRANSPARENCY

A single integer that defines the transparency type of an API object.

The following API objects have this property, each with its own variant of this definition:

- display: How multiple layers are combined. The transparencies that are applicable to a display object are:
  - SCREEN_TRANSPARENCY_SOURCE_COLOR
  - SCREEN_TRANSPARENCY_SOURCE_OVER

  When retrieving this property type for a display object, ensure that you have sufficient storage for one integer.
- window How the alpha channel of the window is used to combine a window with other windows or the background color underneath it. Although the window transparency property can be set, the actual transparency applied is dependent on hardware. If the hardware supports it, the transparency specified by this property will be applied, otherwise a best effort algorithm will be used to apply the window transparency. Transparency must be of the type *Screen transparency types* (p. 251). When retrieving or setting this property type, ensure that you have sufficient storage for one integer. This property is set to SCREEN_TRANSPARENCY_SOURCE_OVER as a convenience when you set SCREEN_PROPERTY_FORMAT to a format with alpha (e.g., SCREEN_FORMAT_RGBA4444). Therefore, if this is not your intention, then we recommend that you use a pixel format type that disregards the alpha channel (e.g. SCREEN_FORMAT_RGBX4444).

SCREEN_PROPERTY_TYPE

A single integer that indicates the type of the specified buffer object.

When retrieving this property type, ensure that you provide sufficient storage for one integer. The following API objects have this property, each with its own variant of this definition:

- device: The type of input device. Valid input device types are:
  - SCREEN_EVENT_POINTER
  - SCREEN_EVENT_KEYBOARD
  - SCREEN_EVENT_GAMEPAD
  - SCREEN_EVENT_JOYSTICK
  - SCREEN_EVENT_MTOUCH_TOUCH

- display: The type of display port. Valid display ports must be of type *Screen display types* (p. 316).
- event: The type of event. Valid event types must be of type *Screen event types* (p. 334).
- window: The type of window. Valid window types must be of type *Screen window types* (p. 395).

SCREEN_PROPERTY_USAGE

A single integer that is a bitmask indicating the intended usage for the buffers associated with the API object.

The default usage for a buffer is SCREEN_USAGE_READ|SCREEN_USAGE_WRITE. SCREEN_PROPERTY_USAGE must be a combination of type *Screen usage flag types* (p. 252). When retrieving or setting this property type, ensure that you have sufficient storage for one integer. Note that changing SCREEN_PROPERTY_USAGE affects the pipeline when the overlay usage bit (SCREEN_USAGE_OVERLAY) is added or removed. The following API objects have this property, and share the same definition:

- pixmap
- window
  - In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to this value. The following are valid usage flags and their implications that you can use to set this property in graphics.conf:
    - sw (read or write)
    - gles1 (OpenGL 1.X)
    - gles2 (OpenGL 2.X)
    - vg (OpenVG)
    - native (native API operations such as blits or fills)
    - rotation (re-configure orientation without re-allocation)
  - The following is the usage for setting this property in graphics.conf:
    - usage = [usage flag1, usage flag2, ...]
    - e.g., usage = sw, gles1

SCREEN_PROPERTY_USER_DATA

Four integers containing data associated with the user; a property of an event object.

SCREEN_PROPERTY_USER_DATA can be queried or set in association with an event of type SCREEN_EVENT_USER. When retrieving or setting this property type, ensure that you have sufficient storage for four integers.

SCREEN_PROPERTY_USER_HANDLE

A handle that is passed to the application window when events are associated with the window; a handle to an object to associate the API object with user data.

When retrieving or setting this property type, ensure that you have sufficient storage for one `void` pointer. The following API objects have this property, and share this same definition:

- device
- group
- window

SCREEN_PROPERTY_VISIBLE

A single integer that specifies whether or not the window is visible; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. In the configuration file, , the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following are valid settings that can be used for this property in graphics.conf:

- visible = true
- visible = false

SCREEN_PROPERTY_WINDOW

A pointer to a window.

When retrieving or setting this property type, ensure that you have sufficient storage for one `void` pointer. The following API objects have this property, each with its own variant of this definition:

- device The window on which the input device is focused. All input from the device will be directed to this particular window.

- event (SCREEN_PROPERTY_WINDOW can only be retrieved and not set for an event object)For the following events, SCREEN_PROPERTY_WINDOW refers to the window associated with the event:
  - SCREEN_EVENT_CREATE
  - SCREEN_EVENT_POST
  - SCREEN_EVENT_CLOSE
  - SCREEN_EVENT_UNREALIZE

  For the following events, SCREEN_PROPERTY_WINDOW refers to the window associated with the input device for which the event is intended:
  - SCREEN_EVENT_GAMEPAD
  - SCREEN_EVENT_JOYSTICK
  - SCREEN_EVENT_KEYBOARD
  - SCREEN_EVENT_MTOUCH_TOUCH
  - SCREEN_EVENT_MTOUCH_MOVE
  - SCREEN_EVENT_MTOUCH_RELEASE
  - SCREEN_EVENT_POINTER

  For the following event, SCREEN_PROPERTY_WINDOW refers to the window whose property is being set:
  - SCREEN_EVENT_PROPERTY

SCREEN_PROPERTY_RENDER_BUFFER_COUNT

A single integer that indicates he number of render buffers associated with the window; a property of a window object.

When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_ZORDER

A single integer that indicates the distance from the bottom that is used when ordering window groups amongst each other; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- order = [zorder]

SCREEN_PROPERTY_PHYSICAL_ADDRESS

A single long long integer that corresponds to the physical address of the buffer; a property of a buffer object.

This property is only valid when the buffer is physically contiguous. When retrieving or setting this property type, ensure that you provide sufficient storage for one long integer.

SCREEN_PROPERTY_SCALE_QUALITY

A single integer that indicates the amount of filtering performed by the windowing system when scaling is required to draw the window; a property of a window object.

The scale quality must be of type *Screen scale quality types* (p. 245). When retrieving or setting this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_SENSITIVITY

A single integer that indicates the window input behavior; a property of a window object.

The sensitivity must be of type *Screen sensitivity types* (p. 249) or an integer that is a bitmask combination of *Screen sensitivity masks* (p. 246). When retrieving or setting this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_MIRROR_MODE

A single integer that defines whether or not the display is currently in mirror mode.

Mirror mode indicates that the internal and external displays display the same signal. When retrieving or setting this property type, ensure that you have sufficient storage for one integer. The following API objects have this property, and share this same definition:

- display
- event (Applicable only to a `SCREEN_EVENT_DISPLAY` event)

SCREEN_PROPERTY_DISPLAY_COUNT

A single integer containing the number of displays associated with this context; a property of a context object.

When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_DISPLAYS

An array of display pointers; a property of a context object.

When retrieving this property type, ensure that you have sufficient storage for one `void` pointer for each display. Retrieve the `SCREEN_PROPERTY_DISPLAY_COUNT` property to find out how many displays are associated with this context; Once you know the number of displays, you can allocate sufficient storage to retrieve `SCREEN_PROPERTY_DISPLAYS`.

SCREEN_PROPERTY_CBABC_MODE

A single integer that indicates what the window content is; a property of a window object.

The content mode must be of type *Screen content mode types* (p. 190). When getting or setting this property type, ensure that you have sufficient storage for one integer. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following are valid settings that can be used for this property in graphics.conf:

- cbabc = none
- cbabc = video
- cbabc = ui
- cbabc = photo

SCREEN_PROPERTY_EFFECT

A single integer that indicates the effect associated with the specific event; a property of an event object.

Effects must be of type *Screen effect types*. When retrieving or setting this property type, ensure that you have sufficient storage for one integer. This property is only applicable for a `SCREEN_EVENT_EFFECT_COMPLETE` event.

SCREEN_PROPERTY_FLOATING

A single integer that indicates whether or not the window is a floating window; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_ATTACHED

A single integer that indicates whether or not the display is currently attached.

When retrieving or setting this property type, ensure you have sufficient storage for one integer. The following API objects have this property, each with its own variant of this definition:

- display: Indicates whether or not the display is connected. Display objects may exist in a context, but are not considered connected until they are attached.
- event:
  - In the case of the SCREEN_EVENT_DISPLAY event, this indicates that a display has changed its state; the display has either connected or disconnected.
  - In the case of the SCREEN_EVENT_DEVICE event, this indicates that either a new device has been created and is now conntected, or that a device has disconnected and been deleted. Unlike displays, device objects only exist in a context if they are attached. (SCREEN_PROPERTY_ATTACHED can only be retrieved and not set for device event.)

SCREEN_PROPERTY_DETACHABLE

A single integer that indicates whether or not the display can be detached; a property of a display object.

When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_NATIVE_RESOLUTION

A pair of integers that define the width and height of the native video resolution; a property of a display object.

When retrieving this property type, ensure that you have sufficient storage for two integers.

SCREEN_PROPERTY_PROTECTION_ENABLE

A single integer that indicates whether or not content protection is enabled for the API object.

You require a secure link in order to have protection enabled. When retrieving or setting this property type, ensure that you have sufficient storage for one integer. The following API objects have this property, each with its own variant of this definition:

- display: Indicates whether or not content protection is needed for the window(s) on the display. Content protection is considered enabled as long as one window on the display has its content protection enabled and its SCREEN_PROPERTY_VISIBLE property indicates that the window is visible. The SCREEN_PROPERTY_PROTECTION_ENABLE property of a display is dynamic; its value depends on the SCREEN_PROPERTY_PROTECTION_ENABLE property of the window(s) that are on the display. SCREEN_PROPERTY_PROTECTION_ENABLE can only be retrieved and not set for a display object.
- window: Indicates whether or not authentication is to be requested before the content of the window can be displayed. Authentication is requested when the window is posted and its SCREEN_PROPERTY_VISIBLE property indicates that the window is visible.
- event: (Applicable only to a SCREEN_EVENT_DISPLAY event) Indicates that a disabling of content protection is detected. This is likely due to the loss of a secure link to the display.

SCREEN_PROPERTY_SOURCE_CLIP_POSITION

A pair of integers that define the x- and y- position of a clipped source rectangular viewport within the window buffers; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for two integers.

SCREEN_PROPERTY_PHYSICAL_SIZE

A pair of integers that define the width and height, in millimeters, of the display; a property of a display object.

When retrieving this property type, ensure that you have sufficient storage for two integers.

SCREEN_PROPERTY_FORMAT_COUNT

A single integer that indicates the number of formats that the display supports; a property of a display object.

When retrieving this property type, ensure that you have sufficient storage for at least one integer.

SCREEN_PROPERTY_FORMATS

An array of integers of size SCREEN_PROPERTY_FORMAT_COUNT that defines the formats supported by the display; a property of a display object.

If the display has many layers, the list is the union of all the formats supported on all layers. Formats are of type *Screen pixel format types* (p. 196). When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_SOURCE_CLIP_SIZE

A pair of integers that define the width and height, in pixels, of a clipped source rectangular viewport within he window buffers; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for two integers.

SCREEN_PROPERTY_TOUCH_ID

A single integer that indicates the multi-touch contact id associated with the specific event; a property of an event object.

When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_VIEWPORT_POSITION

A pair of integers that define the x and y position of a rectangular region within the API object.

When retrieving or setting this property type, ensure that you have sufficient storage for two integers. The following API objects have this property, each with its own variant of this definition:

- display: The x and y coordinates of the top left corner of a rectangular region within the display that is intended to be mapped to and redrawn to the display. In order for you to access this display property, you need to be working within a privileged context. That is, a the context in which you are accessing this display property must have been created with at least the bit mask of SCREEN_DISPLAY_MANAGER_CONTEXT.
- window: The x and y coordinates of the top left corner of a virtual viewport. The virtual viewport is typically used to achieve the effect of scrolling or panning a source whose size is larger than the size of your window buffer.

SCREEN_PROPERTY_VIEWPORT_SIZE

A pair of integers that define the width and height of a rectangular region within the API object.

When retrieving or setting this property type, ensure that you have sufficient storage for two integers. The following API objects have this property, each with its own variant of this definition:

- display: The width and height, in pixels, of a rectangular region within the display that is intended to be mapped to and redrawn to the display. In order for you to access this display property, you need to be working within a privileged context. That is, a the context in which you are accessing this display property must have been created with at least the bit mask of SCREEN_DISPLAY_MANAGER_CONTEXT.
- window: The width and height, in pixels, of a virtual viewport. The virtual viewport is typically used to achieve the effect of scrolling or panning a source whose size is larger than the size of your window buffer.

SCREEN_PROPERTY_TOUCH_ORIENTATION

A single integer that indicates the multi-touch orientation associated with the specific event; a property of an event object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. This property is only applicable for the following events:

- SCREEN_EVENT_MTOUCH_TOUCH
- SCREEN_EVENT_MTOUCH_MOVE

- SCREEN_EVENT_MTOUCH_RELEASE

SCREEN_PROPERTY_TOUCH_PRESSURE

A single integer that indicates the multi-touch pressure associated with the specific event; a property of an event object.

When retrieving or setting this property type, ensure that you have sufficient storage or one integer. This property is only applicable for the following events: events:

- SCREEN_EVENT_MTOUCH_TOUCH
- SCREEN_EVENT_MTOUCH_MOVE
- SCREEN_EVENT_MTOUCH_RELEASE

SCREEN_PROPERTY_TIMESTAMP

A single long long integer that indicates a timestamp associated with the API object.

When retrieving or setting this property type, ensure that you have sufficient storage for one long long integer. It is important to note that screen uses the realtime clock and not the monotonic clock when calculating the timestamp. The following API objects have this property, each with its own variant of this definition:

- event: The timestamp at which the event was received by screen (SCREEN_PROPERTY_TIMESTAMP can only be retrieved and not set for an event object).
- window: The timestamp to indicate the start of a frame. This timestamp can be used by the application to measure the elapsed time taken to perform functions of interest. For example, the application can measure the time between when the timestamap is set and when the window is posted (e.g., when OpenGL swap buffers). This timestamp allows for the application to track CPU time. The application can set the timestamp to any specific time. Then, the application uses the screen_get_window_property_llv() function to retrieve the SCREEN_PROPERTY_METRICS property of the window to look at the timestamp for comparison to the set timestamp.

SCREEN_PROPERTY_SEQUENCE_ID

A single integer that indicates the the multi-touch sequence id associated with the specific event; a property of a Screen API event object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. This property is only applicable for the following events:

- SCREEN_EVENT_MTOUCH_TOUCH
- SCREEN_EVENT_MTOUCH_MOVE
- SCREEN_EVENT_MTOUCH_RELEASE
- SCREEN_EVENT_KEYBOARD

SCREEN_PROPERTY_IDLE_MODE

A single integer indicating the idle mode of the window; a property of a window object.

The idle mode must be of type *Screen idle mode types* (p. 193). When retrieving or setting this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_IDLE_STATE

A single integer that indicates the idle state of the API object.

The idle state will be `1` if the system is idle, indicating that no input was received after the idle timeout period (`SCREEN_PROPERTY_IDLE_TIMEOUT`). The idle state will be `0`, if an input event was received prior to the idle timeout period expiring. When retrieving this property type, ensure that you have sufficient storage for one integer. The following API objects have this property, each sharing a similiar definition:

- display: The idle state that is applicable to the entire display.
- event (Applicable only to a `SCREEN_EVENT_IDLE` event): Indicates that an idle state change has taken place for either a display or group object. Query the `SCREEN_PROPERTY_OBJECT_TYPE` property of the event to determine the object type of this event.
- group : The idle state that is applicable to only the group. A group is considered in idle state if none of the windows that are part of the group have received input after the idle timeout period for the group.

SCREEN_PROPERTY_KEEP_AWAKES

A single integer that indicates the number of windows with an idle mode of type SCREEN_IDLE_MODE_KEEP_AWAKE that are visible on a display; a property of a display object.

When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_IDLE_TIMEOUT

A single long long integer that indicates the amount of time, in seconds, after which the system will enter an idle state.

When retrieving or setting this property type, ensure that you have sufficient storage for one long long integer. The following API objects have this property, each sharing a similar definition:

- context: The amount of time after which the display of the context will enter an idle state.
- display: The amount of time after which the display will enter an idle state.
- group: The amount of time after which the group will enter in an idle state.

SCREEN_PROPERTY_KEYBOARD_FOCUS

A window handle which corresponds to the window that currently has keyboard focus.

When retrieving or setting this property type, ensure that you have sufficient storage according to the definition of the property for the specific API object. The following API objects have this property, each with its own variant of this definition:

- context: A handle to the top-level window (application window) on the display that currently has the keyboard focus. You must be working within a privileged context of type `SCREEN_WINDOW_MANAGER_CONTEXT` to be able to set this property.
- display: A handle to the top-level window (application window) on the display that currently has the keyboard focus. You must be working within a privileged context of type `SCREEN_WINDOW_MANAGER_CONTEXT` to be able to set this property.
- group: A handle to the immediate window in the group that currently has the keyboard focus. You must be the owner of the group, screen_group_t, to be able to set this property.
- window: A single integer that indicates whether or not the window currently has the keyboard focus. (`SCREEN_PROPERTY_KEYBOARD_FOCUS` can only be retrieved and not set for a window object)

SCREEN_PROPERTY_MTOUCH_FOCUS

A window handle which corresponds to the window that currently has mtouch focus.

When retrieving or setting this property type, ensure that you have sufficient storage for one `void` pointer. The following API objects have this property, each with its own variant of this definition:

- context: A handle to the top-level window (application window) on the display that currently has the mtouch focus. You must be working within a privileged context of type `SCREEN_WINDOW_MANAGER_CONTEXT` to be able to set this property.
- display: A handle to the top-level window (application window) on the display that currently has the mtouch focus. You must be working within a privileged context of type `SCREEN_WINDOW_MANAGER_CONTEXT` to be able to set this property.
- group: A handle to the immediate window in the group that currently has the mtouch focus. You must be the owner of the group, screen_group_t, to be able to set this property.

SCREEN_PROPERTY_POINTER_FOCUS

A window handle which corresponds to the window that currently has pointer focus.

When etrieving or setting this property type, ensure that you have sufficient storage for one `void` pointer. The following API objects have this property, each with its own variant of this definition:

- context: A handle to the top-level window (application window) on the display that currently has the pointer focus. You must be working within a privileged context of type `SCREEN_WINDOW_MANAGER_CONTEXT` to be able to set this property.
- display: A handle to the top-level window (application window) on the display that currently has the pointer focus. You must be working within a privileged context of type `SCREEN_WINDOW_MANAGER_CONTEXT` to be able to set this property.
- group: A handle to the immediate window in the group that currently has the pointer focus. You must be the owner of the group, screen_group_t, to be able to set this property.

SCREEN_PROPERTY_ID

A single integer that indicates the identification of the display; a property of a display object.

When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_POWER_MODE

A single integer that defines the power mode.

Power modes must be of type *Screen power mode types* (p. 199). When retrieving or setting this property type, ensure that you have sufficient storage for one integer. The following API objects have this property, and share this same definition:

- device
- display

SCREEN_PROPERTY_MODE_COUNT

A single integer that indicates the number of modes supported by the display; a property of display object.

When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_MODE

A pointer to a structure of type screen_display_mode_t whose content is based on the current video mode.

When retrieving or setting this property type, ensure that you have sufficient storage for screen_display_mode_t for each mode. Retrieve the `SCREEN_PROPERTY_MODE_COUNT` property to find out how many modes are supported by this display; Once you know the number of displays, you can allocate sufficient storage to retrieve `SCREEN_PROPERTY_MODE`. When setting this property type, you can pass `SCREEN_MODE_PREFERRED_INDEX` to fall back to the default video mode without having to first query all the modes supported by the display to find the one with `SCREEN_MODE_PREFERRED` set in `flags` of screen_display_mode_t. The following API objects have this property, and share this same definition:

- display
- event (Applicable only to a `SCREEN_EVENT_DISPLAY` event)

SCREEN_PROPERTY_CLIP_POSITION

A pair of integers that define the x- and y- position of a clipped rectangular viewport within the window buffers; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for two integers. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- clip-position = [x-position], [y-position]

SCREEN_PROPERTY_CLIP_SIZE

A pair of integers that define the width and height , in pixels, of a clipped rectangular viewport within the window buffers; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for two integers. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- clip-size = [width] x [height]

SCREEN_PROPERTY_COLOR

A single integer that indicates the background color of the window; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. In the configuration file, graphics.conf, the value of this property can be set so that windows of a specified class will have this property initialized to the value. The following is the usage for setting this property in graphics.conf:

- color = [window background color]

SCREEN_PROPERTY_MOUSE_WHEEL

A single integer that indicates the number and direction of mouse wheel ticks in a vertical direction; a property of an event object.

Wheel ticks in an upward direction are indicated by a negative (-) number, while those in the downward direction are indicated by a positive (+) one. When retrieving or setting this property type, ensure that you have sufficient storage for one integer. This property is only applicable for a `SCREEN_EVENT_POINTER` event.

SCREEN_PROPERTY_CONTEXT

A pointer to the context associated with the API object.

When retrieving this property type, ensure that you have sufficient storage for one integer. The following API objects have this property, and share this same definition:

- device
- display
- event
- group
- pixmap
- window

SCREEN_PROPERTY_DEBUG

A single integer that enables an on-screen plot or a list of window statistics as a debugging aid; a property of a window object.

The debug type must be a bitmask that represents a combination of the types *Screen debug graph types* (p. 292). When retrieving or setting this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_ALTERNATE_WINDOW

A handle to an alternate window; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for a `void` pointer.

SCREEN_PROPERTY_DEVICE_COUNT

A single integer containing the number of display devices associated with this context; a property of a context object.

When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_BUFFER_POOL

Deprecated:

This property has been deprecated.

Do not use.

SCREEN_PROPERTY_OBJECT_TYPE

A single integer that indicates the object type associated the with the specific event; a property of an event object.

When retrieving this property type, ensure that you have sufficient storage for one integer. Object types must be of type *Screen object types* (p. 196). This property is only applicable for a `SCREEN_EVENT_PROPERTY` event.

SCREEN_PROPERTY_DEVICES

An array of device pointers; a property of a context object.

When retrieving this property type, ensure that you have sufficient storage for one `void` pointer for each device. Retrieve the `SCREEN_PROPERTY_DEVICE_COUNT` property to find out how many devices are associated with this context; Once you know the number of devices, you can allocate sufficient storage to retrieve `SCREEN_PROPERTY_DEVICES`.

SCREEN_PROPERTY_KEYMAP_PAGE

A single integer that indicates which page of a multi-page keymap must be used to translate scan codes into key caps and key symbols.

Setting the keymap page on a USB or Bluetooth keyboard has no effect. Setting the keymap page on an external keyboard device created by an input provider context will cause the input provider context to receive a notification of the change.

SCREEN_PROPERTY_SELF_LAYOUT

A single integer that indicates whether or not the window has self layout capabilities; a property of a window object.

When set to true(1), the owner of the window can change window properties that could otherwise only be changed by its parent. When set to false(0), the owner of the window is permitted to change only its owner window

properties. Note that only the parent window, or a window manager, can set this property. When retrieving or setting this property type, ensure that you have sufficient storage for one integer. The following are parent window properties that can be set by the owner of the window only if SCREEN_PROPERTY_SELF_LAYOUT is set to true(1):

- SCREEN_PROPERTY_SIZE
- SCREEN_PROPERTY_POSITION
- SCREEN_PROPERTY_CLIP_SIZE
- SCREEN_PROPERTY_CLIP_POSITION
- SCREEN_PROPERTY_VISIBLE
- SCREEN_PROPERTY_ZORDER
- SCREEN_PROPERTY_DISPLAY
- SCREEN_PROPERTY_GLOBAL_ALPHA

SCREEN_PROPERTY_GROUP_COUNT

A single integer containing the number of groups associated with this context a property of a context object.

When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_GROUPS

An array of group pointers; a property of a context object.

When retrieving this property type, ensure that you have sufficient storage for one `void` pointer for each group. Retrieve the SCREEN_PROPERTY_GROUP_COUNT property to find out how many groups are associated with this context; once you know the number of groups, you can allocate sufficient storage to retrieve SCREEN_PROPERTY_GROUPS.

SCREEN_PROPERTY_PIXMAP_COUNT

A single integer containing the number of pixmaps associated with this context; a property of a context object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_PIXMAPS

An array of pixmap pointers; a property of a context object.

When retrieving this property type, ensure that you have sufficient storage for one `void` pointer for each pixmap. Retrieve the `SCREEN_PROPERTY_PIXMAP_COUNT` property to find out how many pixmaps are associated with this context; Once you know the number of pixmaps, you can allocate sufficient storage to retrieve `SCREEN_PROPERTY_PIXMAPS`.

SCREEN_PROPERTY_WINDOW_COUNT

A single integer containing the number of windows associated with this context; a property of a context object.

When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_WINDOWS

An array of window pointers; a property of a context object.

When retrieving this property type, ensure that you have sufficient storage for one `void` pointer for each window. Retrieve the `SCREEN_PROPERTY_WINDOW_COUNT` property to find out how many windows are associated with this context; Once you know the number of windows, you can allocate sufficient storage to retrieve `SCREEN_PROPERTY_WINDOWS`.

SCREEN_PROPERTY_KEYMAP

A character string specifying a keymap.

When retrieving or setting this property type, ensure that you provide a character buffer. The following API objects have this property, each with its own variant of the definition:

- context: The default keymap. Unless specifically assigned, this keymap is applied to all input devices.
- device: The keymap that is assigned to the specified input device. The keymap is only applicable to that device and is not persistent. For example, if the input device is removed and then replaced, the default keymap will be applied to it until a keymap is specifically set for the input device again.

SCREEN_PROPERTY_MOUSE_HORIZONTAL_WHEEL

A single integer that indicates the number and direction of mouse wheel ticks in a horizontal direction; a property of an event object.

Wheel ticks toward the left are indicated by a negative (-) number, while those toward the right are indicated by a positive (+) one. When retrieving or setting this property type, ensure that you have sufficient storage for one integer. This property is only applicable for a `SCREEN_EVENT_POINTER` event.

SCREEN_PROPERTY_TOUCH_TYPE

A single integer that indicates the touch associated the with the specific event; a property of an event object.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. Touch types must be of type *Screen touch types* (p. 250). This property is only applicable for the following events:

- `SCREEN_EVENT_MTOUCH_TOUCH`
- `SCREEN_EVENT_MTOUCH_MOVE`
- `SCREEN_EVENT_MTOUCH_RELEASE`

SCREEN_PROPERTY_NATIVE_IMAGE

A pointer to the image; a property of a buffer object.

When retrieving or setting this property type, ensure that you have sufficient storage for one `void` pointer.

SCREEN_PROPERTY_SCALE_FACTOR

A single integer that indicates the number of bits of the desired sub-pixel precision.

When retrieving or setting this property type, ensure that you have sufficient storage for one integer. The following API objects have this property:

- event: The default value is `0`.
- window: The default value is `16`. Note that setting this property, `SCREEN_PROPERTY_SCALE_FACTOR`, prior to retrieving `SCREEN_PROPERTY_TRANSFORM` will affect the values for the transformation matrix.

SCREEN_PROPERTY_DPI

A pair of integers which represent the dpi measurement; a property of a display object.

The dpi is calculated from the physical dimensions and resolution of the display. When retrieving this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_METRIC_COUNT

A single integer that indicates the number of metrics associated with an API object.

Note that the actual value of the number of metrics may vary between API objects. When retrieving this property type, ensure that you have sufficient storage for one integer. The following API objects have this property, and share this same definition:

- device
- display
- pixmap
- window

SCREEN_PROPERTY_METRICS

A array of metrics associated with an API object.

Note that the size of the array of metrics may vary between API objects. When retrieving this property type, ensure that you have sufficient storage for one `void` pointer for each metric. Retrieve the `SCREEN_PROPERTY_METRIC_COUNT` property to find out how many metrics are associated with the API object; once you know the number of metrics, you can allocate sufficient storage to retrieve `SCREEN_PROPERTY_METRICS`. The following API objects have this property, and share this same definition:

- device
- display
- pixmap
- window

SCREEN_PROPERTY_BUTTON_COUNT

A single integer that indicates the number of buttons on the input device; a property of a device object.

You can set this property on input devices you create. In the case of mtouch, this property refers to the number of buttons on the stylus, not necessarily the touch points. All users can query the value on the created device. When retrieving or setting this property type, ensure that you have sufficient storage for one integer.

SCREEN_PROPERTY_VENDOR

A string that can be used to identify the vendor of the specified API object.

When retrieving or setting this property type, ensure that you provide a character buffer. The following API objects have his property, and share this same definition:

- device
- display

SCREEN_PROPERTY_PRODUCT

A string that can be used to identify the product name of the specified API object.

When retrieving or setting this property type, ensure that you provide a character buffer. The following API objects have his property, and share this same definition:

- device
- display

SCREEN_PROPERTY_BRUSH_CLIP_POSITION

A pair of integers that define the x- and y- position of a clipped rectangular area within the window buffers where brush strokes are allowed to be drawn; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for two integers.

SCREEN_PROPERTY_BRUSH_CLIP_SIZE

A pair of integers that define the width and height, in pixels, of a clipped rectangular area within the window buffers where brush strokes are allowed to be drawn; a property of a window object.

When retrieving or setting this property type, ensure that you have sufficient storage for two integers.

SCREEN_PROPERTY_ANALOG0

The x, y and z values for one analog controller; a property of an event object.

For analog controllers that do not have three degrees of freedom, only x and y values are valid; z will have a value of `0`. Regardless of two or three degress of freedom of your analog controller(s), when retrieving or setting this property type, ensure that you have sufficient storage for three integers. This property is only applicable for the following events:

- `SCREEN_EVENT_GAMEPAD`:
- `SCREEN_EVENT_JOYSTICK`:

SCREEN_PROPERTY_ANALOG1

The rx, ry, and rz values for a second analog controller for the SCREEN_EVENT_GAMEPAD event.

For analog controllers that do not have three degrees of freedom, only x and y values are valid; z will have a value of `0`. Regardless of two or three degress of freedom of your analog controller(s), when retrieving or setting this property type, ensure that you have sufficient storage for three integers.

SCREEN_PROPERTY_BRUSH

The pixmap that contains the brush to be used when the window property, SCREEN_PROPERTY_SENSITIVITY, has a brush bit set and the corresponding type of input event is delivered to the window; a property of a window object.

The sensitivity values that can be set to enable brush drawing in your sensitivity mask are:

- `SCREEN_SENSITIVITY_MASK_POINTER_BRUSH` for a mouse
- `SCREEN_SENSITIVITY_MASK_FINGER_BRUSH` for touch
- `SCREEN_SENSITIVITY_MASK_STYLUS_BRUSH` for a stylus

The pixmap can use an RGB color format for color information. The alpha channel can be used to define the brush shape. The pixmap buffer size will determine the size of the brush. No drawing will occur if the window has a

brush sensitivity bit set and no brush pixmap. When retrieving or setting this property type, ensure that you have sufficient storage for one `void` pointer.

SCREEN_PROPERTY_TRANSFORM

A set of integers that represent the 3x3 transformation matrix used to convert buffer or device coordinates to display coordinates.

The transformation matrix is stored in row-major order. This property is typically used when you have a known coordinate and need to know where it would be located on the display screen. Screen creates a vector V=(X, Y, 1.0) from the raw coordinates, (X, Y). Then, Screen multiplies the transformation matrix T with this vector (T x V) to achieve a resultant vector of (Xa, Ya, Wa). This resultant vector is then divided by Wa to result in (Xab, Yab, 1.0), which provides (Xab, Yab) as the transformed display coordinates. When retrieving this property, ensure that you have sufficient storage for 9 integers. Note that setting the SCREEN_PROPERTY_SCALE_FACTOR property of a window or device prior to retrieving this property affects the values of this transformation matrix. The following API objects have this property, and share this same definition:

- device (input devices of type SCREEN_EVENT_MTOUCH_TOUCH) When setting this property, the last row of the transformation matrix must be [0, 0, non-zero].
- window (SCREEN_PROPERTY_TRANSFORM can only be retrieved and not set for a window object)

SCREEN_PROPERTY_TECHNOLOGY

An integer that is used to identify the technology used by a particular object handle; a property of a display object.

When retrieving this property type, ensure that you provide enough storage for one integer.

SCREEN_PROPERTY_REFERENCE_COLOR

An integer that is a color to be used with specific transparency modes; a property of a window or display object.

The transparency modes that use this property are:

- SCREEN_TRANSPARENCY_TEST
- SCREEN_TRANSPARENCY_REVERSED_TEST

- SCREEN_TRANSPARENCY_SOURCE_COLOR

When the transparency mode is SCREEN_TRANSPARENCY_TEST or SCREEN_TRANSPARENCY_REVERSED_TEST, transparency is applied to each pixel based on a comparison with the reference color.

When the transparency mode is SCREEN_TRANSPARENCY_SOURCE_COLOR, transparency is applied to each pixel matching the reference color.

When retrieving or setting this property, ensure that you have sufficient storage for one integer. If this attribute is not specified, then a default of `0` will be used.

Library:

`libscreen`

Description:

Full read/write access to Screen API object properties is product dependent.

Screen scale quality types

Types of scaling qualities.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_QUALITY_NORMAL = 0
      SCREEN_QUALITY_FASTEST = 1
      SCREEN_QUALITY_NICEST = 2
};
```

Data:

SCREEN_QUALITY_NORMAL

The suggested amount of filtering that is slower than SCALE_QUALITY_FASTEST, but should have better quality.

SCREEN_QUALITY_FASTEST

The suggested amount of filtering that is faster than SCALE_QUALITY_NORMAL, but may have reduced quality.

SCREEN_QUALITY_NICEST

The suggested amount of filtering that is slower than SCALE_QUALITY_NORMAL, but should have better quality.

Library:

`libscreen`

Description:

Each enumerator specifies the suggested amount of filtering to be performed by the windowing system when scaling is required to draw the window. This amount of filtering is not a constant quantity; it is specfied relative to each of the other possible scale qualities.

Screen sensitivity masks

Types of sensitivity masks.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_SENSITIVITY_MASK_ALWAYS = (1 << 0)
      SCREEN_SENSITIVITY_MASK_NEVER = (2 << 0)
      SCREEN_SENSITIVITY_MASK_NO_FOCUS = (1 << 3)
      SCREEN_SENSITIVITY_MASK_FULLSCREEN = (1 << 4)
      SCREEN_SENSITIVITY_MASK_CONTINUE = (1 << 5)
      SCREEN_SENSITIVITY_MASK_STOP = (2 << 5)
      SCREEN_SENSITIVITY_MASK_POINTER_BRUSH = (1 << 7)
      SCREEN_SENSITIVITY_MASK_FINGER_BRUSH = (1 << 8)
      SCREEN_SENSITIVITY_MASK_STYLUS_BRUSH = (1 << 9)
      SCREEN_SENSITIVITY_MASK_OVERDRIVE = (1 << 10)
};
```

Data:

SCREEN_SENSITIVITY_MASK_ALWAYS

Pointer and touch events are always forwarded to the window's context if they interect with the window - regardless of transparency.

The window receives keyboard, gamepad, joystick events if it has input focus. Raising a window, pointer or multi-touch release event in that window will cause it to acquire input focus.

SCREEN_SENSITIVITY_MASK_NEVER

The window never receives pointer or multi-touch events.

The window never acquires input focus, even after it has been raised. The window will only receive input events that are directly injected into it from outside sources.

SCREEN_SENSITIVITY_MASK_NO_FOCUS

Pointer and touch events are forwarded to the window's context if they intersect the window and are in an area of the window that is not fully transparent.

The window does not acquire input focus after being raised or after a pointer or multi-touch release event occurs. Therefore, the window will not receive keyboard, gamepad, or joystick input unless it is sent directly into the window from an outside source.

SCREEN_SENSITIVITY_MASK_FULLSCREEN

Pointer and touch events are forwarded to the window's context no matter where they are on the screen.

The window is considered full screen for the purposes of input hit tests. Transparency is ignored. The window will receive keyboard, gamepad, and joystick events as long as the window is visible.

SCREEN_SENSITIVITY_MASK_CONTINUE

Windows underneath this window can receive pointer or multi-touch events even if this window has input focus.

SCREEN_SENSITIVITY_MASK_STOP

The window never receives pointer or multi-touch events.

The window never acquires input focus, even after it has been raised. The window will only receive input events that are directly injected into it from outside sources.

SCREEN_SENSITIVITY_MASK_POINTER_BRUSH

The window receives pointer events, even in areas of transparency, if the source coordinates of the event are within the brush clip rectangle.

This mode supercedes SCREEN_SENSITIVITY_MASK_NEVER. The windowing system also draws brush strokes based on the pointer events directly onto the screen and the window buffer.

SCREEN_SENSITIVITY_MASK_FINGER_BRUSH

The window receives multi-touch events with a finger contact type, even in areas of transparency, if the source coordinates of the event are within the brush clip rectangle.

This mode supercedes SCREEN_SENSITIVITY_MASK_NEVER. The windowing system also draws brush strokes based on the touch events directly onto the screen and the window buffer. Multiple contacts will cause multiple brush strokes to be drawn.

SCREEN_SENSITIVITY_MASK_STYLUS_BRUSH

The window receives multi-touch events with a stylus contact type, even in areas of transparency, if the source coordinates of the event are within the brush clip rectangle.

This mode supercedes SCREEN_SENSITIVITY_MASK_NEVER. The windowing system also draws brush strokes based on the touch events directly onto the screen and the window buffer. Multiple contacts will cause multiple brush strokes to be drawn.

SCREEN_SENSITIVITY_MASK_OVERDRIVE

Setting this bit causes the system to go into overdrive when the window gets an input event.

The effect of this sensitivity mask depends on the power management algorithms in place and on the platform in general.

Library:

`libscreen`

Description:

These masks are intended to be combined in a single integer bitmask representing combinations of desired senstivites to be applied to a window.

Screen sensitivity types

Types of sensitivities.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_SENSITIVITY_TEST = 0
      SCREEN_SENSITIVITY_ALWAYS = 1
      SCREEN_SENSITIVITY_NEVER = 2
      SCREEN_SENSITIVITY_NO_FOCUS = 3
      SCREEN_SENSITIVITY_FULLSCREEN = 4
};
```

Data:

SCREEN_SENSITIVITY_TEST

The default sensitivity.

Pointer and multi-touch events are forwarded to the window's context if they intersect with the window and are in an area of the window that is not fully transparent. The window receives keyboard, gamepad, joystick events if it has input focus. Raising a window, pointer or multi-touch release event in the window will cause the window to acquire input focus.

SCREEN_SENSITIVITY_ALWAYS

That pointer and touch events are always forwarded to the window's context if they interect with the window - even if the window is transparent in that area.

The window receives keyboard, gamepad, joystick events if it has input focus. Raising a window, pointer or multi-touch release event in that window will cause it to acquire input focus.

SCREEN_SENSITIVITY_NEVER

The window never receives pointer or multi-touch events.

The window never acquires input focus, even after it has been raised. The window will only receive input events that are directly injected into it from outside sources.

SCREEN_SENSITIVITY_NO_FOCUS

Pointer and touch events are forwarded to the window's context if they intersect the window and are in an area of the window that is not fully transparent.

The window does not acquire input focus after being raised or after a pointer or multi-touch release event occurs. Therefore, the window will not receive keyboard, gamepad, or joystick input unless it is sent directly into the window from an outside source.

SCREEN_SENSITIVITY_FULLSCREEN

Pointer and touch events are forwarded to the window's context no matter where they are on the screen.

The window is considered full screen for the purposes of input hit tests. Transparency is ignored. The window will receive keyboard, gamepad, and joystick events if it has input focus. Raising the window or a pointer or multi-touch release event in the window will cause it to acquire input focus.

Library:

`libscreen`

Description:

Screen touch types

Types of touch.

Synopsis:

```
#include <screen/screen.h>

 enum {
      SCREEN_TOUCH_FINGER = 0
      SCREEN_TOUCH_STYLUS = 1
};
```

Data:

SCREEN_TOUCH_FINGER

SCREEN_TOUCH_STYLUS

Library:

libscreen

Description:

Screen transparency types

Types of window transparencies.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_TRANSPARENCY_SOURCE = 0
      SCREEN_TRANSPARENCY_TEST = 1
      SCREEN_TRANSPARENCY_SOURCE_COLOR = 2
      SCREEN_TRANSPARENCY_SOURCE_OVER = 3
      SCREEN_TRANSPARENCY_NONE = 4
      SCREEN_TRANSPARENCY_DISCARD = 5
      SCREEN_TRANSPARENCY_REVERSED_TEST = 6
};
```

Data:

SCREEN_TRANSPARENCY_SOURCE

Destination pixels are replaced by source pixels, including the alpha channel.

SCREEN_TRANSPARENCY_TEST

Destination pixels are replaced by source pixels when the source pixel value is greater than the reference value.

See SCREEN_PROPERTY_REFERENCE_COLOR.

SCREEN_TRANSPARENCY_SOURCE_COLOR

Destination pixels are replaced by source pixels when the source color does not match the reference color value.

See SCREEN_PROPERTY_REFERENCE_COLOR.

SCREEN_TRANSPARENCY_SOURCE_OVER

Typical alpha blending; the source pixels are blended over the destination pixels.

SCREEN_TRANSPARENCY_NONE

Destination pixels are replaced by fully-visible source pixels.

SCREEN_TRANSPARENCY_DISCARD

Source is considered completely transparent; the destination is not modified.

SCREEN_TRANSPARENCY_REVERSED_TEST

Destination pixels are replaced by source pixels when the source pixel value is less than the reference value.

See SCREEN_PROPERTY_REFERENCE_COLOR.

Library:

libscreen

Description:

Screen usage flag types

Types of usage flags.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_USAGE_READ = (1 << 1)
      SCREEN_USAGE_WRITE = (1 << 2)
      SCREEN_USAGE_NATIVE = (1 << 3)
      SCREEN_USAGE_OPENGL_ES1 = (1 << 4)
      SCREEN_USAGE_OPENGL_ES2 = (1 << 5)
      SCREEN_USAGE_OPENGL_ES3 = (1 << 11)
      SCREEN_USAGE_OPENVG = (1 << 6)
      SCREEN_USAGE_VIDEO = (1 << 7)
      SCREEN_USAGE_CAPTURE = (1 << 8)
      SCREEN_USAGE_ROTATION = (1 << 9)
      SCREEN_USAGE_OVERLAY = (1 << 10)
};
```

Data:

SCREEN_USAGE_READ

Flag to indicate that buffer(s) associated with the API object can be read from.

SCREEN_USAGE_WRITE

Flag to indicate that buffer(s) associated with the API object can be written to.

SCREEN_USAGE_NATIVE

Flag to indicate that buffer(s) associated with the API object can be used for native API operations.

If using blits or fills, this flag must be set on the API object.

SCREEN_USAGE_OPENGL_ES1

Flag to indicate that OpenGL ES 1.X is used for rendering the buffer associated with the API object.

SCREEN_USAGE_OPENGL_ES2

Flag to indicate that OpenGL ES 2.X is used for rendering the buffer associated with the API object.

SCREEN_USAGE_OPENGL_ES3

Flag to indicate that OpenGL ES 3.X is used for rendering the buffer associated with the API object.

SCREEN_USAGE_OPENVG

Flag to indicate that OpenVG is used for rendering the buffer associated with the API object.

SCREEN_USAGE_VIDEO

Flag to indicate that the buffer can be written to by a video decoder.

SCREEN_USAGE_CAPTURE

Flag to indicate that the buffer can be written to by capture devices (such as cameras, analog-to-digital-converters, ...), and read by a hardware video encoder.

SCREEN_USAGE_ROTATION

Flag to indicate that the buffer can be re-configured from landscape to portrait orientation without reallocation.

Rotation

SCREEN_USAGE_OVERLAY

Flag to indicate the use of a non-composited layer.

The Screen API uses a composited layer by default. The SCREEN_USAGE_OVERLAY flag is used to override this default behaviour to use a non-composited layer instead. Note that when the overlay usage bit is added or removed, then changing SCREEN_USAGE_OVERLAY affects the pipeline. Set this SCREEN_USAGE_OVERLAY flag when you are targeting a non-composited pipeline.

Library:

`libscreen`

Description:

Usage flags are used when allocating buffers. Depending on the usage, different constraints such as width, height, stride granularity or special alignment must be observed. The usage is also valuable in determining the amount of caching that can be set on a particular buffer.

Blits (screen.h)

Blit API functions and properties are used when combining buffers.

From Blit API functions, the connection to screen should have been acquired with the function *screen_create_context()*.

Any source and destination buffers required by the Blit API functions are specified with buffer handles. These handles are typcially acquired by querying the *SCREEN_PROPERTY_RENDER_BUFFERS* property of a pixmap or window with the *screen_get_pixmap_property()* or *screen_get_window_property()* functions respectively.

Screen blit types

Types of blit attributes.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_BLIT_END = 0
      SCREEN_BLIT_SOURCE_X = 1
      SCREEN_BLIT_SOURCE_Y = 2
      SCREEN_BLIT_SOURCE_WIDTH = 3
      SCREEN_BLIT_SOURCE_HEIGHT = 4
      SCREEN_BLIT_DESTINATION_X = 5
      SCREEN_BLIT_DESTINATION_Y = 6
      SCREEN_BLIT_DESTINATION_WIDTH = 7
      SCREEN_BLIT_DESTINATION_HEIGHT = 8
      SCREEN_BLIT_GLOBAL_ALPHA = 9
      SCREEN_BLIT_TRANSPARENCY = 10
      SCREEN_BLIT_SCALE_QUALITY = 11
      SCREEN_BLIT_COLOR = 12
};
```

Data:

SCREEN_BLIT_END

Used to terminate the token-value pairs in an attribute list.

SCREEN_BLIT_SOURCE_X

The horizontal position of the rectangle in the source buffer.

The offset is the distance, in pixels, from the left edge of the source buffer. If this attribute is not specified, then a default of `0` will be used.

SCREEN_BLIT_SOURCE_Y

The vertical position of the rectangle in the source buffer.

The offset is the distance, in pixels, from the top edge of the source buffer. If this attribute is not specified, then a default of `0` will be used.

SCREEN_BLIT_SOURCE_WIDTH

The width, in pixels, of the rectangle in the source buffer.

If this attribute is not specified, then the source buffer width will be used. The horizontal and vertical scale factors don't have to be equal. It is acceptable to specify a source width that is larger than the destination width while the source height is smaller than the destination height, and vice versa.

SCREEN_BLIT_SOURCE_HEIGHT

The height, in pixels, of the rectangle in the source buffer.

If this attribute is not specified, then the source buffer height will be used. The horizontal and vertical scale factors don't have to be equal. It is acceptable to specify a source width that is larger than the destination width while the source height is smaller than the destination height, and vice versa.

SCREEN_BLIT_DESTINATION_X

The horizontal position of the rectangle in the destination buffer.

The offset is the distance, in pixels, from the left edge of the destination buffer. If this attribute is not specified, then a default of `0` will be used.

SCREEN_BLIT_DESTINATION_Y

The vertical position of the rectangle in the destination buffer.

The offset is the distance, in pixels, from the top edge of the destination buffer. If this attribute is not specified, then a default of `0` will be used.

SCREEN_BLIT_DESTINATION_WIDTH

The width, in pixels, of the rectangle in the destination buffer.

The width does not have to match the source width. If the destination width is larger, the source rectangle will be stretched. If the destination width is smaller than the source width, the source rectangle will be compressed. If this attribute is not specified, then the destination buffer width will be used.

SCREEN_BLIT_DESTINATION_HEIGHT

The height, in pixels, of the rectangle in the destination buffer.

The height does not have to match the source height. If the destination height is larger, the source rectangle will be stretched. If the destination height is smaller than the source height, the source rectangle will be compressed. If this attribute is not specified, then the destination buffer height will be used.

SCREEN_BLIT_GLOBAL_ALPHA

A global transparency value that is used to blend the source onto the destination.

If this attribute is not specified, then a default of 255 will be used; this default indicates that no global transparency will be applied to the source.

SCREEN_BLIT_TRANSPARENCY

A transparency operation.

The transparency setting defines how the alpha channel, if present, is used to combine the source and destination pixels. The transparency values must be of type *Screen transparency types* (p. 251). If this attribute is not specified, then a default of SCREEN_TRANSPARENCY_NONE will be used.

SCREEN_BLIT_SCALE_QUALITY

A scale quality value.

The scale quality setting defines the type and amount of filtering applied when scaling is required. If the source and destination rectangles are identical in size, the scale quality setting is not used. The scale quality value must be of type *Screen scale quality types* (p. 245). If this attribute is not specified, then a default of SCREEN_QUALITY_NORMAL will be used.

SCREEN_BLIT_COLOR

The color used by the blit operation.

The color format is red bits 16 to 23, green in bits 8 to 15 and blue in bits 0 to 7. If this attribute is not specified, then a default of `#ffffff` (white) will be used.

Library:

`libscreen`

Description:

screen_blit()

Copy pixel data from one buffer to another.

Synopsis:

```
#include <screen/screen.h>

int screen_blit(screen_context_t ctx,
                screen_buffer_t dst,
                screen_buffer_t src,
                const int *attribs)
```

Arguments:

ctx

A connection to Screen

dst

The buffer which data will be copied to.

src

The buffer which the pixels will be copied from.

attribs

A list that contains the attributes that define the blit. This list must consist of a series of token-value pairs terminated with a `SCREEN_BLIT_END` token. The tokens used in this list must be of type *Screen blit types* (p. 255).

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function requests pixels from one buffer be copied to another. The operation is not processed until a flushing execution API function is called, or your application posts changes to one of the context's windows.

The `attribs` argument is allowed to be `NULL` or empty (i.e. contains a single element that is set to `SCREEN_BLIT_END`). If `attribs` is empty, then the following defaults will be applied:

- the source rectangle's vertical and horizontal positions are 0
- the destination rectangle's vertical and horizontal positions are 0
- the source rectangle includes the entire source buffer
- the destination buffer includes the entire destination buffer
- the transparency is `SCREEN_TRANSPARENCY_NONE`
- the global alpha value is `255` (or opaque)
- the scale quality is `SCREEN_QUALITY_NORMAL`.

To change any of this default behavior, set `attribs` with pairings of the following valid tokens and their desired values:

- `SCREEN_BLIT_SOURCE_X`
- `SCREEN_BLIT_SOURCE_Y`
- `SCREEN_BLIT_SOURCE_WIDTH`
- `SCREEN_BLIT_SOURCE_HEIGHT`
- `SCREEN_BLIT_DESTINATION_X`
- `SCREEN_BLIT_DESTINATION_Y`
- `SCREEN_BLIT_DESTINATION_WIDTH`
- `SCREEN_BLIT_DESTINATION_HEIGHT`
- `SCREEN_BLIT_SCALE_QUALITY`
- `SCREEN_BLIT_GLOBAL_ALPHA`
- `SCREEN_BLIT_TRANSPARENCY` (valid transparency values are: `SCREEN_TRANSPARENCY_NONE`, `SCREEN_TRANSPARENCY_TEST`, and `SCREEN_TRANSPARENCY_SOURCE_OVER`)

Returns:

`0` if the blit command was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_fill()

Fill an area of a specified buffer.

Synopsis:

```
#include <screen/screen.h>

int screen_fill(screen_context_t ctx,
                screen_buffer_t dst,
                const int *attribs)
```

Arguments:

ctx

A connection to Screen

dst

The buffer which data will be copied to.

attribs

A list that contains the attributes that define the blit. This list must consist of a series of token-value pairs terminated with a SCREEN_BLIT_END token. The tokens used in this list must be of type *Screen blit types* (p. 255).

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function requests that a rectangular area of the destination buffer be filled with a solid color.

The `attribs` argument is allowed to be NULL or empty (i.e. contains a single element that is set to SCREEN_BLIT_END). If `attribs` is empty, then the following defaults will be applied:

- the destination rectangle's vertical and horizontal positions are 0
- the destination buffer includes the entire destination buffer
- the global alpha value is 255 (or opaque)
- the color is #ffffff (white)

To change any of this default behavior, set `attribs` with pairings of the following valid tokens and their desired values:

- `SCREEN_BLIT_DESTINATION_X`
- `SCREEN_BLIT_DESTINATION_Y`
- `SCREEN_BLIT_DESTINATION_WIDTH`
- `SCREEN_BLIT_DESTINATION_HEIGHT`
- `SCREEN_BLIT_GLOBAL_ALPHA`
- `SCREEN_BLIT_COLOR`

Returns:

`0` if the blit command was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_flush_blits()

Flush all the blits issued.

Synopsis:

```
#include <screen/screen.h>

int screen_flush_blits(screen_context_t ctx,
                       int flags)
```

Arguments:

ctx

A connection to Screen

flags

A flag used by the mutex. Specify `SCREEN_WAIT_IDLE` if the function is required to block until all the blits have been completed.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function flushes all delayed blits and fills since the last call to this function, or since the last call to screen_post_window(). Note that this is a flush of delayed blits

and does not imply a flush of the command buffer. The blits will start executing shortly after you call the function. The blits may not be complete when the function returns, unless the SCREEN_WAIT_IDLE flag is set. This function has no effect on other non-blit delayed calls. The screen_post_window() function performs an implicit flush of any pending blits. The content that is to be presented via the call to screen_post_window() is most likely the result of any pending blit operations completing.

The connection to Screen must have been acquired with the function screen_create_context().

Returns:

`0` if the blit buffer was flushed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

Buffers (screen.h)

A buffer is memory where pixels can be drawn to or read from.

The information and state variables associated with each buffer is stored in memory allocated when the buffer is created with *screen_create_buffer()* Note that memory is allocated to store all information pertaining to the buffer, but not for the buffer itself. When buffers are created by the composited windowing system through calls to *screen_create_window_buffers()* and *screen_create_pixmap_buffer()*, it isn't necessary to create buffer objects with *screen_create_buffer()*. *screen_create_buffer()* is used to create buffers which must be attached to windows or pixmaps.

Usage flags are used when allocating buffers. Depending on the usage, different constraints such as width, height, stride granularity or special alignment must be observed. The usage is also valuable in determining the amount of caching that can be set on a particular buffer.

Depending on which function was called, the buffers can be queried using the *SCREEN_PROPERTY_RENDER_BUFFERS* property with either *screen_get_window_property()* or *screen_get_pixmap_property()* API functions.

Screen buffer properties

Types of properties that are associated with Screen buffer API objects.

Full read/write access to Screen API object properties is system dependent.

These properties are described in full under *Screen property types* (p. 200).

Buffer property	Gettable?	Settable?
SCREEN_PROPERTY_BUFFER_SIZE	Yes	Yes
SCREEN_PROPERTY_FORMAT	Yes	Yes
SCREEN_PROPERTY_INTERLACED	Yes	Yes
SCREEN_PROPERTY_PHYSICALLY_CONTIGUOUS	Yes	Yes
SCREEN_PROPERTY_PLANAR_OFFSETS	Yes	Yes
SCREEN_PROPERTY_POINTER	Yes	Yes
SCREEN_PROPERTY_PROTECTED	Yes	Yes
SCREEN_PROPERTY_SIZE	Yes	Yes
SCREEN_PROPERTY_STRIDE	Yes	Yes
SCREEN_PROPERTY_PHYSICAL_ADDRESS	Yes	Yes

screen_buffer_t

A handle for the screen buffer.

Synopsis:

```
#include <screen/screen.h>

typedef struct _screen_buffer* screen_buffer_t;
```

Library:

`libscreen`

Description:

screen_create_buffer()

Create a buffer handle that can later be attached to a window or a pixmap.

Synopsis:

```
#include <screen/screen.h>

int screen_create_buffer(screen_buffer_t *pbuf)
```

Arguments:

pbuf

An address where the function can store a handle for the native buffer.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function creates a buffer object, which describes memory where pixels can be drawn to or read from. Applications must use screen_destroy_buffer() when a buffer is no longer used.

Returns:

`0` if the buffer was created, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_destroy_buffer()

Destroy a buffer and frees associated resources.

Synopsis:

```
#include <screen/screen.h>

int screen_destroy_buffer(screen_buffer_t buf)
```

Arguments:

buf

The handle of the buffer you want to destroy. This buffer must have been created with screen_create_buffer().

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function destroys the buffer object associated with the buffer handle. Any resources created for this buffer will also be released. The buffer handle can no longer be used as argument in subsequent screen calls. The actual memory buffer described by this buffer handle is not released by this operation. The application is responsible for freeing its own external buffers. Only buffers created with screen_create_buffer() must be destroyed with this function.

Returns:

`0` if the buffer was destroyed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_get_buffer_property_cv()

Retrieve the current value of the specified buffer property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_get_buffer_property_cv(screen_buffer_t buf,
                                  int pname,
```

```
                                                int len,
                                                char *param)
```

Arguments:

buf

The handle of the buffer whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be written to `param`.

param

The buffer where the retrieved value(s) will be stored. This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function stores the current value of a buffer property in a user-provided buffer. No more than `len` bytes of the specified type will be written.

Currently there are no buffer properties which can be retrieved using this function.

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_get_buffer_property_iv()

Retrieve the current value of the specified buffer property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_buffer_property_iv(screen_buffer_t buf,
                                  int pname,
                                  int *param)
```

Arguments:

buf

The handle of the buffer whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function stores the current value of a buffer property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_BUFFER_SIZE`
- `SCREEN_PROPERTY_FORMAT`
- `SCREEN_PROPERTY_INTERLACED`
- `SCREEN_PROPERTY_PHYSICALLY_CONTIGUOUS`
- `SCREEN_PROPERTY_PLANAR_OFFSETS`
- `SCREEN_PROPERTY_PROTECTED`

- SCREEN_PROPERTY_SIZE
- SCREEN_PROPERTY_STRIDE

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_get_buffer_property_llv()

Retrieve the current value of the specified buffer property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_buffer_property_llv(screen_buffer_t buf,
                                   int pname,
                                   long long *param)
```

Arguments:

buf

The handle of the buffer whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function stores the current value of a buffer property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- SCREEN_PROPERTY_PHYSICAL_ADDRESS

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_get_buffer_property_pv()

Retrieve the current value of the specified buffer property of type `void*`.

Synopsis:

```
#include <screen/screen.h>

int screen_get_buffer_property_pv(screen_buffer_t buf,
                                  int pname,
                                  void **param)
```

Arguments:

buf

The handle of the buffer whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function stores the current value of a buffer property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- SCREEN_PROPERTY_EGL_HANDLE

- SCREEN_PROPERTY_POINTER
- SCREEN_PROPERTY_NATIVE_IMAGE

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_buffer_property_cv()

Set the value of the specified buffer property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_set_buffer_property_cv(screen_buffer_t buf,
                                  int pname,
                                  int len,
                                  const char *param)
```

Arguments:

buf

The handle of the buffer whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be read from `param`.

param

A pointer to a buffer containing the new value(s). This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function sets the value of a buffer property from a user-provided buffer. The buffer must have been created with the function screen_create_buffer().

Currently there are no buffer properties which can be set using this function.

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_buffer_property_iv()

Set the value of the specified buffer property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_buffer_property_iv(screen_buffer_t buf,
                                  int pname,
                                  const int *param)
```

Arguments:

buf

The handle of the buffer whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function sets the value of a buffer property from a user-provided buffer. The buffer must have been created with the function screen_create_buffer().

You can use this function to set the value of the following properties:

- SCREEN_PROPERTY_BUFFER_SIZE
- SCREEN_PROPERTY_FORMAT
- SCREEN_PROPERTY_INTERLACED
- SCREEN_PROPERTY_PHYSICALLY_CONTIGUOUS
- SCREEN_PROPERTY_PLANAR_OFFSETS
- SCREEN_PROPERTY_PROTECTED
- SCREEN_PROPERTY_SIZE
- SCREEN_PROPERTY_STRIDE

Returns:

0 if the value(s) of the property was set to new value(s), or -1 if an error occurred (errno is set; refer to /usr/include/errno.h for more details).

screen_set_buffer_property_llv()

Set the value of the specified buffer property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_buffer_property_llv(screen_buffer_t buf,
                                   int pname,
                                   const long long *param)
```

Arguments:

buf

The handle of the buffer whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function sets the value of a buffer property from a user-provided buffer. The buffer must have been created with the function screen_create_buffer().

You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_PHYSICAL_ADDRESS`

Returns:

`0` if the value(s) of the property was set to new value(s), or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_buffer_property_pv()

Set the value of the specified buffer property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_set_buffer_property_pv(screen_buffer_t buf,
                                  int pname,
                                  void **param)
```

Arguments:

buf

The handle of the buffer whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function sets the value of a buffer property from a user-provided buffer. The buffer must have been created with the function screen_create_buffer().

You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_EGL_HANDLE`
- `SCREEN_PROPERTY_POINTER`
- `SCREEN_PROPERTY_NATIVE_IMAGE`

Returns:

`0` if the value(s) of the property was set to new value(s), or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

Contexts (screen.h)

A context defines the relationship with the underlying windowing system.

Once connected to the windowing system, you can use the context to:

- create and control windows
- get and send events
- query and set state variables (properties)

The connection you have to the windowing system through the context remains active until you call *screen_destroy_context()*. Each context has its own event queue, even when several contexts are created in the same process. Permissions are also per context, not per process.

A context can be associated with one or more windows, or with one or more displays.

Screen context properties

Types of properties that are associated with Screen context API objects.

Full read/write access to Screen API object properties is system dependent.

These properties are described in full under *Screen property types* (p. 200)

Context property	Gettable?	Settable?
SCREEN_PROPERTY_DISPLAYS	Yes	No
SCREEN_PROPERTY_IDLE_TIMEOUT	Yes	Yes
SCREEN_PROPERTY_KEYBOARD_FOCUS	Yes	Yes
SCREEN_PROPERTY_MTOUCH_FOCUS	Yes	Yes
SCREEN_PROPERTY_POINTER_FOCUS	Yes	Yes
SCREEN_PROPERTY_DEVICE_COUNT	Yes	No
SCREEN_PROPERTY_DEVICES	Yes	No
SCREEN_PROPERTY_GROUP_COUNT	Yes	No
SCREEN_PROPERTY_GROUPS	Yes	No
SCREEN_PROPERTY_PIXMAP_COUNT	Yes	No
SCREEN_PROPERTY_PIXMAPS	Yes	No
SCREEN_PROPERTY_WINDOW_COUNT	Yes	No
SCREEN_PROPERTY_WINDOWS	Yes	No
SCREEN_PROPERTY_KEYMAP	Yes	Yes

Screen notification types

Types of notifications.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_NOTIFY_VSYNC = 0
      SCREEN_NOTIFY_UPDATE = 1
      SCREEN_NOTIFY_INPUT = 2
      SCREEN_NOTIFY_EVENT = 3
};
```

Data:

SCREEN_NOTIFY_VSYNC

Notification of a vsync.

SCREEN_NOTIFY_UPDATE

Notification of an update.

SCREEN_NOTIFY_INPUT

Notification of an event from an input device.

SCREEN_NOTIFY_EVENT

Notification of an event.

Library:

`libscreen`

Description:

screen_context_t

A handle for the screen context.

Synopsis:

```
#include <screen/screen.h>

typedef struct _screen_context* screen_context_t;
```

Library:

```
libscreen
```

Description:

This handle is used to identify the scope of the relationship with the underlying windowing system. A handle to the screen context is used to:

- create screen API objects
- retrieve and send events

Screen context types

The types of context masks.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_APPLICATION_CONTEXT = 0
      SCREEN_WINDOW_MANAGER_CONTEXT = (1 << 0)
      SCREEN_INPUT_PROVIDER_CONTEXT = (1 << 1)
      SCREEN_POWER_MANAGER_CONTEXT = (1 << 2)
      SCREEN_DISPLAY_MANAGER_CONTEXT = (1 << 3)
};
```

Data:

SCREEN_APPLICATION_CONTEXT

A context type that allows a process to create its own windows and control some of the window properties.

Applications can't modify windows that were created by other applications and can't send events outside their process space. Application contexts aren't aware of other top-level windows in the system; neither are they allowed to operate on them. Application contexts are allowed to parent other windows, even if they are created in other contexts in other processes, and are allowed to control those windows.

SCREEN_WINDOW_MANAGER_CONTEXT

A context type that requests a privileged context to allow a process to modify all windows in the system when new application windows are created or destroyed.

The context also receives notifications when applications create new windows, existing application windows are destroyed, or when an application tries to change certain window properties. A process must have an effective user ID of root to create a context of this type successfully.

SCREEN_INPUT_PROVIDER_CONTEXT

A context type that requests a privileged context to allow a process to send events to any application in the system.

This context type doesn't receive notifications when applications create new windows, when applications destroy existing windows, or when an application attempts to change certain window properties. A process must have an effective user ID of root to create a context of this type successfully.

SCREEN_POWER_MANAGER_CONTEXT

A context type that requests a privileged context to provide access to power management functionality in order to change display power modes.

A process must have an effective user ID of root to create a context of this type successfully.

SCREEN_DISPLAY_MANAGER_CONTEXT

A context type that requests a privileged context to allow a process to modify all display properties in the system.

A process must have an effective user ID of root to create a context of this type successfully.

Library:

`libscreen`

Description:

These bits are intended to be combined in a single integer representing combinations of desired privileges to be applied to a context.

screen_create_context()

Establish a connection with the composited windowing system.

Synopsis:

```
#include <screen/screen.h>

int screen_create_context(screen_context_t *pctx,
                          int flags)
```

Arguments:

pctx

A pointer to a `screen_context_t` where a handle for the new context can be stored.

flags

The type of context to be created. The value must be of type *Screen context types* (p. 277).

Library:

`libscreen`

Description:

Function type: *Immediate Execution* (p. 183)

The screen_create_context() function tries to establish communication with the composited windowing system resource manager (Screen). To do this, the function opens /dev/screen and sends the proper connect sequence. If the call succeeds, memory is allocated to store context state. The composition manager then creates an event queue and associates it with the connecting process.

Returns:

`0` if the context was created, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_destroy_context()

Terminate a connection with the composited windowing system.

Synopsis:

```
#include <screen/screen.h>

int screen_destroy_context(screen_context_t ctx)
```

Arguments:

ctx

The connection to Screen that is to be terminated. This context must have been created with screen_create_context().

Library:

`libscreen`

Description:

Function type: *Apply Execution* (p. 182)

This function closes an existing connection with the composited windowing system resource manager; the context is freed and can no longer be used. All windows and pixmaps associated with this connection will be destroyed. All events waiting in the event queue will be discarded. This operation does not flush the command buffer. Any pending asynchronous commands are discarded.

Returns:

`0` if the context was destroyed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_flush_context()

Flush a context, given a context and a set of flags.

Synopsis:

```
#include <screen/screen.h>

int screen_flush_context(screen_context_t ctx,
                         int flags)
```

Arguments:

ctx

The connection to Screen that is to be flushed. This context must have been created with screen_create_context().

flags

The flag to indicate whether or not to wait until contents of all displays have been updated or to execute immediately.

Library:

`libscreen`

Description:

Function type: *Apply Execution* (p. 182)

This function flushes any delayed command and causes the contents of displays to be updated, when applicable. If `SCREEN_WAIT_IDLE` is specified, the function will not return until the contents of all affected displays have been updated. Passing no flags causes the function to return immediately.

If debugging, you can call this function after all delayed execution function calls as a way to determine the exact function call which may have caused an error.

Returns:

`0` if the context was flushed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_context_property_cv()

Retrieve the current value of the specified context property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_get_context_property_cv(screen_context_t ctx,
                                   int pname,
                                   int len,
                                   char *param)
```

Arguments:

ctx

The handle of the context whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be written to `param`.

param

The buffer where the retrieved value(s) will be stored. This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a context property in a user-provided buffer. No more than `len` bytes of the specified type will be written.

The values of the following properties can be retrieved using this function:

- SCREEN_PROPERTY_KEYMAP

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_context_property_iv()

Retrieve the current value of the specified context property of type integer.

Synopsis:

```
#include <screen/screen.h>
```

```
int screen_get_context_property_iv(screen_context_t ctx,
                                   int pname,
                                   int *param)
```

Arguments:

ctx

The handle of the context whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a context property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_DEVICE_COUNT`
- `SCREEN_PROPERTY_DISPLAY_COUNT`
- `SCREEN_PROPERTY_GROUP_COUNT`
- `SCREEN_PROPERTY_IDLE_STATE`
- `SCREEN_PROPERTY_PIXMAP_COUNT`
- `SCREEN_PROPERTY_WINDOW_COUNT`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_context_property_llv()

Retrieve the current value of the specified context property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_context_property_llv(screen_context_t ctx,
                                    int pname,
                                    long long *param)
```

Arguments:

ctx

The handle of the context whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a context property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_IDLE_TIMEOUT`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_context_property_pv()

Retrieve the current value of the specified context property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_get_context_property_pv(screen_context_t ctx,
                                   int pname,
                                   void **param)
```

Arguments:

ctx

The handle of the context whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a context property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_DEVICES`
- `SCREEN_PROPERTY_DISPLAYS`
- `SCREEN_PROPERTY_GROUPS`
- `SCREEN_PROPERTY_KEYBOARD_FOCUS`
- `SCREEN_PROPERTY_MTOUCH_FOCUS`
- `SCREEN_PROPERTY_POINTER_FOCUS`
- `SCREEN_PROPERTY_PIXMAPS`

- SCREEN_PROPERTY_WINDOWS

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_notify()

Send asynchronous notifications to Screen.

Synopsis:

```
#include <screen/screen.h>

int screen_notify(screen_context_t ctx,
                  int flags,
                  const void *obj,
                  const struct sigevent *event)
```

Arguments:

ctx

The handle of the context of the notification.

flags

The type of notification that you want to be notified of. Valid notification types are of type *Screen notification flag types* (p. 276).

obj

The object within the specified context that the notification is for. For example, this object could be a window, or a display, etc.

event

The notification.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function sends an asynchronous event to Screen. For security reasons, the notification is sent without its associated data.

Returns:

`0` if , or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_context_property_cv()

Set the value of the specified context property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_set_context_property_cv(screen_context_t ctx,
                                   int pname,
                                   int len,
                                   const char *param)
```

Arguments:

ctx

The handle of the context whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be read from `param`.

param

A pointer to a buffer containing the new value(s). This buffer must be of an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a context property from a user-provided buffer. No more than `len` bytes will be read from `param`.

You can use this function to set the value of the following properties:

- SCREEN_PROPERTY_KEYMAP

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_context_property_iv()

Set the value of the specified context property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_context_property_iv(screen_context_t ctx,
                                   int pname,
                                   const int *param)
```

Arguments:

ctx

The handle of the context whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of `type` int. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a context property from a user-provided buffer.

Currently, there are no context properties which can be set using this function.

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_context_property_llv()

Set the value of the specified context property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_context_property_llv(screen_context_t ctx,
                                    int pname,
                                    const long long *param)
```

Arguments:

ctx

The handle of the context whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a context property from a user-provided buffer.

You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_IDLE_TIMEOUT`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_context_property_pv()

Set the value of the specified context property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_set_context_property_pv(screen_context_t ctx,
                                   int pname,
                                   void **param)
```

Arguments:

ctx

The handle of the context whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a context property from a user-provided buffer.

You can use this function to set the value of the following properties:

- SCREEN_PROPERTY_KEYBOARD_FOCUS
- SCREEN_PROPERTY_MTOUCH_FOCUS
- SCREEN_PROPERTY_POINTER_FOCUS

Returns:

0 if the command to set the new property value(s) was queued, or -1 if an error occurred (errno is set; refer to /usr/include/errno.h for more details).

Debugging (screen.h)

Statistics that can be enabled to facilitate debugging your applications.

When debugging an error in your application, it's a good idea to call *screen_flush_context()* after you call any API function which is of the type: delayed execution. Calling *screen_flush_context()* will help you in determining the exact function call that caused the error.

Screen debug graph types

Types of debug graphs.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_DEBUG_GRAPH_FPS = (1 << 0)
      SCREEN_DEBUG_GRAPH_POSTS = (1 << 1)
      SCREEN_DEBUG_GRAPH_BLITS = (1 << 2)
      SCREEN_DEBUG_GRAPH_UPDATES = (1 << 3)
      SCREEN_DEBUG_GRAPH_CPU_TIME = (1 << 4)
      SCREEN_DEBUG_GRAPH_GPU_TIME = (1 << 5)
      SCREEN_DEBUG_STATISTICS = (1 << 7)
};
```

Data:

SCREEN_DEBUG_GRAPH_FPS

Frames per second; the number of posts over time.

SCREEN_DEBUG_GRAPH_POSTS

Pixel count of pixels in dirty rectangles over time.

SCREEN_DEBUG_GRAPH_BLITS

Pixel count of pixels that were in blit requests over time.

SCREEN_DEBUG_GRAPH_UPDATES

Pixel count of pixels used by composition manager in the window to update the framebuffer over time.

SCREEN_DEBUG_GRAPH_CPU_TIME

The time spent on the CPU drawing each frame.

SCREEN_DEBUG_GRAPH_GPU_TIME

The time spent on the GPU drawing each frame.

SCREEN_DEBUG_STATISTICS

Certain staticstics of a window.

The statistics are updated once per second and therefore represent a one second average. The statistics that are displayed are:

- cpu usage, cpu time, gpu time
- private mappings, free memory
- window fps, display fps
- events
- objects
- draws
- triangles
- vertices

Library:

`libscreen`

Description:

All masks except `SCREEN_DEBUG_STATISTICS` are intended to be combined in a single integer bitmask. The bitmask represents combinations of desired debug graphs to be displayed. Only one window can enable debug graphs at a time; the last window to have enabled debug will have its values displayed in the graph. All data but the FPS is normalized to buffer size and refresh rate of display.

Screen packet types

Types of packets.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_REQUEST_PACKET = 0
      SCREEN_BLIT_PACKET = 1
      SCREEN_INPUT_PACKET = 2
      SCREEN_EVENT_PACKET = 3
};
```

Data:

SCREEN_REQUEST_PACKET

A binary chunk from the request ring buffer.

(/dev/screen/request/)

SCREEN_BLIT_PACKET

A binary chunk from the blit ring buffer or log.

(/dev/screen/0/blit#/)

SCREEN_INPUT_PACKET

A binary chunk from the input ring buffer or log.

(/dev/screen/input/)

SCREEN_EVENT_PACKET

A binary chunk from the event queue.

(/dev/screen/pid/)

Library:

`libscreen`

Description:

Screen packet types are for debugging purposes only. It identifies binary chunks that are used only by the screeninfo utility (a command- line tool in /dev/screen/ that is only visible if you have root access) that is used to decode these packets.

screen_print_packet()

Print a screen packet to a specified file.

Synopsis:

```
#include <screen/screen.h>

int screen_print_packet(int type,
                        void *packet,
                        FILE *fd)
```

Arguments:

type

The type of packet to be printed. The packet must be of type Screen_Packet_Types.

packet

The address of the packet to be printed.

fd

The file object where the packet is to be printed to.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function prints out the information relevant to the specified packet to a specified file.

Returns:

`0` if the operation was successful, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

Devices (screen.h)

Devices represent input devices. Input devices can be focused to specific displays.

A valid input device includes:

- keyboard
- mouse
- joystick
- gamepad
- multi-touch

The information and state variables associated with each input device is stored in memory allocated when the device is created with *screen_create_device_type()* You need to be within a privileged context to be able to create input devices. You can create a privileged context by calling the function *screen_create_context()* with a context type of *SCREEN_INPUT_PROVIDER_CONTEXT*. Your process can have an effective user ID of any type to create this context.

Screen device metric count types

Types of metric counts for devices.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_DEVICE_METRIC_EVENT_COUNT = 0
      SCREEN_DEVICE_METRIC_POWER_ON_COUNT = 1
};
```

Data:

SCREEN_DEVICE_METRIC_EVENT_COUNT

The number of input events generated by the device since the last time Screen device metric count types were queried.

SCREEN_DEVICE_METRIC_POWER_ON_COUNT

The number of times that the device has been powered on since the last time Screen device metrics were queried.

Library:

```
libscreen
```

Description:

The metrics are on a per device basis and the counts are reset after being queried. That is, the counts are reset to `0` after you call screen_get_device_property_llv() to retrieve `SCREEN_PROPERTY_METRICS`.

Screen device properties

Types of properties that are associated with Screen device API objects.

Full read/write access to Screen API object properties is system dependent.

These properties are described in full under *Screen property types* (p. 200).

Device property	Gettable?	Settable?
SCREEN_PROPERTY_BUTTONS	Yes	Yes
SCREEN_PROPERTY_DISPLAY	Yes	Yes
SCREEN_PROPERTY_ID_STRING	Yes	Yes
SCREEN_PROPERTY_KEY_MODIFIERS	Yes	No
SCREEN_PROPERTY_TYPE	Yes	No
SCREEN_PROPERTY_USER_HANDLE	Yes	Yes
SCREEN_PROPERTY_WINDOW	Yes	Yes
SCREEN_PROPERTY_POWER_MODE	Yes	Yes
SCREEN_PROPERTY_CONTEXT	Yes	Yes
SCREEN_PROPERTY_KEYMAP_PAGE	Yes	Yes
SCREEN_PROPERTY_KEYMAP	Yes	Yes
SCREEN_PROPERTY_METRIC_COUNT	Yes	No
SCREEN_PROPERTY_BUTTON_COUNT	Yes	Yes
SCREEN_PROPERTY_VENDOR	Yes	Yes
SCREEN_PROPERTY_PRODUCT	Yes	Yes
SCREEN_PROPERTY_ANALOG0	Yes	Yes
SCREEN_PROPERTY_ANALOG1	Yes	Yes
SCREEN_PROPERTY_MAXIMUM_TOUCH_ID	Yes	Yes

Screen game button types

Types of game buttons.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_A_GAME_BUTTON = (1 << 0)
      SCREEN_B_GAME_BUTTON = (1 << 1)
      SCREEN_C_GAME_BUTTON = (1 << 2)
      SCREEN_X_GAME_BUTTON = (1 << 3)
      SCREEN_Y_GAME_BUTTON = (1 << 4)
      SCREEN_Z_GAME_BUTTON = (1 << 5)
      SCREEN_MENU1_GAME_BUTTON = (1 << 6)
      SCREEN_MENU2_GAME_BUTTON = (1 << 7)
      SCREEN_MENU3_GAME_BUTTON = (1 << 8)
      SCREEN_MENU4_GAME_BUTTON = (1 << 9)
      SCREEN_L1_GAME_BUTTON = (1 << 10)
      SCREEN_L2_GAME_BUTTON = (1 << 11)
      SCREEN_L3_GAME_BUTTON = (1 << 12)
      SCREEN_R1_GAME_BUTTON = (1 << 13)
      SCREEN_R2_GAME_BUTTON = (1 << 14)
      SCREEN_R3_GAME_BUTTON = (1 << 15)
      SCREEN_DPAD_UP_GAME_BUTTON = (1 << 16)
      SCREEN_DPAD_DOWN_GAME_BUTTON = (1 << 17)
      SCREEN_DPAD_LEFT_GAME_BUTTON = (1 << 18)
      SCREEN_DPAD_RIGHT_GAME_BUTTON = (1 << 19)
};
```

Data:

SCREEN_A_GAME_BUTTON

SCREEN_B_GAME_BUTTON

SCREEN_C_GAME_BUTTON

SCREEN_X_GAME_BUTTON

SCREEN_Y_GAME_BUTTON

SCREEN_Z_GAME_BUTTON

SCREEN_MENU1_GAME_BUTTON

SCREEN_MENU2_GAME_BUTTON

SCREEN_MENU3_GAME_BUTTON

SCREEN_MENU4_GAME_BUTTON

SCREEN_L1_GAME_BUTTON

SCREEN_L2_GAME_BUTTON

SCREEN_L3_GAME_BUTTON

SCREEN_R1_GAME_BUTTON

SCREEN_R2_GAME_BUTTON

SCREEN_R3_GAME_BUTTON

SCREEN_DPAD_UP_GAME_BUTTON

SCREEN_DPAD_DOWN_GAME_BUTTON

SCREEN_DPAD_LEFT_GAME_BUTTON

SCREEN_DPAD_RIGHT_GAME_BUTTON

Library:

`libscreen`

Description:

These enumerator values are used as constants to map buttons from different controllers to a common game control layout. Typically, you create a structure to represent your game controller and map the buttons to constants in this enumeration.

screen_device_t

A handle for the screen device.

Synopsis:

```
#include <screen/screen.h>

typedef struct _screen_device* screen_device_t;
```

Library:

```
libscreen
```

Description:

screen_create_device_type()

Create a device of specified type to be associated with a context.

Synopsis:

```
#include <screen/screen.h>

int screen_create_device_type(screen_device_t *pdev,
                              screen_context_t ctx,
                              int type)
```

Arguments:

pdev

A pointer to a screen_device_t where a handle for the new input device can be stored.

ctx

The handle of the context in which the input device is to be created. This context must have been created with the context type of `SCREEN_INPUT_PROVIDER_CONTEXT` using screen_create_context().

type

The type of input device to be created.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

The screen_create_device_type() function creates a input device object to be associated with a context. Note that you need to be within a privileged context to call this function. The following are valid input devices which can be created:

- `SCREEN_EVENT_KEYBOARD`
- `SCREEN_EVENT_POINTER`
- `SCREEN_EVENT_JOYSTICK`
- `SCREEN_EVENT_GAMEPAD`
- `SCREEN_EVENT_MTOUCH_TOUCH` Applications must use screen_destroy_device() when a device is no longer used.

Returns:

`0` if the input device was created, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_destroy_device()

Destroy a input device and frees associated resources.

Synopsis:

```
#include <screen/screen.h>

int screen_destroy_device(screen_device_t dev)
```

Arguments:

dev

The handle of the input device that you want to destroy.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function destroys the a input device object associated with the device handle. Any resources created for this input device will be released. Input devices created with screen_create_device_type() must be destroyed with this function.

This function is of type flushing execution because must there be any entries in the command buffer that have reference to this device, the entries will be flushed and processed before destroying the device.

Returns:

`0` if the input device was destroyed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_device_property_cv()

Retrieve the current value of the specified device property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_get_device_property_cv(screen_device_t dev,
                                  int pname,
                                  int len,
                                  char *param)
```

Arguments:

dev

The handle of the device whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be written to `param`.

param

The buffer where the retrieved value(s) will be stored. This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a device property in a user-provided buffer. No more than `len` bytes of the specified type will be written.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_KEYMAP`
- `SCREEN_PROPERTY_ID_STRING`
- `SCREEN_PROPERTY_VENDOR`
- `SCREEN_PROPERTY_PRODUCT`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_device_property_iv()

Retrieve the current value of the specified device property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_device_property_iv(screen_device_t dev,
                                  int pname,
                                  int *param)
```

Arguments:

dev

The handle of the device whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a device property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_BUTTON_COUNT`
- `SCREEN_PROPERTY_BUTTONS`
- `SCREEN_PROPERTY_KEY_MODIFIERS`
- `SCREEN_PROPERTY_KEYMAP_PAGE`
- `SCREEN_PROPERTY_METRIC_COUNT`
- `SCREEN_PROPERTY_POWER_MODE`
- `SCREEN_PROPERTY_TYPE`
- `SCREEN_PROPERTY_ANALOG0`
- `SCREEN_PROPERTY_ANALOG1`
- `SCREEN_PROPERTY_TRANSFORM`
- `SCREEN_PROPERTY_SCALE_FACTOR`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_device_property_llv()

Retrieve the current value of the specified device property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_device_property_llv(screen_device_t dev,
                                   int pname,
                                   long long *param)
```

Arguments:

dev

The handle of the device whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a device property in a user-provided buffer. The values of the following properties can be queried using this function:

- `SCREEN_PROPERTY_METRICS`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_device_property_pv()

Retrieve the current value of the specified device property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_get_device_property_pv(screen_device_t dev,
                                  int pname,
                                  void **param)
```

Arguments:

dev

The handle of the device whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a device property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_CONTEXT`
- `SCREEN_PROPERTY_DISPLAY`
- `SCREEN_PROPERTY_USER_HANDLE`
- `SCREEN_PROPERTY_WINDOW`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_set_device_property_cv()

Set the value of the specified device property of type char.

Synopsis:

```
#include <screen/screen.h>
```

```
int screen_set_device_property_cv(screen_device_t dev,
                                  int pname,
                                  int len,
                                  const char *param)
```

Arguments:

dev

The handle of the device whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be read from `param`.

param

A pointer to a buffer containing the new value(s). This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a device property from a user-provided buffer. No more than `len` bytes will be read from `param`.

You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_ID_STRING`
- `SCREEN_PROPERTY_KEYMAP`
- `SCREEN_PROPERTY_PRODUCT`
- `SCREEN_PROPERTY_VENDOR`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_device_property_iv()

Set the value of the specified device property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_device_property_iv(screen_device_t dev,
                                  int pname,
                                  const int *param)
```

Arguments:

dev

The handle of the device whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a device property from a user-provided buffer.

You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_BUTTON_COUNT`
- `SCREEN_PROPERTY_KEYMAP_PAGE`
- `SCREEN_PROPERTY_POWER_MODE`
- `SCREEN_PROPERTY_TRANSFORM`
- `SCREEN_PROPERTY_SCALE_FACTOR`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_device_property_llv()

Set the value of the specified device property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_device_property_llv(screen_device_t dev,
                                   int pname,
                                   const long long *param)
```

Arguments:

dev

The handle of the device whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a device property from a user-provided buffer.

Currently there are no device properties which can be set using this function.

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_device_property_pv()

Set the value of the specified device property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_set_device_property_pv(screen_device_t dev,
                                  int pname,
                                  void **param)
```

Arguments:

dev

The handle of the device whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a device property from a user-provided buffer.

You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_DISPLAY`
- `SCREEN_PROPERTY_USER_HANDLE`
- `SCREEN_PROPERTY_WINDOW`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

Displays (screen.h)

A display represents the physical display hardware such as a monitor or touch screen display.

You can use display API functions to:

- query and set display properties
- get display modes that are specific to a given hardware display
- perform vsync operations

Note that to have full access to the display properties of the system, you must be working within a privileged context. You create a privileged context by calling the function *screen_create_context()* with a context type of SCREEN_DISPLAY_MANAGER_CONTEXT. Your process must have an effective user ID of root to be able to create this context type. Some API functions will fail to execute if you are not the the correct context.

Screen display metric count types

Types of metric counts for displays.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_DISPLAY_METRIC_ATTACH_COUNT = 0
      SCREEN_DISPLAY_METRIC_POWER_ON_COUNT = 1
      SCREEN_DISPLAY_METRIC_IDLE_COUNT = 2
      SCREEN_DISPLAY_METRIC_EVENT_COUNT = 3
      SCREEN_DISPLAY_METRIC_UPDATE_COUNT = 4
      SCREEN_DISPLAY_METRIC_UPDATE_PIXELS = 5
      SCREEN_DISPLAY_METRIC_UPDATE_READS = 6
      SCREEN_DISPLAY_METRIC_UPDATE_WRITES = 7
};
```

Data:

SCREEN_DISPLAY_METRIC_ATTACH_COUNT

The number of times the display has been attached (connected) since the last time Screen display metrics were queried.

SCREEN_DISPLAY_METRIC_POWER_ON_COUNT

The number of times the display has been powered on since the last time Screen display metrics were queried.

SCREEN_DISPLAY_METRIC_IDLE_COUNT

The number of times the display has been in idle state since the last time Screen display metrics were queried.

SCREEN_DISPLAY_METRIC_EVENT_COUNT

The number of input events that has been focused (or sent) to the display since the last time Screen display metrics were queried.

SCREEN_DISPLAY_METRIC_UPDATE_COUNT

The number of times that the framebuffer of the display has been updated since the last time Screen display metrics were queried.

SCREEN_DISPLAY_METRIC_UPDATE_PIXELS

The number of pixels that the framebuffer of the display has updated since the last time Screen display metrics were queried.

SCREEN_DISPLAY_METRIC_UPDATE_READS

The number of bytes that has been read from the framebuffer of the display since the last time Screen display metrics have been queried.

The number of bytes read is an estimation calculated based on the number of pixels updated by the framebuffer.

SCREEN_DISPLAY_METRIC_UPDATE_WRITES

The number of bytes that has been written to the framebuffer of the display since the last time Screen display metrics have been queried.

The number of bytes written is an estimation calculated based on the number of pixels updated by the framebuffer.

Library:

`libscreen`

Description:

The metrics are on a per display basis and the counts are reset after being queried. That is, the counts are reset to `0` after you call `screen` get_display_property_llv() to retrieve `SCREEN_PROPERTY_METRICS`.

Screen mode types

The screen mode types.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_MODE_PREFERRED = 0x1
};
```

Data:

SCREEN_MODE_PREFERRED

Used in the flags of the type screen_display_mode_t to indicate that this mode is the preferred mode.

Library:

`libscreen`

Description:

Screen display properties

Types of properties that are associated with Screen display API objects.

Full read/write access to Screen API object properties is system dependent.

These properties are described in full under *Screen property types* (p. 200).

Display property	Configurable?	Gettable?	Settable?
SCREEN_PROPERTY_GAMMA	Yes	Yes	Yes
SCREEN_PROPERTY_ID_STRING	No	Yes	No
SCREEN_PROPERTY_KEY_MODIFIERS	No	Yes	No
SCREEN_PROPERTY_ROTATION	No	Yes	No
SCREEN_PROPERTY_SIZE	No	Yes	Yes
SCREEN_PROPERTY_TRANSPARENCY	No	Yes	No
SCREEN_PROPERTY_TYPE	No	Yes	No
SCREEN_PROPERTY_MIRROR_MODE	No	Yes	Yes
SCREEN_PROPERTY_ATTACHED	No	Yes	No

Display property	Configurable?	Gettable?	Settable?
SCREEN_PROPERTY_DETACHABLE	No	Yes	No
SCREEN_PROPERTY_NATIVE_RESOLUTION	No	Yes	No
SCREEN_PROPERTY_PROTECTION_ENABLE	No	Yes	No
SCREEN_PROPERTY_PHYSICAL_SIZE	No	Yes	No
SCREEN_PROPERTY_FORMAT_COUNT	No	Yes	No
SCREEN_PROPERTY_FORMATS	No	Yes	No
SCREEN_PROPERTY_VIEWPORT_POSITION	No	Yes	No
SCREEN_PROPERTY_VIEWPORT_SIZE	No	Yes	No
SCREEN_PROPERTY_IDLE_STATE	No	Yes	No
SCREEN_PROPERTY_KEEP_AWAKES	No	Yes	No
SCREEN_PROPERTY_IDLE_TIMEOUT	No	Yes	Yes
SCREEN_PROPERTY_KEYBOARD_FOCUS	No	Yes	Yes
SCREEN_PROPERTY_MTOUCH_FOCUS	No	Yes	Yes
SCREEN_PROPERTY_POINTER_FOCUS	No	Yes	Yes
SCREEN_PROPERTY_ID	No	Yes	Yes
SCREEN_PROPERTY_POWER_MODE	No	Yes	Yes
SCREEN_PROPERTY_MODE_COUNT	No	Yes	No
SCREEN_PROPERTY_MODE	No	Yes	Yes
SCREEN_PROPERTY_CONTEXT	No	Yes	Yes
SCREEN_PROPERTY_DPI	No	Yes	Yes
SCREEN_PROPERTY_METRIC_COUNT	No	Yes	No
SCREEN_PROPERTY_METRICS	No	Yes	No
SCREEN_PROPERTY_BUTTON_COUNT	No	Yes	Yes
SCREEN_PROPERTY_VENDOR	No	Yes	Yes
SCREEN_PROPERTY_PRODUCT	No	Yes	Yes
SCREEN_PROPERTY_TECHNOLOGY	No	Yes	No

Screen display technology types

Types of technologies for a display.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_DISPLAY_TECHNOLOGY_UNKNOWN = 0
      SCREEN_DISPLAY_TECHNOLOGY_CRT = 1
      SCREEN_DISPLAY_TECHNOLOGY_LCD = 2
      SCREEN_DISPLAY_TECHNOLOGY_PLASMA = 3
      SCREEN_DISPLAY_TECHNOLOGY_LED = 4
      SCREEN_DISPLAY_TECHNOLOGY_OLED = 5
};
```

Data:

SCREEN_DISPLAY_TECHNOLOGY_UNKNOWN

Any other display technology that isn't enumerated.

SCREEN_DISPLAY_TECHNOLOGY_CRT

All monochrome and standard tricolor CRTs.

SCREEN_DISPLAY_TECHNOLOGY_LCD

All active and passive matrix LCDs.

SCREEN_DISPLAY_TECHNOLOGY_PLASMA

All PDP types including DC and AC plasma displays.

SCREEN_DISPLAY_TECHNOLOGY_LED

Inorganic LED.

SCREEN_DISPLAY_TECHNOLOGY_OLED

Organic LED/OEL.

Library:

`libscreen`

Description:

Screen display types

Types of connections to a display.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_DISPLAY_TYPE_INTERNAL = 0x7660
      SCREEN_DISPLAY_TYPE_COMPOSITE = 0x7661
      SCREEN_DISPLAY_TYPE_SVIDEO = 0x7662
      SCREEN_DISPLAY_TYPE_COMPONENT_YPbPr = 0x7663
      SCREEN_DISPLAY_TYPE_COMPONENT_RGB = 0x7664
      SCREEN_DISPLAY_TYPE_COMPONENT_RGBHV = 0x7665
      SCREEN_DISPLAY_TYPE_DVI = 0x7666
      SCREEN_DISPLAY_TYPE_HDMI = 0x7667
      SCREEN_DISPLAY_TYPE_DISPLAYPORT = 0x7668
      SCREEN_DISPLAY_TYPE_OTHER = 0x7669
};
```

Data:

SCREEN_DISPLAY_TYPE_INTERNAL

An internal connection type to the display.

SCREEN_DISPLAY_TYPE_COMPOSITE

A composite connection type to the display.

SCREEN_DISPLAY_TYPE_SVIDEO

A S-Video connection type to the display.

SCREEN_DISPLAY_TYPE_COMPONENT_YPbPr

A component connection type to the display - specifically the YPbPr signal of the component connection.

SCREEN_DISPLAY_TYPE_COMPONENT_RGB

A component connection type to the display - specifically the RGB signal of the component connection.

SCREEN_DISPLAY_TYPE_COMPONENT_RGBHV

A component connection type to the display - specifically the RBGHV signal of the component connection.

SCREEN_DISPLAY_TYPE_DVI

A DVI connection type to the display.

SCREEN_DISPLAY_TYPE_HDMI

A HDMI connection type to the display.

SCREEN_DISPLAY_TYPE_DISPLAYPORT

A DisplayPort connection type to the display.

SCREEN_DISPLAY_TYPE_OTHER

A connection type to the display which is one other than internal, composite, S-Video, component, DVI, HDMI, or DisplayPort.

Library:

```
libscreen
```

Description:

screen_display_mode_t

A structure to contain values related to Screen display mode.

Synopsis:

```
#include <screen/screen.h>

typedef struct _screen_mode  screen_display_mode_t;
```

Library:

```
libscreen
```

Description:

screen_display_t

A handle for the screen display.

Synopsis:

```
#include <screen/screen.h>

typedef struct _screen_display* screen_display_t;
```

Library:

libscreen

Description:

screen_get_display_modes()

Retrieve the display modes supported by a specified display.

Synopsis:

```
#include <screen/screen.h>

int screen_get_display_modes(screen_display_t display,
                             int max,
                             screen_display_mode_t *param)
```

Arguments:

display

The handle of the display whose display modes are being queried.

max

The maximum number of display modes that can be written to the array of modes pointed to by `param`.

param

The buffer where the retrieved display modes will be stored.

Library:

libscreen

Description:

Function Type: *Flushing Execution* (p. 183)

This function returns the video modes that are supported by a display. All elements in the list are unique. Note that several modes can have identical resolutions and differ only in refresh rate or aspect ratio. You can obtain the number of modes supported by querying the SCREEN_PROPERTY_MODE_COUNT property. No more than `max` modes will be stored.

Returns:

`0` if a query was successful and the display mode is stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_display_property_cv()

Retrieve the current value of the specified display property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_get_display_property_cv(screen_display_t disp,
                                   int pname,
                                   int len,
                                   char *param)
```

Arguments:

disp

The handle of the display whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be written to `param`.

param

The buffer where the retrieved value(s) will be stored. This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a display property in a user-provided buffer. No more than `len` bytes of the specified type will be written.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_ID_STRING`
- `SCREEN_PROPERTY_VENDOR`
- `SCREEN_PROPERTY_PRODUCT`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_display_property_iv()

Retrieve the current value of the specified display property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_display_property_iv(screen_display_t disp,
                                   int pname,
                                   int *param)
```

Arguments:

disp

The handle of the display whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a display property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_ID`
- `SCREEN_PROPERTY_ATTACHED`
- `SCREEN_PROPERTY_DETACHABLE`
- `SCREEN_PROPERTY_FORMAT_COUNT`
- `SCREEN_PROPERTY_GAMMA`
- `SCREEN_PROPERTY_IDLE_STATE`
- `SCREEN_PROPERTY_KEEP_AWAKES`
- `SCREEN_PROPERTY_KEY_MODIFIERS`
- `SCREEN_PROPERTY_MIRROR_MODE`
- `SCREEN_PROPERTY_MODE_COUNT`
- `SCREEN_PROPERTY_POWER_MODE`
- `SCREEN_PROPERTY_PROTECTION_ENABLE`
- `SCREEN_PROPERTY_ROTATION`
- `SCREEN_PROPERTY_TRANSPARENCY`
- `SCREEN_PROPERTY_TYPE`
- `SCREEN_PROPERTY_DPI`
- `SCREEN_PROPERTY_NATIVE_RESOLUTION`
- `SCREEN_PROPERTY_PHYSICAL_SIZE`
- `SCREEN_PROPERTY_SIZE`
- `SCREEN_PROPERTY_FORMATS`
- `SCREEN_PROPERTY_VIEWPORT_POSITION`
- `SCREEN_PROPERTY_VIEWPORT_SIZE`
- `SCREEN_PROPERTY_METRIC_COUNT`
- `SCREEN_PROPERTY_TECHNOLOGY`
- `SCREEN_PROPERTY_REFERENCE_COLOR`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_display_property_llv()

Retrieve the current value of the specified display property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_display_property_llv(screen_display_t disp,
                                    int pname,
                                    long long *param)
```

Arguments:

disp

The handle of the device whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a display property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_IDLE_TIMEOUT`
- `SCREEN_PROPERTY_METRICS`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more

details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_display_property_pv()

Retrieve the current value of the specified display property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_get_display_property_pv(screen_display_t disp,
                                   int pname,
                                   void **param)
```

Arguments:

disp

The handle of the display whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of a display property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_CONTEXT`
- `SCREEN_PROPERTY_MODE`
- `SCREEN_PROPERTY_KEYBOARD_FOCUS`
- `SCREEN_PROPERTY_MTOUCH_FOCUS`
- `SCREEN_PROPERTY_POINTER_FOCUS`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_read_display()

Take a screenshot of the display and store the resulting image in the specified buffer.

Synopsis:

```
#include <screen/screen.h>

int screen_read_display(screen_display_t disp,
                        screen_buffer_t buf,
                        int count,
                        const int *read_rects,
                        int flags)
```

Arguments:

disp

The handle of the display that is the target of the screenshot.

buf

The buffer where the resulting image will be stored.

count

The number of rectables supplied in the `read_rects` argument.

read_rects

A pointer to (`count` * 4) integers that define the areas of display that need to be grabbed for the screenshot.

flags

The mutex flags; must be set to 0.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function takes a screenshot of a display and stores the result in a user-provided buffer. The buffer can be a pixmap buffer or a window buffer. The buffer must have been created with the usage flag SCREEN_USAGE_NATIVE in order for the operation to succeed. You need to be working within a privileged context so that you have full access to the display properties of the system. Therefore, a context which was created with the type SCREEN_DISPLAY_MANAGER_CONTEXT must be used. When capturing screenshots of multiple displays, you will need to make one screen_read_display() function call per display. The call blocks until the operation is completed. If count is 0 and read_rects is NULL, the entire display is grabbed. Otherwise, read_rects must point to count * 4 integers defining rectangles in screen coordinates that need to be grabbed. Note that the buffer size does not have to match the display size. Scaling will be applied to make the screenshot fit into the buffer provided.

Returns:

`0` if a the operation was successful and the pixels are written to `buf`, or `-1` of an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_display_property_cv()

Set the value of the specified display property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_set_display_property_cv(screen_display_t disp,
                                   int pname,
                                   int len,
                                   const char *param)
```

Arguments:

disp

The handle of the display whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be read from `param`.

param

A pointer to a buffer containing the new value(s). This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a display property from a user-provided buffer. No more than `len` bytes will be read from `param`.

Currently there are no display properties that can be queried using this function.

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_display_property_iv()

Set the value of the specified display property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_display_property_iv(screen_display_t disp,
                                   int pname,
                                   const int *param)
```

Arguments:

disp

The handle of the display whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a display property from a user-provided buffer.

You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_GAMMA`
- `SCREEN_PROPERTY_MIRROR_MODE`
- `SCREEN_PROPERTY_MODE`
- `SCREEN_PROPERTY_POWER_MODE`
- `SCREEN_PROPERTY_PROTECTION_ENABLE`
- `SCREEN_PROPERTY_ROTATION`
- `SCREEN_PROPERTY_VIEWPORT_POSITION`
- `SCREEN_PROPERTY_VIEWPORT_SIZE`
- `SCREEN_PROPERTY_REFERENCE_COLOR`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_display_property_llv()

Set the value of the specified display property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_display_property_llv(screen_display_t disp,
                                    int pname,
                                    const long long *param)
```

Arguments:

disp

The handle of the display whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a display property from a user-provided buffer.

You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_IDLE_TIMEOUT`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_display_property_pv()

Set the value of the specified display property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_set_display_property_pv(screen_display_t disp,
                                   int pname,
                                   void **param)
```

Arguments:

disp

The handle of the display whose property is to be set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a display property from a user-provided buffer.

You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_KEYBOARD_FOCUS`
- `SCREEN_PROPERTY_MTOUCH_FOCUS`
- `SCREEN_PROPERTY_POINTER_FOCUS`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_share_display_buffers()

Cause a window to share its buffers with a display.

Synopsis:

```
#include <screen/screen.h>

int screen_share_display_buffers(screen_window_t win,
                                 screen_display_t share,
                                 int count)
```

Arguments:

win

The handle of the window who will be sharing its buffer(s).

share

The handle of the display who is sharing buffer(s).

count

The number of buffer st that is shared by the window to the display. A value of `0` will default to the Screen services to select the appropriate values for properties such as SCREEN_PROPERTY_FORMAT, SCREEN_PROPERTY_USAGE and SCREEN_PROPERTY_BUFFER_SIZE.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function creates a `count` number of buffers with the size defined by the SCREEN_PROPERTY_BUFFER_SIZE window property of `win`. These buffers are rendered by the windowing system. The display will be used to generate content for the (window) buffers. Once there is a post for the window `win`, the content of the buffers will be displayed on the display `share`. To share display buffers, you need to be working within a privileged context. Therefore, a context that was created with the type SCREEN_DISPLAY_MANAGER_CONTEXT must be used.

If the display has a framebuffer, then screen_share_display_buffers() is similar to screen_share_window_buffers().

Returns:

`0` if the window shared its buffers, or `-1` of an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_wait_vsync()

Block the calling thread until the next vsync happens on the specified display.

Synopsis:

```
#include <screen/screen.h>
```

```
int screen_wait_vsync(screen_display_t display)
```

Arguments:

display

An instance of the display on which to perform the the vsync operation.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function blocks the calling thread and returns when the next vsync operation occurs on the specified display.

Returns:

`0` if a vsync operation occurred, or `-1` of an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

Events (screen.h)

Events are associated with a given context.

The windowing system manages an event queue per context. An event can be transferred from the event queue to the application by using *screen_get_event()*. Once transferred to the application, it can be used with *screen_get_event_property()* in an event handling routine to handle the event accordingly.

Several variations of *screen_get_event_property()* and *screen_set_event_property()* functions are available to query and set event properties respectively. It is important to note that different types of events will permit a different selection of properties that can be queried or set. The exception to this is the property which indicates the type of event (*SCREEN_PROPERTY_TYPE*). Therefore if you have an event to handle, it is common practice in your event handling routine to first query the type of the event using *screen_get_event_property_iv()* with *SCREEN_PROPERTY_TYPE* as the name of the property. Once you know which type of event you are handling, you can proceed to call the appropriate query and set API functions for the other event properties. Refer to the example below for the logic of a simple event handling routine.

```
...
screen_event_t screen_ev;
screen_create_event(&screen_ev);

 while (1){
  do {
   /* Call screen_get_event with a timeout of -1, or ~0, so that you block
      until an event is put into the event queue. */
   screen_get_event(screen_ctx, screen_ev, vis ? 0 : ~0);

   /* Get the type of the event */
   screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &type);

            /* Handle events of interest to your application */
   if (type == SCREEN_EVENT_POST) {
    /* Handle SCREEN_EVENT_POST event accordingly;
       query or set properties valid for POST events */
   }
   else if (type == SCREEN_EVENT_CLOSE) {
    /* Handle SCREEN_EVENT_CLOSE event accordingly;
       query or set properties valid for CLOSE events */
   }
   else if
   ...
  } while (type != SCREEN_EVENT_NONE);
 }
...
```

Screen event properties

Types of properties that are associated with Screen event API objects.

Full read/write access to Screen API object properties is system dependent.

These properties are described in full under *Screen property types* (p. 200)

Event property	Gettable?	Settable?
SCREEN_PROPERTY_BUTTONS	Yes	Yes

Event property	Gettable?	Settable?
SCREEN_PROPERTY_DEVICE	Yes	Yes
SCREEN_PROPERTY_DISPLAY	Yes	Yes
SCREEN_PROPERTY_GROUP	Yes	Yes
SCREEN_PROPERTY_INPUT_VALUE	Yes	Yes
SCREEN_PROPERTY_JOG_COUNT	Yes	Yes
SCREEN_PROPERTY_KEY_CAP	Yes	Yes
SCREEN_PROPERTY_KEY_MODIFIERS	Yes	Yes
SCREEN_PROPERTY_KEY_SCAN	Yes	Yes
SCREEN_PROPERTY_KEY_SYM	Yes	Yes
SCREEN_PROPERTY_NAME	Yes	Yes
SCREEN_PROPERTY_POSITION	Yes	Yes
SCREEN_PROPERTY_SIZE	Yes	Yes
SCREEN_PROPERTY_SOURCE_POSITION	Yes	Yes
SCREEN_PROPERTY_SOURCE_SIZE	Yes	Yes
SCREEN_PROPERTY_TYPE	Yes	No
SCREEN_PROPERTY_USER_DATA	Yes	Yes
SCREEN_PROPERTY_WINDOW	Yes	Yes
SCREEN_PROPERTY_MIRROR_MODE	Yes	Yes
SCREEN_PROPERTY_EFFECT	Yes	Yes
SCREEN_PROPERTY_ATTACHED	Yes	Yes
SCREEN_PROPERTY_PROTECTION_ENABLE	Yes	Yes
SCREEN_PROPERTY_TOUCH_ID	Yes	Yes
SCREEN_PROPERTY_TOUCH_ORIENTATION	Yes	Yes
SCREEN_PROPERTY_TOUCH_PRESSURE	Yes	Yes
SCREEN_PROPERTY_TIMESTAMP	Yes	Yes
SCREEN_PROPERTY_SEQUENCE_ID	Yes	Yes
SCREEN_PROPERTY_IDLE_STATE	Yes	Yes
SCREEN_PROPERTY_MODE	Yes	Yes
SCREEN_PROPERTY_MOUSE_WHEEL	Yes	Yes

Event property	Gettable?	Settable?
SCREEN_PROPERTY_CONTEXT	Yes	Yes
SCREEN_PROPERTY_OBJECT_TYPE	Yes	No
SCREEN_PROPERTY_MOUSE_HORIZONTAL_WHEEL	Yes	Yes
SCREEN_PROPERTY_TOUCH_TYPE	Yes	Yes
SCREEN_PROPERTY_SCALE_FACTOR	Yes	Yes
SCREEN_PROPERTY_ANALOG0	Yes	Yes
SCREEN_PROPERTY_ANALOG1	Yes	Yes
SCREEN_PROPERTY_KEY_ALTERNATE_SYM	Yes	Yes

Screen event types

Types of events.

Synopsis:

```
#include <screen/screen.h>

 enum {
      SCREEN_EVENT_NONE = 0
      SCREEN_EVENT_CREATE = 1
      SCREEN_EVENT_PROPERTY = 2
      SCREEN_EVENT_CLOSE = 3
      SCREEN_EVENT_INPUT = 4
      SCREEN_EVENT_JOG = 5
      SCREEN_EVENT_POINTER = 6
      SCREEN_EVENT_KEYBOARD = 7
      SCREEN_EVENT_USER = 8
      SCREEN_EVENT_POST = 9
      SCREEN_EVENT_EFFECT_COMPLETE = 10
      SCREEN_EVENT_DISPLAY = 11
      SCREEN_EVENT_IDLE = 12
      SCREEN_EVENT_UNREALIZE = 13
      SCREEN_EVENT_GAMEPAD = 14
      SCREEN_EVENT_JOYSTICK = 15
      SCREEN_EVENT_DEVICE = 16
      SCREEN_EVENT_MTOUCH_TOUCH = 100
      SCREEN_EVENT_MTOUCH_MOVE = 101
      SCREEN_EVENT_MTOUCH_RELEASE = 102
};
```

Data:

SCREEN_EVENT_NONE

A blocking event indicating that there are currently no events in the queue.

SCREEN_EVENT_CREATE

Dispatched when a child window is created.

SCREEN_EVENT_PROPERTY

Dispatched when a property is set.

SCREEN_EVENT_CLOSE

Dispatched when a child window is destroyed.

SCREEN_EVENT_INPUT

Dispatched when an unknown input event occurs.

SCREEN_EVENT_JOG

Dispatched when a jog dial input event occurs.

SCREEN_EVENT_POINTER

Used to describe either a device or event API object:

- Device: represents a valid input device type; used for a device object's SCREEN_PROPERTY_TYPE
- Event: dispatched when a pointer input event occurs

SCREEN_EVENT_KEYBOARD

Used to describe either a device or event API object:

- Device: represents a valid input device type; used for a device object's SCREEN_PROPERTY_TYPE
- Event: dispatched when a keyboard input event occurs

SCREEN_EVENT_USER

Dispatched when a user event is detected.

SCREEN_EVENT_POST

Dispatched when a child window has posted its first frame.

SCREEN_EVENT_EFFECT_COMPLETE

Dispatched to the window manager indicating that a rotation effect has completed.

SCREEN_EVENT_DISPLAY

Dispatched when an external display is detected.

SCREEN_EVENT_IDLE

Dispatched when the window enters an idle state.

SCREEN_EVENT_UNREALIZE

Dispatched when a handle to a window is lost.

SCREEN_EVENT_GAMEPAD

Used to describe either a device or event API object:

- Device: represents a valid input device type; used for a device object's SCREEN_PROPERTY_TYPE
- Event: dispatched when a gamepad input event occurs

SCREEN_EVENT_JOYSTICK

Used to describe either a device or event API object:

- Device: represents a valid input device type; used for a device object's SCREEN_PROPERTY_TYPE
- Event: dispatched when a joystick input event occurs

SCREEN_EVENT_DEVICE

Dispatched when an input device is detected.

SCREEN_EVENT_MTOUCH_TOUCH

Used to describe either a device or event API object:

- Device: represents a valid input device type; used for a device object's SCREEN_PROPERTY_TYPE
- Event: dispatched when a multi-touch event is detected

SCREEN_EVENT_MTOUCH_MOVE

Dispatched when a multi-touch move event is detected.

For example, when the user moves his or her fingers to make an input gesture.

SCREEN_EVENT_MTOUCH_RELEASE

Dispatched when a multi-touch release event occurs, or when the user completes the multi-touch gesture.

Library:

`libscreen`

Description:

screen_create_event()

Create an event that can later be filled with event data.

Synopsis:

```
#include <screen/screen.h>

int screen_create_event(screen_event_t *pev)
```

Arguments:

pev

An address where the function can store a handle to the native event.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function creates an event object. This event can be used to store events from the process's event queue using screen_get_event(). Event data can also be filled in with the `screen_set_event_property()` functions and sent to other applications using screen_inject_event() or screen_send_event(). Events are opaque handles. `screen_get_event_property()` functions must be used to get information from the event. You must destroy event objects when you no longer need them by using screen_destroy_event().

Returns:

`0` if a new event was created, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_destroy_event()

Destroy an event and free associated memory.

Synopsis:

```
#include <screen/screen.h>

int screen_destroy_event(screen_event_t ev)
```

Arguments:

ev

The handle of the event to destroy. This event must have been created with screen_create_event().

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function frees the memory allocated to hold an event. The event can no longer be used as an argument in subsequent screen calls.

Returns:

`0` if the event was destroyed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_event_t

A handle for the screen event.

Synopsis:

```
#include <screen/screen.h>

typedef struct _screen_event* screen_event_t;
```

Library:

libscreen

Description:

screen_get_event()

Wait for a screen event.

Synopsis:

```
#include <screen/screen.h>

int screen_get_event(screen_context_t ctx,
                     screen_event_t ev,
                     uint64_t timeout)
```

Arguments:

ctx

The context to retrieve events from. This context must have been created using screen_create_context().

ev

An event previously allocated with screen_create_event(). Any contents in this event will be replaced with the next event.

timeout

The maximum time to wait for an event to occur if one has not been queued up already. `0` indicates that the call must not wait at all if there are no events associated with the specified context. `-1` indicates that the call must not return until an event is ready.

Library:

libscreen

Description:

Function Type: *Immediate Execution* (p. 183)

This function gets the next event associated with the given context. If no events have been queued, the function will wait up to the specified amount of time for an event

to occur. If the function times out before an event becomes available, an event with a SCREEN_EVENT_NONE type will be returned.

Returns:

`0` if the event was retrieved, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_get_event_property_cv()

Retrieve the current value of the specified event property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_get_event_property_cv(screen_event_t ev,
                                 int pname,
                                 int len,
                                 char *param)
```

Arguments:

ev

The handle of the event whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be written to `param`.

param

The buffer where the retrieved value(s) will be stored. This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function stores the current value of an event property in a user-provided buffer. No more than `len` bytes of the specified type will be written. The list of properties that can be queried per event type are listed as follows:

- `SCREEN_EVENT_CREATE`
- `SCREEN_PROPERTY_GROUP`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_get_event_property_iv()

Retrieve the current value of the specified event property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_event_property_iv(screen_event_t ev,
                                 int pname,
                                 int *param)
```

Arguments:

ev

The handle of the event whose property is being queried. The event must have an event type of *Screen event types* (p. 334).

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function stores the current value of an event property in a user-provided buffer. The list of properties that can be queried per event type are listed as follows:

Event Type: Any

- `SCREEN_PROPERTY_TYPE`
- `SCREEN_PROPERTY_SCALE_FACTOR`

Event Type: `SCREEN_EVENT_DISPLAY`

- `SCREEN_PROPERTY_ATTACHED`
- `SCREEN_PROPERTY_MIRROR_MODE`
- `SCREEN_PROPERTY_MODE`
- `SCREEN_PROPERTY_PROTECTION_ENABLE`

Event Type: `SCREEN_EVENT_EFFECT_COMPLETE`

- `SCREEN_PROPERTY_EFFECT`

Event Type: `SCREEN_EVENT_GAMEPAD`

- `SCREEN_PROPERTY_BUTTONS`
- `SCREEN_PROPERTY_DEVICE`
- `SCREEN_PROPERTY_ANALOG0`
- `SCREEN_PROPERTY_ANALOG1`
- `SCREEN_PROPERTY_KEY_MODIFIERS`

Event Type: `SCREEN_EVENT_IDLE`

- `SCREEN_PROPERTY_IDLE_STATE`
- `SCREEN_PROPERTY_OBJECT_TYPE`

Event Type: `SCREEN_EVENT_INPUT`

- `SCREEN_PROPERTY_DEVICE`
- `SCREEN_PROPERTY_INPUT_VALUE`

Event Type: `SCREEN_EVENT_JOG`

- `SCREEN_PROPERTY_DEVICE`
- `SCREEN_PROPERTY_JOG_COUNT`

Event Type: `SCREEN_EVENT_JOYSTICK`

- `SCREEN_PROPERTY_BUTTONS`

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_ANALOG0
- SCREEN_PROPERTY_KEY_MODIFIERS

Event Type: SCREEN_EVENT_KEYBOARD

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_KEY_CAP
- SCREEN_PROPERTY_KEY_FLAGS
- SCREEN_PROPERTY_KEY_MODIFIERS
- SCREEN_PROPERTY_KEY_SCAN
- SCREEN_PROPERTY_KEY_SYM
- SCREEN_PROPERTY_SEQUENCE_ID

Event Types: SCREEN_EVENT_MTOUCH_TOUCH, SCREEN_EVENT_MTOUCH_MOVE, SCREEN_EVENT_MTOUCH_RELEASE

- SCREEN_PROPERTY_BUTTONS
- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_KEY_MODIFIERS
- SCREEN_PROPERTY_POSITION
- SCREEN_PROPERTY_SEQUENCE_ID
- SCREEN_PROPERTY_SIZE
- SCREEN_PROPERTY_SOURCE_POSITION
- SCREEN_PROPERTY_SOURCE_SIZE
- SCREEN_PROPERTY_TOUCH_ID
- SCREEN_PROPERTY_TOUCH_ORIENTATION
- SCREEN_PROPERTY_TOUCH_PRESSURE
- SCREEN_PROPERTY_TOUCH_TYPE

Event Type: SCREEN_EVENT_POINTER

- SCREEN_PROPERTY_BUTTONS
- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_KEY_MODIFIERS
- SCREEN_PROPERTY_MOUSE_HORIZONTAL_WHEEL
- SCREEN_PROPERTY_MOUSE_WHEEL
- SCREEN_PROPERTY_POSITION
- SCREEN_PROPERTY_SOURCE_POSITION

Event Type: SCREEN_EVENT_PROPERTY

- SCREEN_PROPERTY_NAME
- SCREEN_PROPERTY_OBJECT_TYPE

Event Type: SCREEN_EVENT_USER

- SCREEN_PROPERTY_USER_DATA

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_get_event_property_llv()

Retrieve the current value of the specified event property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_event_property_llv(screen_event_t ev,
                                  int pname,
                                  long long *param)
```

Arguments:

ev

The handle of the event whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of long long integer.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function stores the current value of an event property in a user-provided buffer.

Event Type: Any

- SCREEN_PROPERTY_TIMESTAMP

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_get_event_property_pv()

Retrieve the current value of the specified event property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_get_event_property_pv(screen_event_t ev,
                                 int pname,
                                 void **param)
```

Arguments:

ev

The handle of the event whose property is being queried. The event must have an event type of *Screen event types* (p. 334).

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function stores the current value of an event property in a user-provided buffer. The list of properties that can be queried per event type are listed as follows:

Event Type: SCREEN_EVENT_CLOSE

- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_CREATE

- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_DISPLAY

- SCREEN_PROPERTY_DISPLAY

Event Type: SCREEN_EVENT_GAMEPAD

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_IDLE

- SCREEN_PROPERTY_DISPLAY
- SCREEN_PROPERTY_GROUP

Event Type: SCREEN_EVENT_INPUT

- SCREEN_PROPERTY_DEVICE

Event Type: SCREEN_EVENT_JOG

- SCREEN_PROPERTY_DEVICE

Event Type: SCREEN_EVENT_JOYSTICK

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_KEYBOARD

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_WINDOW

Event Types: SCREEN_EVENT_MTOUCH_TOUCH, SCREEN_EVENT_MTOUCH_MOVE, SCREEN_EVENT_MTOUCH_RELEASE

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_POINTER

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_POST

- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_PROPERTY

- SCREEN_PROPERTY_GROUP
- SCREEN_PROPERTY_DISPLAY
- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_UNREALIZE

- SCREEN_PROPERTY_WINDOW

Returns:

0 if a query was successful and the value(s) of the property are stored in param, or -1 if an error occurred (errno is set; refer to /usr/include/errno.h for more details).

screen_inject_event()

Send an input event to the window that has input focus on a given display.

Synopsis:

```
#include <screen/screen.h>

int screen_inject_event(screen_display_t disp,
                        screen_event_t ev)
```

Arguments:

disp

The display into which the event will be injected. You can obtain a handle to the display by either screen_get_context_property() or screen_get_window_property() functions.

ev

An event handle that was created with screen_create_event(). This event must contain all the relevant event data pertaining to its type when injected into the system.

Library:

libscreen

Description:

Function Type: *Immediate Execution* (p. 183)

A window manager and an input provider can use this function when they need to inject an event in the system. You need to be within a privileged context to be able to inject input events. You can create a privileged context by calling the function screen_create_context() with a context type of SCREEN_WINDOW_MANAGER_CONTEXT or SCREEN_INPUT_PROVIDER_CONTEXT. Prior to calling screen_inject_event(), you must have set all relevant event properties to valid values - especially the event type property. When using screen_inject_event(), the event will be sent to the window that has input focus on the specified display. If you want to send an event to a particular window other than the one who has input focus, then use screen_send_event().

Returns:

`0` if the event was sent to the window that has input focus on the display, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_send_event()

Send an input event to a process.

Synopsis:

```
#include <screen/screen.h>

int screen_send_event(screen_context_t ctx,
                      screen_event_t ev,
                      pid_t pid)
```

Arguments:

ctx

A context within Screen that was created with screen_create_context().

ev

An event handle that was created with screen_create_event(). This event must contain all the relevant event data pertaining to its type when injected into the system.

pid

The process the event is to be sent to.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

A window manager and an input provider can use this function when they need to inject an event in the system. You need to be within a privileged context to be able to inject input events. You can create a privileged context by calling the function screen_create_context() with a context type of SCREEN_WINDOW_MANAGER_CONTEXT or SCREEN_INPUT_PROVIDER_CONTEXT. Prior to calling screen_inject_event(), you must have set all relevant event properties to valid values - especially the event type property. When using screen_inject_event(), the event will be sent to the window that has input focus on the specified display. If you want to send an event to a particular window other than the one who has input focus, then use screen_send_event().

Returns:

`0` if the event was sent to the specified process, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_event_property_cv()

Set the value of the specified event property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_set_event_property_cv(screen_event_t ev,
                                 int pname,
                                 int len,
                                 const char *param)
```

Arguments:

ev

The handle of the event whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be read from `param`.

param

A pointer to a buffer containing the new value(s). This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function sets the value of an event property from a user-provided buffer. No more than `len` bytes of the specified type will be written. The list of properties that can be set per event type are listed as follows:

Event Type: SCREEN_EVENT_CREATE

- SCREEN_PROPERTY_GROUP

Returns:

`0` if the value(s) of the property was set to new value(s), or `-1` if an error occurred error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_event_property_iv()

Set the value of the specified event property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_event_property_iv(screen_event_t ev,
                                 int pname,
                                 const int *param)
```

Arguments:

ev

The handle of the event whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function sets the value of an event property from a user-provided buffer. The list of properties that can be set per event type are listed as follows:

Event Type: Any

- `SCREEN_PROPERTY_TYPE`
- `SCREEN_PROPERTY_SCALE_FACTOR`

Event Type: `SCREEN_EVENT_DISPLAY`

- `SCREEN_PROPERTY_ATTACHED`
- `SCREEN_PROPERTY_DISPLAY`
- `SCREEN_PROPERTY_MIRROR_MODE`
- `SCREEN_PROPERTY_MODE`
- `SCREEN_PROPERTY_PROTECTION_ENABLE`

Event Type: `SCREEN_EVENT_EFFECT_COMPLETE`

- `SCREEN_PROPERTY_EFFECT`

Event Type: `SCREEN_EVENT_GAMEPAD`

- `SCREEN_PROPERTY_BUTTONS`
- `SCREEN_PROPERTY_DEVICE`
- `SCREEN_PROPERTY_ANALOG0`
- `SCREEN_PROPERTY_ANALOG1`
- `SCREEN_PROPERTY_KEY_MODIFIERS`

Event Type: `SCREEN_EVENT_IDLE`

- `SCREEN_PROPERTY_IDLE_STATE`

Event Type: `SCREEN_EVENT_INPUT`

- `SCREEN_PROPERTY_DEVICE`
- `SCREEN_PROPERTY_INPUT_VALUE`

Event Type: SCREEN_EVENT_JOG

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_JOG_COUNT

Event Type: SCREEN_EVENT_JOYSTICK

- SCREEN_PROPERTY_BUTTONS
- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_ANALOG0
- SCREEN_PROPERTY_KEY_MODIFIERS

Event Type: SCREEN_EVENT_KEYBOARD

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_KEY_CAP
- SCREEN_PROPERTY_KEY_FLAGS
- SCREEN_PROPERTY_KEY_MODIFIERS
- SCREEN_PROPERTY_KEY_SCAN
- SCREEN_PROPERTY_KEY_SYM
- SCREEN_PROPERTY_SEQUENCE_ID

Event Types: SCREEN_EVENT_MTOUCH_TOUCH, SCREEN_EVENT_MTOUCH_MOVE, SCREEN_EVENT_MTOUCH_RELEASE

- SCREEN_PROPERTY_BUTTONS
- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_KEY_MODIFIERS
- SCREEN_PROPERTY_POSITION
- SCREEN_PROPERTY_SEQUENCE_ID
- SCREEN_PROPERTY_SIZE
- SCREEN_PROPERTY_SOURCE_POSITION
- SCREEN_PROPERTY_SOURCE_SIZE
- SCREEN_PROPERTY_TOUCH_ID
- SCREEN_PROPERTY_TOUCH_ORIENTATION
- SCREEN_PROPERTY_TOUCH_PRESSURE
- SCREEN_PROPERTY_TOUCH_TYPE

Event Type: SCREEN_EVENT_POINTER

- SCREEN_PROPERTY_BUTTONS
- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_KEY_MODIFIERS
- SCREEN_PROPERTY_MOUSE_HORIZONTAL_WHEEL
- SCREEN_PROPERTY_MOUSE_WHEEL

- SCREEN_PROPERTY_POSITION
- SCREEN_PROPERTY_SOURCE_POSITION

Event Type: SCREEN_EVENT_PROPERTY

- SCREEN_PROPERTY_NAME

Event Type: SCREEN_EVENT_USER

- SCREEN_PROPERTY_USER_DATA

Returns:

0 if the value(s) of the property was set to new value(s), or -1 if an error occurred error occurred (errno is set; refer to /usr/include/errno.h for more details).

screen_set_event_property_llv()

Set the current value of the specified event property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_event_property_llv(screen_event_t ev,
                                  int pname,
                                  const long long *param)
```

Arguments:

ev

The handle of the event whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type long long.

Library:

libscreen

Description:

Function Type: *Immediate Execution* (p. 183)

This function sets the value of an event property from a user-provided array.

Currently, there are no event properties that can be set using this function.

Returns:

`0` if the value(s) of the property was set to new value(s), or `-1` if an error occurred error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_event_property_pv()

Set the value of the specified event property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_set_event_property_pv(screen_event_t ev,
                                 int pname,
                                 void **param)
```

Arguments:

ev

The handle of the event whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function sets the value of an event property from a user-provided array. The list of properties that can be set per event type are listed as follows:

Event Type: SCREEN_EVENT_CREATE

- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_DISPLAY

- SCREEN_PROPERTY_DISPLAY

Event Type: SCREEN_EVENT_GAMEPAD

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_IDLE

- SCREEN_PROPERTY_DISPLAY
- SCREEN_PROPERTY_GROUP

Event Type: SCREEN_EVENT_INPUT

- SCREEN_PROPERTY_DEVICE

Event Type: SCREEN_EVENT_JOG

- SCREEN_PROPERTY_DEVICE

Event Type: SCREEN_EVENT_JOYSTICK

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_KEYBOARD

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_WINDOW

Event Types: SCREEN_EVENT_MTOUCH_TOUCH, SCREEN_EVENT_MTOUCH_MOVE, SCREEN_EVENT_MTOUCH_RELEASE

- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_POINTER

- SCREEN_PROPERTY_DEVICE
- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_POST

- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_PROPERTY

- SCREEN_PROPERTY_GROUP
- SCREEN_PROPERTY_DISPLAY
- SCREEN_PROPERTY_WINDOW

Event Type: SCREEN_EVENT_UNREALIZE

- SCREEN_PROPERTY_WINDOW

Returns:

0 if the value(s) of the property was set to new value(s), or -1 if an error occurred (errno is set; refer to /usr/include/errno.h for more details).

Groups (screen.h)

A window group is used to organize windows who share properties and a context.

You can query and set properties based on the group type. These properties when set, are then applied across each window in the group. Properties that windows can share when in the same group include:

- idle time
- keyboard focus
- multi-touch focus

All windows in the same group also share the same context.

Screen group properties

Types of properties that are associated with Screen group API objects.

Full read/write access to Screen API object properties is system dependent.

These properties are described in full under *Screen property types* (p. 200).

Group property	Gettable?	Settable?
SCREEN_PROPERTY_NAME	Yes	Yes
SCREEN_PROPERTY_USER_HANDLE	Yes	Yes
SCREEN_PROPERTY_IDLE_STATE	Yes	Yes
SCREEN_PROPERTY_IDLE_TIMEOUT	Yes	Yes
SCREEN_PROPERTY_KEYBOARD_FOCUS	Yes	Yes
SCREEN_PROPERTY_MTOUCH_FOCUS	Yes	Yes
SCREEN_PROPERTY_POINTER_FOCUS	Yes	Yes
SCREEN_PROPERTY_CONTEXT	Yes	Yes

screen_create_group()

Create a window group.

Synopsis:

```
#include <screen/screen.h>

int screen_create_group(screen_group_t *pgrp,
                        screen_context_t ctx)
```

Arguments:

pgrp

The handle of the group.

ctx

The connection to the composition manager. This context must have been created with screen_create_context().

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function creates a window group given a group object and a context. The context is shared by all windows in this group. You can use groups in order to organize your application windows.

Returns:

`0` if a new window group was created, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_destroy_group()

Destroy a window group.

Synopsis:

```
#include <screen/screen.h>

int screen_destroy_group(screen_group_t grp)
```

Arguments:

grp

The window group to be destroyed. The group must have been created with screen_create_group().

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function destroys a window group given a screen_group_t instance. When a window group is destroyed, all windows that belonged to the group are no longer associated with the group. You must destroy each screen_group_t after it is no longer needed.

Returns:

`0` if the window group was destroyed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_group_property_cv()

Retrieve the current value of the specified group property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_get_group_property_cv(screen_group_t grp,
                                 int pname,
                                 int len,
                                 char *param)
```

Arguments:

grp

The handle of the group whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be written to `param`.

param

The buffer where the retrieved value(s) will be stored. This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of group property in a user-provided buffer. No more than `len` bytes of the specified type will be written.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_NAME`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_group_property_iv()

Retrieve the current value of the specified group property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_group_property_iv(screen_group_t grp,
                                 int pname,
                                 int *param)
```

Arguments:

grp

The handle of the group whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of group property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_BUFFER_POOL`
- `SCREEN_PROPERTY_IDLE_STATE`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_group_property_llv()

Retrieve the current value of the specified group property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_group_property_llv(screen_group_t grp,
                                  int pname,
                                  long long *param)
```

Arguments:

grp

The handle of the group whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of group property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_IDLE_TIMEOUT`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_group_property_pv()

Retrieve the current value of the specified group property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_get_group_property_pv(screen_group_t grp,
                                 int pname,
                                 void **param)
```

Arguments:

grp

The handle of the group whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for query are of type *Screen property types* (p. 200).

param

The buffer where the retrieved value(s) will be stored. This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function stores the current value of group property in a user-provided buffer.

The values of the following properties can be retrieved using this function:

- `SCREEN_PROPERTY_CONTEXT`
- `SCREEN_PROPERTY_KEYBOARD_FOCUS`
- `SCREEN_PROPERTY_MTOUCH_FOCUS`
- `SCREEN_PROPERTY_POINTER_FOCUS`
- `SCREEN_PROPERTY_USER_HANDLE`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_group_t

A handle for the screen group.

Synopsis:

```
#include <screen/screen.h>

typedef struct _screen_group* screen_group_t;
```

Library:

`libscreen`

Description:

screen_set_group_property_cv()

Set the value of the specified group property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_set_group_property_cv(screen_group_t grp,
                                 int pname,
                                 int len,
                                 const char *param)
```

Arguments:

grp

The handle of the group whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be read from `param`.

param

A pointer to a buffer containing the new value(s). This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a group property from a user-provided buffer. You can use this function to set the value of the following properties:

- SCREEN_PROPERTY_NAME

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_group_property_iv()

Set the value of the specified group property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_group_property_iv(screen_group_t grp,
                                 int pname,
                                 const int *param)
```

Arguments:

grp

The handle of the group whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a group property from a user-provided buffer. You can use this function to set the value of the following properties:

- SCREEN_PROPERTY_BUFFER_POOL

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_group_property_llv()

Set the value of the specified group property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_group_property_llv(screen_group_t grp,
                                  int pname,
                                  const long long *param)
```

Arguments:

grp

The handle of the group whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a group property from a user-provided buffer. You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_IDLE_TIMEOUT`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_group_property_pv()

Set the value of the specified group property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_set_group_property_pv(screen_group_t grp,
                                 int pname,
                                 void **param)
```

Arguments:

grp

The handle of the group whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a group property from a user-provided buffer. You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_KEYBOARD_FOCUS`
- `SCREEN_PROPERTY_MTOUCH_FOCUS`
- `SCREEN_PROPERTY_POINTER_FOCUS`
- `SCREEN_PROPERTY_USER_HANDLE`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

Pixmaps (screen.h)

A pixmap is an off-screen rendering target.

The information and state variables associated with each pixmap is stored in memory allocated when the pixmap is created with *screen_create_pixmap()*. The composited windowing system keeps track of pixmaps that are allocated to ensure resources are released when the application terminates.

Before a pixmap can be used for rendering, a buffer must be created with *screen_create_pixmap_buffer()* or attached with *screen_attach_pixmap_buffer()*. Provided that the usage flags are set appropriately before creating or attaching the pixmap buffer, the contents of pixmaps can then be updated using various rendering APIs. The rendering can be made visible by copying parts of the pixmap to a window using *screen_blit()*.

Pixmaps are restricted to a single buffer. If the pixmap's buffer size hasn't been set explicitly, the buffer size will default to the size of the first display of the pixmap. Trying to change the buffer size or the usage once a buffer has already been allocated will result in an error. Additionally, trying to change the pixel format once the buffer has been added will also result in an error - unless the depth is identical, e.g. changing between RGBA8888 and RGBX8888 is acceptable. Note that buffers cannot be detached from pixmaps.

Screen pixmap metric count types

Types of metric counts for pixmaps.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_PIXMAP_METRIC_BLIT_COUNT = 0
      SCREEN_PIXMAP_METRIC_BLIT_PIXELS = 1
      SCREEN_PIXMAP_METRIC_BLIT_READS = 2
      SCREEN_PIXMAP_METRIC_BLIT_WRITES = 3
};
```

Data:

SCREEN_PIXMAP_METRIC_BLIT_COUNT

The number of blit requests (when the pixmap was a target of a blit) since the last time Screen pixmap metrics were queried.

SCREEN_PIXMAP_METRIC_BLIT_PIXELS

The number of pixels affected by the blit requests (when the pixmap was a target of a blit) since the last time Screen pixmap metrics have been queried.

SCREEN_PIXMAP_METRIC_BLIT_READS

An estimate of the number of bytes that has been read from the pixmap since the last time Screen pixmap metrics were queried.

The number of bytes read is an estimation calculated based on the number of pixels affected by the blit requests.

SCREEN_PIXMAP_METRIC_BLIT_WRITES

An estimate of the number of bytes that has been written to the pixmap since the last time Screen pixmap metrics were queried.

The number of bytes written is an estimation calculated based on the number of pixels affected by the blit requests.

Library:

`libscreen`

Description:

The metrics are on a per pixmap basis and the counts are reset after being queried. That is, the counts are reset to `0` after you call screen_get_pixmap_property_llv() to retrieve `SCREEN_PROPERTY_METRICS`.

Screen pixmap properties

Types of properties that are associated with Screen pixmap API objects.

Full read/write access to Screen API object properties is system dependent.

These properties are described in full under *Screen property types* (p. 200).

Pixmap property	Gettable?	Settable?
SCREEN_PROPERTY_ALPHA_MODE	Yes	Yes
SCREEN_PROPERTY_BUFFER_SIZE	Yes	Yes
SCREEN_PROPERTY_GROUP	Yes	Yes
SCREEN_PROPERTY_ID_STRING	Yes	Yes
SCREEN_PROPERTY_RENDER_BUFFERS	Yes	Yes
SCREEN_PROPERTY_USAGE	Yes	No

Pixmap property	Gettable?	Settable?
SCREEN_PROPERTY_CONTEXT	Yes	Yes
SCREEN_PROPERTY_METRIC_COUNT	Yes	No
SCREEN_PROPERTY_METRICS	Yes	No

screen_attach_pixmap_buffer()

Associate an externally allocated buffer with a pixmap.

Synopsis:

```
#include <screen/screen.h>

int screen_attach_pixmap_buffer(screen_pixmap_t pix,
                                screen_buffer_t buf)
```

Arguments:

pix

The handle of a pixmap that does not already have a buffer created or associated to it.

buf

A buffer that was allocated by the application.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function can be used to force a pixmap to use a buffer that was allocated by the application. Since pixmaps can have only one buffer, it is not possible to call this function or screen_create_pixmap_buffer() more than once. Whoever allocates the buffer is required to meet all alignment and granularity constraints required for the usage flags. The buffer `buf` must have been created with the function screen_create_buffer(), screen_create_pixmap_buffer(), or screen_create_window_buffers().

Returns:

`0` if the buffer was used by the specified pixmap, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_create_pixmap()

Create a pixmap that can be used to do off-screen rendering.

Synopsis:

```
#include <screen/screen.h>

int screen_create_pixmap(screen_pixmap_t *ppix,
                         screen_context_t ctx)
```

Arguments:

ppix

An address where the function can store the handle to the newly created native pixmap.

ctx

The connection to the composition manager. This context must have been created with screen_create_context().

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function creates a pixmap object, which is an off-screen rendering target. The results of this rendering can later be copied to a window object. Applications must use screen_destroy_pixmap() when a pixmap is no longer used.

Returns:

`0` if a new pixmap was created,or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_create_pixmap_buffer()

Send a request to the composition manager to add a new buffer to a pixmap.

Synopsis:

```
#include <screen/screen.h>

int screen_create_pixmap_buffer(screen_pixmap_t pix)
```

Arguments:

pix

The handle of the pixmap for which a new buffer will be created.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function to adds a buffer to a pixmap. A buffer cannot be created if a buffer was previously attached using screen_attach_pixmap_buffer().

Returns:

`0` if a new pixmap buffer was created,or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_destroy_pixmap()

Destroy a pixmap and frees associated resources.

Synopsis:

```
#include <screen/screen.h>

int screen_destroy_pixmap(screen_pixmap_t pix)
```

Arguments:

pix

The handle of the pixmap which is to be destroyed.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function destroys the pixmap associated with the specified pixmap. Any resources and buffer created for this pixmap, whether locally or by the composition manager, will be released. The pixmap handle can no longer be used as argument in subsequent screen calls. Pixmap buffers that are not created by composition manager but are registered with screen_attach_pixmap_buffer() are not freed by this operation. The application is responsible for freeing its own external buffers.

Returns:

`0` if the pixmap buffer was destroyed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_destroy_pixmap_buffer()

Send a request to the composition manager to destory the buffer of the specified pixmap.

Synopsis:

```
#include <screen/screen.h>

int screen_destroy_pixmap_buffer(screen_pixmap_t pix)
```

Arguments:

pix

The handle of the pixmap whose buffer is to be destroyed.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function releases the buffer that was allocated for a pixmap, without destroying the pixmap. If the buffer was created with screen_create_pixmap_buffer(), the memory is released and can be used for other window or pixmap buffers. If the buffer was attached using screen_attach_pixmap_buffer(), the buffer is destroyed but no memory is actually released. In this case the application is responsible for freeing the memory after calling screen_destroy_pixmap_buffer(). Once a pixmap buffer has been destroyed, you can change the format, usage and buffer size before creating a new buffer again. The memory that is released by this call is not reserved and can be used for any subsequent buffer allocation by the windowing system.

Returns:

`0` if the memory used by the pixmap buffer was freed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_pixmap_property_cv()

Retrieve the current value of the specified pixmap property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_get_pixmap_property_cv(screen_pixmap_t pix,
                                  int pname,
                                  int len,
                                  char *param)
```

Arguments:

pix

The handle of the pixmap whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for querying are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be written to `param`.

param

A pointer to a buffer containing the new value(s). This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function retrieves the value of pixmap property from a user-provided buffer. The values of the following properties can be queried using this function:

- `SCREEN_PROPERTY_GROUP`
- `SCREEN_PROPERTY_ID_STRING`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_pixmap_property_iv()

Retrieve the current value of the specified pixmap property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_pixmap_property_iv(screen_pixmap_t pix,
                                  int pname,
                                  int *param)
```

Arguments:

pix

The handle of the pixmap whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for querying are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function retrieves the value of pixmap property from a user-provided buffer. The values of the following properties can be queried using this function:

- `SCREEN_PROPERTY_ALPHA_MODE`
- `SCREEN_PROPERTY_COLOR_SPACE`
- `SCREEN_PROPERTY_FORMAT`
- `SCREEN_PROPERTY_USAGE`
- `SCREEN_PROPERTY_BUFFER_SIZE`
- `SCREEN_PROPERTY_METRIC_COUNT`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_pixmap_property_llv()

Retrieve the current value of the specified pixmap property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_pixmap_property_llv(screen_pixmap_t pix,
                                   int pname,
                                   long long *param)
```

Arguments:

pix

The handle of the pixmap whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for querying are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function retrieves the value of pixmap property from a user-provided array. The values of the following properties can be queried using this function:

- `SCREEN_PROPERTY_METRICS`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_pixmap_property_pv()

Retrieve the current value of the specified pixmap property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_get_pixmap_property_pv(screen_pixmap_t pix,
                                  int pname,
                                  void **param)
```

Arguments:

pix

The handle of the pixmap whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for querying are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function retrieves the value of pixmap property from a user-provided buffer. The values of the following properties can be queried using this function:

- `SCREEN_PROPERTY_CONTEXT`
- `SCREEN_PROPERTY_GROUP`
- `SCREEN_PROPERTY_RENDER_BUFFERS`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_join_pixmap_group()

Cause a pixmap to join a group.

Synopsis:

```
#include <screen/screen.h>

int screen_join_pixmap_group(screen_pixmap_t pix,
                             const char *name)
```

Arguments:

pix

The handle of the pixmap that is to be joining the group.

name

A unique string that identifies the group.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function is used to add a pixmap to a group.

Returns:

`0` if the request for the pixmap to join the group was queued for processing, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_leave_pixmap_group()

Cause a pixmap to leave a group.

Synopsis:

```
#include <screen/screen.h>

int screen_leave_pixmap_group(screen_pixmap_t pix)
```

Arguments:

pix

The handle of the pixmap that is to be leaving the group.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function is used to remove a pixmap from a group.

Returns:

`0` if the request for the pixmap to leave the group was queued for processing, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_pixmap_t

A handle for the screen pixmap.

Synopsis:

```
#include <screen/screen.h>

typedef struct _screen_pixmap* screen_pixmap_t;
```

Library:

`libscreen`

Description:

screen_ref_pixmap()

Create a reference to a pixmap.

Synopsis:

```
#include <screen/screen.h>

int screen_ref_pixmap(screen_pixmap_t pix)
```

Arguments:

pix

The handle of the pixmap for which the reference is to be created.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function creates a reference to a pixmap. This function can be used by libraries to prevent the pixmap or its buffer from disappearing while the library is making use of it. The pixmap and its buffer will not be destroyed until all references have been cleared with screen_unref_pixmap(). In the event that a pixmap is destroyed before

the reference is cleared, screen_unref_pixmap() will cause the pixmap buffer and/or the pixmap to be destroyed.

Returns:

`0` if the reference to the specified window was created, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_pixmap_property_cv()

Set the value of the specified pixmap property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_set_pixmap_property_cv(screen_pixmap_t pix,
                                  int pname,
                                  int len,
                                  const char *param)
```

Arguments:

pix

handle of the pixmap whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be read from `param`.

param

A pointer to a buffer containing the new value(s). This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a pixmap property from a user-provided buffer. You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_ID_STRING`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_pixmap_property_iv()

Set the value of the specified pixmap property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_pixmap_property_iv(screen_pixmap_t pix,
                                  int pname,
                                  const int *param)
```

Arguments:

pix

handle of the pixmap whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a pixmap property from a user-provided buffer. You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_ALPHA_MODE`
- `SCREEN_PROPERTY_COLOR_SPACE`
- `SCREEN_PROPERTY_FORMAT`
- `SCREEN_PROPERTY_USAGE`
- `SCREEN_PROPERTY_BUFFER_SIZE`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_pixmap_property_llv()

Set the value of the specified pixmap property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_pixmap_property_llv(screen_pixmap_t pix,
                                   int pname,
                                   const long long *param)
```

Arguments:

pix

handle of the pixmap whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a pixmap property from a user-provided buffer. Currently, there are no pixmap properties that can be set using this function.

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_pixmap_property_pv()

Set the value of the specified pixmap property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_set_pixmap_property_pv(screen_pixmap_t pix,
                                  int pname,
                                  void **param)
```

Arguments:

pix

handle of the pixmap whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a pixmap property from a user-provided buffer. You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_CONTEXT`
- `SCREEN_PROPERTY_GROUP`
- `SCREEN_PROPERTY_RENDER_BUFFERS`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_unref_pixmap()

Remove a reference from a specified pixmap.

Synopsis:

```
#include <screen/screen.h>

int screen_unref_pixmap(screen_pixmap_t pix)
```

Arguments:

pix

The handle of the pixmap for which the reference is to be removed.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function removes a reference to a pixmap. If the pixmap and its buffer haven't been destroyed yet, the effect of screen_unref_pixmap() is simply to decrease a reference count. If the pixmap or the pixmap buffer was destroyed while still being referenced, screen_unref_pixmap() will cause the pixmap and/or its buffer to be destroyed when the reference count reaches zero.

Returns:

`0` if the reference to the specified pixmap was removed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

Windows (screen.h)

A window is used to display different types of content.

The information and state variables associated with each window are stored in memory allocated when the window is created with *screen_create_window()*.

A window exists in the composited windowing system space. A window will not be visible until it is associated with at least one buffer that has been created with *screen_create_window_buffers()*, and one frame has been posted with *screen_post_window()*.

If a window's buffer size has not been set explicitly before the first buffer is created, the source size is used as the default value. If the window's source size has not been set explicitly, its destination size is used as the default value. If the destination size has not been set either, the buffer size will default to the screen size. Trying to change the usage once buffers have already been allocated will result in an error. Buffers must be created before a frame can be posted with *screen_post_window()*.

Screen window metric count types

Types of metric counts for windows.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_WINDOW_METRIC_OBJECT_COUNT = 0
      SCREEN_WINDOW_METRIC_API_COUNT = 1
      SCREEN_WINDOW_METRIC_DRAW_COUNT = 2
      SCREEN_WINDOW_METRIC_TRIANGLE_COUNT = 3
      SCREEN_WINDOW_METRIC_VERTEX_COUNT = 4
      SCREEN_WINDOW_METRIC_IMAGE_DATA_BYTES = 5
      SCREEN_WINDOW_METRIC_BUFFER_DATA_BYTES = 6
      SCREEN_WINDOW_METRIC_EVENT_COUNT = 7
      SCREEN_WINDOW_METRIC_BLIT_COUNT = 8
      SCREEN_WINDOW_METRIC_BLIT_PIXELS = 9
      SCREEN_WINDOW_METRIC_BLIT_READS = 10
      SCREEN_WINDOW_METRIC_BLIT_WRITES = 11
      SCREEN_WINDOW_METRIC_POST_COUNT = 12
      SCREEN_WINDOW_METRIC_POST_PIXELS = 13
      SCREEN_WINDOW_METRIC_UPDATE_COUNT = 14
      SCREEN_WINDOW_METRIC_UPDATE_PIXELS = 15
      SCREEN_WINDOW_METRIC_UPDATE_READS = 16
      SCREEN_WINDOW_METRIC_UPDATE_WRITES = 17
      SCREEN_WINDOW_METRIC_CPU_TIME = 18
      SCREEN_WINDOW_METRIC_GPU_TIME = 19
      SCREEN_WINDOW_METRIC_VISIBLE_TIME = 20
      SCREEN_WINDOW_METRIC_FULLY_VISIBLE_TIME = 21
};
```

Data:

SCREEN_WINDOW_METRIC_OBJECT_COUNT

A general purpose counter whose meaning is defined by Cascades, Screen and other SDKs (e.g., WebKit, Adobe AIR, ...).

SCREEN_WINDOW_METRIC_API_COUNT

The number of OpenGL ES 1.X, OpenGL ES 2.X, and OpenVG API calls that were made by the process owning the window since the last time Screen window metrics were queried.

Note that if multiple processes, other than the one that owns the window, made OpenGL ES 1.X, OpenGL ES 2.X, OpenVG API calls to the window, these API calls would not be counted.

SCREEN_WINDOW_METRIC_DRAW_COUNT

The number of draw API calls (e.g., glDrawArrays(), glDrawElements(), ...) that were made by in the window since the last time Screen window metrics were queried.

This metric is not counted for OpenVG API calls.

SCREEN_WINDOW_METRIC_TRIANGLE_COUNT

An estimate of the number of triangles drawn in the window since the last time Screen window metrics were queried.

This count is an estimate because two triangles are counted per line and two triangles are also counted per point. This metric is not counted for OpenVG API calls.

SCREEN_WINDOW_METRIC_VERTEX_COUNT

An estimate of the number of vertices passed to OpenGL in the window since the last time Screen window metrics were queried.

This metric is not counted for OpenVG API calls.

SCREEN_WINDOW_METRIC_IMAGE_DATA_BYTES

An estimate of the number of bytes requested to upload the texture in the window since the last time Screen window metrics were queried.

This metric is not counted for OpenVG API calls.

SCREEN_WINDOW_METRIC_BUFFER_DATA_BYTES

An estimate of the number of bytes uploaded to vertex buffers in the window (e.g., from calls such as glBufferData(), glBufferSubData(), ...) since the last time Screen window metrics were queried.

This metric is not counted for OpenVG API calls.

SCREEN_WINDOW_METRIC_EVENT_COUNT

The number of events that are sent directly to the window since the last time Screen window metrics were queried.

This metric doesn't include events for any children windows that the window may have.

SCREEN_WINDOW_METRIC_BLIT_COUNT

The number of blit requests (when the window was a target of a blit) since the last time Screen window metrics were queried.

SCREEN_WINDOW_METRIC_BLIT_PIXELS

The number of pixels affected by the blit requests (when the window was a target of a blit) since the last time Screen window metrics have been queried.

SCREEN_WINDOW_METRIC_BLIT_READS

An estimate of the number of bytes that have been read from the window since the last time Screen window metrics were queried.

The number of bytes read is an estimation calculated based on the number of pixels affected by the blit requests.

SCREEN_WINDOW_METRIC_BLIT_WRITES

An estimate of the number of bytes that have been written to the window since the last time Screen window metrics were queried.

The number of bytes written is an estimate based on the number of pixels affected by the blit requests.

SCREEN_WINDOW_METRIC_POST_COUNT

The number times that the window has posted since the last time Screen window metrics were queried.

SCREEN_WINDOW_METRIC_POST_PIXELS

The number of pixels that were marked as dirty in all of the window's posts since the last time Screen window metrics were queried.

SCREEN_WINDOW_METRIC_UPDATE_COUNT

The number of times that the window was in an update since the last time Screen window metrics were queried.

The window must be visible (its `SCREEN_PROPERTY_VISIBLE` is set) in order for this count to be incremented. If the window is static (i.e., the window property `SCREEN_PROPERTY_STATIC` is set), this count can still increment if there is another window or layer on top so that there is blending required for this window.

SCREEN_WINDOW_METRIC_UPDATE_PIXELS

The number of pixels that has been used in the updates of the window since the last time Screen window metrics were queried.

SCREEN_WINDOW_METRIC_UPDATE_READS

An estimate of the number of bytes that have been read from the window buffer (if there are multiple buffers, it's the front buffer) since the last time Screen window metrics were queried.

The number of bytes read is an estimate based on the number of pixels affected by the update.

SCREEN_WINDOW_METRIC_UPDATE_WRITES

An estimate of the number of bytes that has been written to the window framebuffer since the last time Screen window metrics have been queried.

The number of bytes written is an estimate based on the number of pixels affected by the update.

SCREEN_WINDOW_METRIC_CPU_TIME

An estimate of the total CPU time spent preparing updates.

The quantity is estimated by measuring the time between the window timestamp property and the time when screen_post_window() is called. The SCREEN_PROPERTY_TIMESTAMP must be set on the window for this metric to be reliable.

SCREEN_WINDOW_METRIC_GPU_TIME

An estimate of the total GPU time spent rendering to back buffers.

The quantity is estimated by measuring the time between when `eglSwapBuffers()` is called and when the post is actually flushed out to the server. This metric is only reliable if the GPU does most of its rendering after `eglSwapBuffers()` is called.

SCREEN_WINDOW_METRIC_VISIBLE_TIME

An estimate of the total number of nanoseconds for which the window was visible.

The quantity is estimated by measuring the time spent between scene rebuilds where the window is at least partially visible. If the window is covered by another window with transparency, the counter will be incremented.

SCREEN_WINDOW_METRIC_FULLY_VISIBLE_TIME

An estimate of the total number of nanoseconds for which the window was fully visible.

The quantity is estimated by measuring the time spent between scene rebuilds where the window is completely visible. If the window is covered by another window with transparency, the counter will not be incremented even though the window may actually be visible.

Library:

`libscreen`

Description:

The metrics are on a per-window basis, and the counts are reset after being queried. That is, the counts are reset to `0` after you call screen_get_window_property_llv() to retrieve SCREEN_PROPERTY_METRICS.

Screen window properties

Types of properties that are associated with Screen window API objects.

Full read/write access to Screen API object properties is system dependent.

These properties are described in full under *Screen property types* (p. 200)

Window property	Configurable?	Gettable?	Settable?
SCREEN_PROPERTY_ALPHA_MODE	Yes	Yes	Yes
SCREEN_PROPERTY_BRIGHTNESS	Yes	Yes	Yes
SCREEN_PROPERTY_BUFFER_COUNT	Yes	Yes	Yes
SCREEN_PROPERTY_BUFFER_SIZE	Yes	Yes	Yes
SCREEN_PROPERTY_CLASS	Yes	Yes	Yes
SCREEN_PROPERTY_COLOR_SPACE	No	Yes	Yes
SCREEN_PROPERTY_CONTRAST	Yes	Yes	Yes
SCREEN_PROPERTY_DISPLAY	Yes	Yes	Yes
SCREEN_PROPERTY_FLIP	No	Yes	Yes
SCREEN_PROPERTY_FORMAT	Yes	Yes	Yes
SCREEN_PROPERTY_FRONT_BUFFER	No	Yes	Yes
SCREEN_PROPERTY_GLOBAL_ALPHA	Yes	Yes	Yes
SCREEN_PROPERTY_PIPELINE	Yes	Yes	Yes
SCREEN_PROPERTY_GROUP	Yes	Yes	Yes
SCREEN_PROPERTY_HUE	Yes	Yes	Yes
SCREEN_PROPERTY_ID_STRING	Yes	Yes	Yes
SCREEN_PROPERTY_MIRROR	No	Yes	Yes
SCREEN_PROPERTY_OWNER_PID	No	Yes	Yes
SCREEN_PROPERTY_POSITION	Yes	Yes	Yes
SCREEN_PROPERTY_RENDER_BUFFERS	No	Yes	Yes
SCREEN_PROPERTY_ROTATION	Yes	Yes	Yes
SCREEN_PROPERTY_SATURATION	Yes	Yes	Yes
SCREEN_PROPERTY_SIZE	Yes	Yes	Yes
SCREEN_PROPERTY_SOURCE_POSITION	Yes	Yes	Yes
SCREEN_PROPERTY_SOURCE_SIZE	Yes	Yes	Yes

Window property	Configurable?	Gettable?	Settable?
SCREEN_PROPERTY_SWAP_INTERVAL	Yes	Yes	Yes
SCREEN_PROPERTY_SWAP_INTERVAL	Yes	Yes	Yes
SCREEN_PROPERTY_TRANSPARENCY	Yes	Yes	Yes
SCREEN_PROPERTY_TYPE	No	Yes	No
SCREEN_PROPERTY_USAGE	Yes	Yes	No
SCREEN_PROPERTY_USER_HANDLE	No	Yes	Yes
SCREEN_PROPERTY_VISIBLE	Yes	Yes	Yes
SCREEN_PROPERTY_RENDER_BUFFER_COUNT	No	Yes	No
SCREEN_PROPERTY_ZORDER	Yes	Yes	Yes
SCREEN_PROPERTY_SCALE_QUALITY	No	Yes	Yes
SCREEN_PROPERTY_SENSITIVITY	No	Yes	Yes
SCREEN_PROPERTY_CBABC_MODE	Yes	Yes	Yes
SCREEN_PROPERTY_CBABC_MODE	No	Yes	Yes
SCREEN_PROPERTY_FLOATING	No	Yes	Yes
SCREEN_PROPERTY_PROTECTION_ENABLE	No	Yes	Yes
SCREEN_PROPERTY_SOURCE_CLIP_POSITION	No	Yes	Yes
SCREEN_PROPERTY_SOURCE_CLIP_SIZE	No	Yes	Yes
SCREEN_PROPERTY_VIEWPORT_POSITION	No	Yes	Yes
SCREEN_PROPERTY_VIEWPORT_SIZE	No	Yes	Yes
SCREEN_PROPERTY_TIMESTAMP	No	Yes	Yes
SCREEN_PROPERTY_IDLE_MODE	No	Yes	Yes
SCREEN_PROPERTY_KEYBOARD_FOCUS	No	Yes	No
SCREEN_PROPERTY_CLIP_POSITION	Yes	Yes	Yes
SCREEN_PROPERTY_CLIP_SIZE	Yes	Yes	Yes
SCREEN_PROPERTY_COLOR	Yes	Yes	Yes
SCREEN_PROPERTY_CONTEXT	No	Yes	Yes
SCREEN_PROPERTY_DEBUG	No	Yes	Yes
SCREEN_PROPERTY_ALTERNATE_WINDOW	No	Yes	Yes
SCREEN_PROPERTY_SELF_LAYOUT	No	Yes	Yes

Window property	Configurable?	Gettable?	Settable?
SCREEN_PROPERTY_SCALE_FACTOR	No	Yes	Yes
SCREEN_PROPERTY_METRIC_COUNT	No	Yes	No
SCREEN_PROPERTY_METRICS	No	Yes	No
SCREEN_PROPERTY_BRUSH_CLIP_POSITION	No	Yes	Yes
SCREEN_PROPERTY_BRUSH_CLIP_SIZE	No	Yes	Yes
SCREEN_PROPERTY_BRUSH	No	Yes	Yes
SCREEN_PROPERTY_TRANSFORM	No	Yes	No

Screen window types

Types of windows that can be created.

Synopsis:

```
#include <screen/screen.h>


 enum {
      SCREEN_APPLICATION_WINDOW = 0
      SCREEN_CHILD_WINDOW = 1
      SCREEN_EMBEDDED_WINDOW = 2
};
```

Data:

SCREEN_APPLICATION_WINDOW

A window type used to display the main application.

The X and Y coordinates are always relative to the dimensions of the display.

SCREEN_CHILD_WINDOW

A window type commonly used to display a dialog.

You must add a child window to an application's window group; otherwise the child window is invisible. A child window's display properties are relative to the application window to which it belongs. For example, the X and Y coordinates of the child window are all relative to the top left corner of the application window. This window type has its property, SCREEN_PROPERTY_FLOATING, defaulted to indicate that the window is floating.

SCREEN_EMBEDDED_WINDOW

A window type used to embed a window control within an object.

Like the child window, the X and Y coordinates of the embedded window are all relative to the top left corner of the application window. You must add an embedded window to an application's window group, otherwise the embedded window is invisible. This window type has its property, `SCREEN_PROPERTY_FLOATING`, defaulted to indicate that the window is non-floating.

Library:

`libscreen`

Description:

screen_attach_window_buffers()

Associate an externally allocated buffer with a window.

Synopsis:

```
#include <screen/screen.h>

int screen_attach_window_buffers(screen_window_t win,
                                 int count,
                                 screen_buffer_t *buf)
```

Arguments:

win

The handle of a window that doesn't already share a buffer with another window, and that doesn't have one or more buffers created or associated to it.

count

The number of buffers to be attached.

buf

An array of `count` buffers to be attached that was allocated by the application.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function can be used by drivers and other middleware components that must allocate their own buffers. The client must ensure that all usage constraints are met when allocating the buffers. Failure to do so may prevent the buffers from being successfully attached, or may result in artifacts and system instability. Calling both screen_attach_window_buffers() and screen_create_window_buffers() is not permitted.

Returns:

`0` if the buffers were successfully attached to the specified window, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_create_window()

Create a window that can be used to make graphical content visible on a display.

Synopsis:

```
#include <screen/screen.h>

int screen_create_window(screen_window_t *pwin,
                         screen_context_t ctx)
```

Arguments:

pwin

An address where the function can store the handle to the newly created native window.

ctx

The connection to the composition manager. This context must have been created with screen_create_context().

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function creates a window object. The window size defaults to full screen when it is created. This is equivalent to calling screen_create_window_type() with a type of SCREEN_APPLICATION_WINDOW.

Returns:

0 if a new window was created, or -1 if an error occurred (errno is set; refer to /usr/include/errno.h for more details).

screen_create_window_buffers()

Send a request to the composition manager to add new buffers to a window.

Synopsis:

```
#include <screen/screen.h>

int screen_create_window_buffers(screen_window_t win,
                                 int count)
```

Arguments:

win

The handle of the window for which the new buffers must be allocated.

count

The number of buffers to be created for this window.

Library:

libscreen

Description:

Function Type: *Flushing Execution* (p. 183)

This function adds buffers to a window. Windows need at least one buffer in order to be visible. Buffers cannot be created using screen_create_window_buffers() if at some point prior, buffers were attached to this window using screen_attach_window_buffers(). Buffers will be created with the size of SCREEN_PROPERTY_BUFFER_SIZE as set for the window.

Returns:

`0` if new buffers were created for the specified window, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_create_window_group()

Create a window group that other windows can join.

Synopsis:

```
#include <screen/screen.h>

int screen_create_window_group(screen_window_t win,
                               const char *name)
```

Arguments:

win

The handle of the window for which the group is created. This window must have been created with screen_create_window_type() with a type of `SCREEN_APPLICATION_WINDOW` or `SCREEN_CHILD_WINDOW`.

name

A unique string that will be used to identify the window group. This name must be communicated to any window wishing to join the group as a child of `win`. Other than uniqueness, there are no other constraints on this name (for example, lower case and special characters are permitted). If `name` is a `NULL` pointer, then a a string is generated for you with this format: auto-<pid>-<32 alpha-numeric characters> For example:

```
auto-1564694-00007FFF000000370000138A00002567
```

It is recommended that, unless a static name is explicitly required, you should call this function with `name` as `NULL` so that a unique group name is automatically generated. You can use screen_get_window_property_cv() with `SCREEN_PROPERTY_GROUP` as the property to retrieve the name of the window group.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function creates a window group and assigns it to the specified window. The group is identified by the name string, which must be unique. The request will fail if another group was previously created with the same name.

Windows can parent only one group. Therefore, screen_create_window_group() can be called successfully only once for any given window. Additionally, only windows of certain types can parent a group of windows. Windows with a type of `SCREEN_APPLICATION_WINDOW` can parent windows of type `SCREEN_CHILD_WINDOW` and `SCREEN_EMBEDDED_WINDOW`. Windows with a type of `SCREEN_CHILD_WINDOW` can also create a group and parent windows of type `SCREEN_EMBEDDED_WINDOW`.

Once a group is created, it exists until the window that parents the group is destroyed. When a parent window is destroyed, all children are orphaned and made invisible. Destroying a child has no effect on the group other than removing the window from the group.

Group owners have privileged access to the windows that they parent. When windows join the group, the parent will receive a `SCREEN_EVENT_CREATE` that contains a handle to the child window that can be used by the parent to set properties or send events. Conversely, the parent gets notified when a child window gets destroyed. The parent window is expected to destroy its local copy of the window handle when one of its children is destroyed.

Returns:

`0` if request for the new window group was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_create_window_type()

Create a new window of a specified type.

Synopsis:

```
#include <screen/screen.h>

int screen_create_window_type(screen_window_t *pwin,
                              screen_context_t ctx,
                              int type)
```

Arguments:

pwin

An address where the function can store the handle to the newly created native window.

ctx

The connection to the composition manager to be used to create the window. This context must have been created with screen_create_context().

type

The type of window to be created. `type` must be of type Screen_Window_Types.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function creates a window object of the specified type.

Returns:

`0` if a new window type was created, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_destroy_window()

Destroy a window and free associated resources.

Synopsis:

```
#include <screen/screen.h>

int screen_destroy_window(screen_window_t win)
```

Arguments:

win

The handle of the window to be destroyed. This must have been created with screen_create_window().

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function destroys the window associated with the given window handle. If the window is visible, it is removed from the display. Any resources or buffers created for this window, both locally and by the composition manager, are released.

The window handle can no longer be used as argument to subsequent screen calls. Buffers that are not created by the composition manager and registered with screen_attach_window_buffers() are not freed by this operation.

The application is responsible for releasing its own external buffers. Any window that shares buffers with the window is also destroyed. screen_destroy_window() must be used to free windows that were obtained by querying context or event properties. In this case, the window is not removed from its display and destroyed. Only the local state associated with the external window is released.

Returns:

`0` if the specified window was destroyed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_destroy_window_buffers()

Send a request to the composition manager to destroy buffers of the specified window.

Synopsis:

```
#include <screen/screen.h>

int screen_destroy_window_buffers(screen_window_t win)
```

Arguments:

win

The handle of the window whose buffer(s) you want to destroy.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function releases one or more buffers allocated for the specified window, without destroying the window. If buffers were created with screen_create_window_buffers(), the memory is released and can be used for other window or pixmap buffers. If buffers were attached using screen_attach_window_buffers(), these buffers are destroyed but no memory is actually released. In this case, the application is responsible for freeing the memory after calling screen_destroy_window_buffers(). Once a window's buffers have been destroyed, you can change the format, the usage and the buffer size before creating any new buffers again. The memory that is released by this call is not reserved and can be used for any subsequent buffer allocation by the windowing system.

Returns:

`0` if the memory used by the window buffer was freed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_discard_window_regions()

Discard the specified window regions.

Synopsis:

```
#include <screen/screen.h>

int screen_discard_window_regions(screen_window_t win,
                                  int count,
                                  const int *rects)
```

Arguments:

win

The handle of the window in which you want to specify regions to discard.

count

The number of rectangles (retangular regions) you want to discard, specified in the `rects` argument. The value of `count` can be `0`.

rects

An array of integers containing the x, y, width, and height coordinates of rectangles that bound areas in the window you want to discard. The `rects`

argument must provide at least 4 times `count` integers(quadruples of x, y, width and height).

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function is a hole-punching API. Use this function to specify window regions you want to discard. The regions behave as if they were transparent, or as if there were no transparency on the window. When you call this function, it invalidates any regions you might have defined previously. You can call the function with count set to 0 to remove discarded regions.

Returns:

`0` if the request for discarding window regions have been queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_get_window_property_cv()

Retrieve the current value of the specified window property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_get_window_property_cv(screen_window_t win,
                                  int pname,
                                  int len,
                                  char *param)
```

Arguments:

win

The handle of the window whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for querying are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be written to `param`.

param

A pointer to a buffer containing the new value(s). This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function retrieves the value of window property from a user-provided array. The values of the following properties can be queried using this function:

- `SCREEN_PROPERTY_CLASS`
- `SCREEN_PROPERTY_ID_STRING`
- `SCREEN_PROPERTY_GROUP`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_window_property_iv()

Retrieve the current value of the specified window property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_get_window_property_iv(screen_window_t win,
                                  int pname,
                                  int *param)
```

Arguments:

win

The handle of the window whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for querying are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function retrieves the value of window property from a user-provided array. The values of the following properties can be queried using this function:

- `SCREEN_PROPERTY_ALPHA_MODE`
- `SCREEN_PROPERTY_BUFFER_COUNT`
- `SCREEN_PROPERTY_COLOR_SPACE`
- `SCREEN_PROPERTY_FORMAT`
- `SCREEN_PROPERTY_OWNER_PID`
- `SCREEN_PROPERTY_RENDER_BUFFER_COUNT`
- `SCREEN_PROPERTY_SCALE_FACTOR`
- `SCREEN_PROPERTY_SIZE`
- `SCREEN_PROPERTY_SWAP_INTERVAL`
- `SCREEN_PROPERTY_USAGE`
- `SCREEN_PROPERTY_BRIGHTNESS`
- `SCREEN_PROPERTY_CBABC_MODE`
- `SCREEN_PROPERTY_CONTRAST`
- `SCREEN_PROPERTY_DEBUG`
- `SCREEN_PROPERTY_FLIP`
- `SCREEN_PROPERTY_FLOATING`
- `SCREEN_PROPERTY_GLOBAL_ALPHA`
- `SCREEN_PROPERTY_HUE`
- `SCREEN_PROPERTY_IDLE_MODE`
- `SCREEN_PROPERTY_KEYBOARD_FOCUS`
- `SCREEN_PROPERTY_MIRROR`

- SCREEN_PROPERTY_PIPELINE
- SCREEN_PROPERTY_PROTECTION_ENABLE
- SCREEN_PROPERTY_ROTATION
- SCREEN_PROPERTY_SATURATION
- SCREEN_PROPERTY_SCALE_QUALITY
- SCREEN_PROPERTY_SELF_LAYOUT
- SCREEN_PROPERTY_SENSITIVITY
- SCREEN_PROPERTY_STATIC
- SCREEN_PROPERTY_TRANSPARENCY
- SCREEN_PROPERTY_TYPE
- SCREEN_PROPERTY_VISIBLE
- SCREEN_PROPERTY_ZORDER
- SCREEN_PROPERTY_BUFFER_SIZE
- SCREEN_PROPERTY_CLIP_POSITION
- SCREEN_PROPERTY_CLIP_SIZE
- SCREEN_PROPERTY_POSITION
- SCREEN_PROPERTY_SOURCE_CLIP_POSITION
- SCREEN_PROPERTY_SOURCE_CLIP_SIZE
- SCREEN_PROPERTY_SOURCE_POSITION
- SCREEN_PROPERTY_SOURCE_SIZE
- SCREEN_PROPERTY_VIEWPORT_POSITION
- SCREEN_PROPERTY_VIEWPORT_SIZE
- SCREEN_PROPERTY_METRIC_COUNT
- SCREEN_PROPERTY_TRANSFORM
- SCREEN_PROPERTY_REFERENCE_COLOR

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_window_property_llv()

Retrieve the current value of the specified window property of type long long integer.

Synopsis:

```
#include <screen/screen.h>
```

```
int screen_get_window_property_llv(screen_window_t win,
                                   int pname,
                                   long long *param)
```

Arguments:

win

handle of the window whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for querying are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function retrieves the value of a window property from a user-provided array. The values of the following properties can be queried using this function:

- `SCREEN_PROPERTY_METRICS`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_get_window_property_pv()

Retrieve the current value of the specified window property of type void.*

Synopsis:

```
#include <screen/screen.h>
```

```
int screen_get_window_property_pv(screen_window_t win,
                                  int pname,
                                  void **param)
```

Arguments:

win

handle of the window whose property is being queried.

pname

The name of the property whose value is being queried. The properties available for querying are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function retrieves the value of a window property from a user-provided array. The values of the following properties can be queried using this function:

- `SCREEN_PROPERTY_ALTERNATE_WINDOW`
- `SCREEN_PROPERTY_CONTEXT`
- `SCREEN_PROPERTY_DISPLAY`
- `SCREEN_PROPERTY_FRONT_BUFFER`
- `SCREEN_PROPERTY_GROUP`
- `SCREEN_PROPERTY_RENDER_BUFFERS`
- `SCREEN_PROPERTY_USER_HANDLE`
- `SCREEN_PROPERTY_BRUSH`

Returns:

`0` if a query was successful and the value(s) of the property are stored in `param`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_join_window_group()

Cause a window to join a window group.

Synopsis:

```
#include <screen/screen.h>

int screen_join_window_group(screen_window_t win,
                             const char *name)
```

Arguments:

win

The handle for the window that is to join the group. This window must have been created with screen_create_window_type() with a type of either SCREEN_CHILD_WINDOW or SCREEN_EMBEDDED_WINDOW.

name

A unique string that identifies the group. This string must have been communicated down from the parent window.

Library:

libscreen

Description:

Function Type: *Delayed Execution* (p. 182)

This function is used to add a window to a group. Child and embedded windows will remain invisible until they're properly parented.

Until the window joins a group, a window of any type behaves like an application window. The window's positioning and visibility are not relative to any other window on the display. In order to join a group parented by an application window, a window must have a type of SCREEN_CHILD_WINDOW or SCREEN_EMBEDDED_WINDOW. Windows with a type of SCREEN_EMBEDDED_WINDOW can join only groups parented by windows of type SCREEN_CHILD_WINDOW.

Once a window successfully joins a group, its position on the screen will be relative to the parent. The type of the window determines exactly how the window will be positioned. Child windows are positioned relative to their parent (i.e., their window position is added to the parent's window position. Embedded windows are positioned relative to the source viewport of the parent.

Windows in a group inherit the visibility and the global transparency of their parent.

Returns:

`0` if the request for the window joining the specified group was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_leave_window_group()

Cause a window to leave a window group.

Synopsis:

```
#include <screen/screen.h>

int screen_leave_window_group(screen_window_t win)
```

Arguments:

win

The handle for the window that is to leave the group. This window must have been created with screen_create_window_type() with a type of either `SCREEN_CHILD_WINDOW` or `SCREEN_EMBEDDED_WINDOW`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function removes a window from a window group.

Returns:

`0` if the request for the window leaving its group was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_post_window()

Make window content updates visible on the display.

Synopsis:

```
#include <screen/screen.h>
```

```
int screen_post_window(screen_window_t win,
                       screen_buffer_t buf,
                       int count,
                       const int *dirty_rects,
                       int flags)
```

Arguments:

win

The handle for the window whose content has changed.

buf

The rendering buffer of the window that contains the changes needed to be made visible. Most applications use only two buffers for rendering. Therefore, the simplest way is to use the buffer in the first element of the SCREEN_PROPERTY_RENDER_BUFFERS property of the window. Screen rotates the buffer handles accordingly so that the first buffer handle of SCREEN_PROPERTY_RENDER_BUFFERS is one that is available.

count

The number of rectangles provided in the `dirty_rects` argument.

dirty_rects

An array of integers containing the x1, y1, x2, and y2 coordinates of a rectangle that bounds the area of the rendering buffer that has changed since the last posting of the window. The `dirty_rects` argument must provide at least `count` * 4 integers.

flags

A bitmask that can be used to alter the default posting behaviour. Valid flags are of type Screen_Flushing_Types.

Library:

`libscreen`

Description:

Function Type: *Apply Execution* (p. 182)

This function makes some pixels in a rendering buffer visible. The pixels to be posted are defined by the dirty rectangles contained in the `dirty_rects` argument. Note

that a window will not be made visible until screen_post_window() has been called at least once.

In addition to the area(s) defined by `dirty_rects`, Screen may update the other pixels in the buffer (i.e., Screen posts entire buffers, using `dirty_rects` as a guide). Screen may also retrieve data from the buffer at times other than when screen_post_window() is called (e.g., when the contents or properties of overlapping windows are updated, or when the window's entire buffer is continuously read by the display hardware until another buffer is posted to replace it). Therefore, your application must ensure that the entire contents of a window buffer is suitable for display at all times until a new buffer is posted to the window and the content of the display has been updated from the new buffer.

screen_post_window() returns immediately if render buffers are available and if SCREEN_WAIT_IDLE is not set. The use of multiple threads or application buffer management schemes to render at the full display frame rate are not necessary because unlike equivalent calls in other graphics systems, screen_post_window() does not always block.

If SCREEN_WAIT_IDLE is set in the flags, the function will return only when the contents of the display have been updated.

This function may cause the SCREEN_PROPERTY_RENDER_BUFFERS property of the posting window to change. At any time, only one thread must operate on, or render, into this window. If your application uses multiple threads, you must ensure that access to this window's handle by these threads is guarded. If not, SCREEN_PROPERTY_RENDER_BUFFERS may reflect out-of-date information that can lead to animation artifacts. The presentation of new content may result in a copy or a buffer flip, depending on how the composited windowing system chooses to perform the operation. Use the window property SCREEN_PROPERTY_RENDER_BUFFER_COUNT to determine the number of buffers you have that are available for rendering.

If the window is currently locked, posting updates has the effect of flushing all pending property changes and blocks until all other locked windows have released the lock or posted updates of their own. In this case, the window remains locked when screen_post_window() returns, and any subsequent property change is delayed until the window lock is released or another frame is posted.

If count is `0`, the buffer is discarded and a new set of rendering buffers is returned. The current front buffer remains unchanged and the contents of the screen will not be updated.

Returns:

`0` if the area of the rendering buffer that is marked dirty has updated on the screen and a new set of rendering buffers was returned (this new set of buffers can be used for the next updates), or `-1` if an error occurred (`errno` is set; refer to `/usr/in`

clude/errno.h for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_read_window()

Take a screenshot of the window and stores the resulting image in the specified buffer.

Synopsis:

```
#include <screen/screen.h>

int screen_read_window(screen_window_t win,
                       screen_buffer_t buf,
                       int count,
                       const int *save_rects,
                       int flags)
```

Arguments:

win

The handle of the window that is the target of the screenshot.

buf

The buffer where the pixel data will be copied to.

count

The number of rectables supplied in the `read_rects` argument.

save_rects

A pointer to (`count` * 4) integers that define the areas of the window that need to be grabbed for the screenshot.

flags

The mutex flags; must be set to 0.

Library:

`libscreen`

Description:

Function Type: *Apply Execution* (p. 182)

This function takes a screenshot of a window and stores the result in a user-provided buffer. The buffer can be a pixmap buffer or a window buffer. The buffer must have been created with the usage flag SCREEN_USAGE_NATIVE in order for the operation to succeed. The call blocks until the operation is completed. If `count` is 0 and `read_rects` is NULL, the entire window is grabbed. Otherwise, `read_rects` must point to `count` * 4 integers defining rectangles in screen coordinates that need to be grabbed. Note that the buffer size does not have to match the window size. Scaling will be applied to make the screenshot fit into the buffer provided.

Returns:

`0` if the operation was successful and the pixels are written to `buf`, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_ref_window()

Create a reference to a window.

Synopsis:

```
#include <screen/screen.h>

int screen_ref_window(screen_window_t win)
```

Arguments:

win

The handle of the window for which the reference is to be created.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function creates a reference to a window. This function can be used by window managers and group parents to prevent a window from disappearing, even when the process that originally created the window terminates abnormally. If this happens, ownership of the window is transferred to the window manager or group parent. The restrictions imposed on buffers still exist. The contents of the buffers can't be changed. The buffers cannot be destroyed until the window is unreferenced. When the original

process owner is no longer a client of the windowing system, the window will be destroyed when screen_destroy_window() is called by the reference owner.

Returns:

`0` if a reference to the specified window was created, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_window_property_cv()

Set the value of the specified window property of type char.

Synopsis:

```
#include <screen/screen.h>

int screen_set_window_property_cv(screen_window_t win,
                                  int pname,
                                  int len,
                                  const char *param)
```

Arguments:

win

handle of the window whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

len

The maximum number of bytes that can be read from `param`.

param

A pointer to a buffer containing the new value(s). This buffer must be an array of type `char` with a maximum length of `len`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a window property from a user-provided buffer. You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_CLASS`
- `SCREEN_PROPERTY_ID_STRING`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_window_property_iv()

Set the value of the specified window property of type integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_window_property_iv(screen_window_t win,
                                  int pname,
                                  const int *param)
```

Arguments:

win

handle of the window whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `int`. `param` may be a single integer or an array of integers depending on the property being set.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a window property from a user-provided buffer. You can use this function to set the value of the following properties:

- SCREEN_PROPERTY_SCALE_FACTOR
- SCREEN_PROPERTY_ALPHA_MODE
- SCREEN_PROPERTY_BRIGHTNESS
- SCREEN_PROPERTY_CBABC_MODE
- SCREEN_PROPERTY_COLOR
- SCREEN_PROPERTY_COLOR_SPACE
- SCREEN_PROPERTY_CONTRAST
- SCREEN_PROPERTY_DEBUG
- SCREEN_PROPERTY_FLIP
- SCREEN_PROPERTY_FLOATING
- SCREEN_PROPERTY_GLOBAL_ALPHA
- SCREEN_PROPERTY_HUE
- SCREEN_PROPERTY_IDLE_MODE
- SCREEN_PROPERTY_MIRROR
- SCREEN_PROPERTY_PIPELINE
- SCREEN_PROPERTY_PROTECTION_ENABLE
- SCREEN_PROPERTY_ROTATION
- SCREEN_PROPERTY_SATURATION
- SCREEN_PROPERTY_SCALE_QUALITY
- SCREEN_PROPERTY_SELF_LAYOUT
- SCREEN_PROPERTY_SENSITIVITY
- SCREEN_PROPERTY_STATIC
- SCREEN_PROPERTY_SWAP_INTERVAL
- SCREEN_PROPERTY_TRANSPARENCY
- SCREEN_PROPERTY_VISIBLE
- SCREEN_PROPERTY_ZORDER
- SCREEN_PROPERTY_BUFFER_SIZE
- SCREEN_PROPERTY_FORMAT
- SCREEN_PROPERTY_USAGE
- SCREEN_PROPERTY_CLIP_POSITION
- SCREEN_PROPERTY_CLIP_SIZE
- SCREEN_PROPERTY_POSITION
- SCREEN_PROPERTY_SIZE
- SCREEN_PROPERTY_SOURCE_CLIP_POSITION

- SCREEN_PROPERTY_SOURCE_CLIP_SIZE
- SCREEN_PROPERTY_SOURCE_POSITION
- SCREEN_PROPERTY_SOURCE_SIZE
- SCREEN_PROPERTY_VIEWPORT_POSITION
- SCREEN_PROPERTY_VIEWPORT_SIZE
- SCREEN_PROPERTY_REFERENCE_COLOR

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_window_property_llv()

Set the value of the specified window property of type long long integer.

Synopsis:

```
#include <screen/screen.h>

int screen_set_window_property_llv(screen_window_t win,
                                   int pname,
                                   const long long *param)
```

Arguments:

win

handle of the window whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `long long`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a window property from a user-provided buffer. You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_TIMESTAMP`:
  - Note that when the specified value for this property is `NULL`, screen automatically calcuates and sets this property to the current time. Screen uses the realtime clock and not the monotonic clock when calculating the timestamp.

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_set_window_property_pv()

Set the value of the specified window property of type void.*

Synopsis:

```
#include <screen/screen.h>

int screen_set_window_property_pv(screen_window_t win,
                                  int pname,
                                  void **param)
```

Arguments:

win

handle of the window whose property is being set.

pname

The name of the property whose value is being set. The properties that you can set are of type *Screen property types* (p. 200).

param

A pointer to a buffer containing the new value(s). This buffer must be of type `void*`.

Library:

`libscreen`

Description:

Function Type: *Delayed Execution* (p. 182)

This function sets the value of a window property from a user-provided buffer. You can use this function to set the value of the following properties:

- `SCREEN_PROPERTY_ALTERNATE_WINDOW`
- `SCREEN_PROPERTY_DISPLAY`
- `SCREEN_PROPERTY_USER_HANDLE`
- `SCREEN_PROPERTY_BRUSH`

Returns:

`0` if the command to set the new property value(s) was queued, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_share_window_buffers()

Cause a window to share buffers which have been created for or attached to another window.

Synopsis:

```
#include <screen/screen.h>

int screen_share_window_buffers(screen_window_t win,
                                screen_window_t share)
```

Arguments:

win

The handle of the window that will be sharing the buffer(s) owned by another window.

share

The handle of the window whose buffer(s) is to be shared.

Library:

`libscreen`

Description:

Function Type: *Flushing Execution* (p. 183)

This function is used when a window needs to share the same buffers created for, or attached to, another window. For this operation to be successful, the window that is the owner of the buffer(s) to be shared must have at least one buffer that was created with screen_create_window_buffers() or attached with `screen_attach_window_buffer()`. Buffers cannot be created or attached to any window that is sharing the buffers owned by another window. Updates can only be posted using the window that is the owner of the buffers (i.e. the window whose handle is identified as `share`). Any window that is sharing buffers with another window is orphaned from the buffers and made invisible when the window who owns the buffer(s) is destroyed. At this time, that status of each orphaned window is such that a new buffer can be created for it, or screen_share_window_buffers() can be called again. You can use the screen_share_window_buffers() function to improve performance by reducing the amount of blending on the screen. For example, a window might be entirely transparent except for a watermark that needs to be blended in a corner. Blending the entire window is costly and can be avoided by setting the transparency of this window to `SCREEN_TRANSPARENCY_DISCARD`. To keep the watermark visible, another window can be created and made to share buffers with the main window. This way, most of the window is discarded and a much smaller area is actually blended. Any window property, such as `SCREEN_PROPERTY_FORMAT`, `SCREEN_PROPERTY_USAGE`, and `SCREEN_PROPERTY_BUFFER_SIZE`, which was set prior to calling screen_share_window_buffers(), is ignored and reset to the values of the parent window.

Returns:

`0` if the windows are sharing buffers, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details). Note that the error may also have been caused by any delayed execution function that's just been flushed.

screen_unref_window()

Remove a reference from a specified window.

Synopsis:

```
#include <screen/screen.h>

int screen_unref_window(screen_window_t win)
```

Arguments:

win

The handle of the window for which the reference is to be removed.

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This function removes a reference to a window. When a window is being referenced, its buffers cannot be destroyed until all references to that window have been removed.

Returns:

`0` if a reference to the specified window was removed, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_wait_post()

Add a wait for a post on a window.

Synopsis:

```
#include <screen/screen.h>

int screen_wait_post(screen_window_t win,
                     int flags)
```

Arguments:

win

The handle for the window whose post you are waiting on.

flags

A bitmask that can be used to alter the default posting behaviour. Valid flags are of type *Screen flushing types* (p. 193).

Library:

`libscreen`

Description:

Function Type: *Immediate Execution* (p. 183)

This call blocks until there is a post event for the window you are waiting on. This function is typically used in conjunction with screen_share_display_buffers() and/or screen_share_window_buffers().

Returns:

`0` if a wait for a post on the specified window was added, or `-1` if an error occurred (`errno` is set; refer to `/usr/include/errno.h` for more details).

screen_window_t

A handle for the screen window.

Synopsis:

```
#include <screen/screen.h>

typedef struct _screen_window* screen_window_t;
```

Library:

`libscreen`

Description:

This handle is used to identify the window that you are performing actions on. Such actions could include:

- querying or setting properties
- posting
- sharing buffers