Application and Window Management

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: March 31, 2015

Contents

About This Guide.....5

Typographical conventions.....6

Technical support.....8

Chapter 1: Packaging, Installing, and Launching Apps.....9

Packaging a native C/C++ app for installation.....10

Installing a packaged app on the target.....11

Launching an app on the target.....12

Stopping all apps on the target.....13

Uninstalling Apps.....14

Sample `bar-descriptor.xml` files.....15

Chapter 2: Creating Your Own Application Window Manager.....19

Application management.....20

Interacting with the `/pps/services/launcher/control` object.....20

Starting an application 23

Stopping an application.....24

Window management.....25

Set up Screen 25

Handle Screen events.....28

An example of a simple application window manager.....32

`struct.h`.....33

`main.c`.....36

`screen.c`.....38

`launcher.c`.....58

`pps.c`.....60

`core.c`.....65

Index.....67

About This Guide

Application and Window Management describes the process of starting and stopping applications. This guide also explains how windows interact with the HMI and how to write your own window manager.

This guide is intended for developers who will be creating and deploying apps for embedded devices running QNX Neutrino.

The following table may help you find information quickly:

To find out about:	Go to:
Creating an archive for a native C/C++ app	*Packaging a native C/C++ app for installation* (p. 10)
Installing an app on your target	*Installing a packaged app on the target* (p. 11)
Launching an app	*Launching an app on the target* (p. 12)
Stopping apps	*Stopping all apps on the target* (p. 13)
Removing apps from your target	*Uninstalling Apps* (p. 14)
Managing the life cycle and appearance of your applications	*Creating Your Own Application Window Manager* (p. 19)

For instructions on how to *create* apps, see the following:

- *HTML5 Developer's Guide*
- *Qt Developer's Guide*
- *IDE User's Guide*

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.

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

WARNING: 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
Packaging, Installing, and Launching Apps

Before you can install an app on your target, you must first package it.

Packaging is the process of creating a compressed archive of all the files that comprise your app. The tools you use will differ, depending on the type of app you're packaging.

For any type of app, you need to specify:

- the app's resources
- the services the app needs to access on the target

You then run the approprate packaging tool, which produces the final BlackBerry ARchive (BAR) file (with a **.bar** extension) that you'll install on your target. Note that the format of a BAR file is essentially RAR, so you can use an extractor tool such as WinRAR to view the contents of any **.bar** file.

When your app is packaged as a **.bar** file, you copy it to your target, where you can install and launch it.

Packaging a native C/C++ app for installation

To create a **.bar** file for a native C/C++ app, you run the `blackberry-nativepackager` utility.

The `blackberry-nativepackager` utility creates a **.bar** file that includes all of your app's resources as well as the following key components:

- an app descriptor file (**bar-descriptor.xml**), which specifies capabilities in `<action>` elements. You should list the assets required by your app in this file. See "*Sample bar-descriptor.xml files* (p. 15)" as well as the *HTML5 Developer's Guide* and the *Qt Developer's Guide*.
- a launch icon (***icon*.png**)

To package your app:

1. Open a command prompt.
2. Navigate to the directory where your native project is stored.
3. Run the following command:

   ```
   blackberry-nativepackager -package -devMode bar-package [app-desc files]
   ```

   where:

   - *bar-package* is the path to the location where you want the BAR file to be created
   - *app-desc* is the path to the app descriptor file (**bar-descriptor.xml**)
   - *files* is a sequence of paths to files or directories to include in the package

   These paths can be absolute or relative to a current directory.

   You run `blackberry-nativepackager` in developer mode because you don't need to sign your app. For information on other parameters, run the command:

   ```
   blackberry-nativepackager -help
   ```

The **.bar** file is created at the path you specified. You can now install and run the app on the target.

Installing a packaged app on the target

After you've packaged your app (by running the packager to create a **.bar** file), you're ready to install the app on your target system.

To install your app:

1. Copy your app's **.bar** file from your host machine to your target.

   You could use FTP or a tool such as *WinSCP* (Windows) for copying files. Or you could use a USB stick, which should appear on your target as a directory called **/fs/usb0/**.

2. From a terminal window, execute the **bar-install** script:

   ```
   /scripts/bar-install my_app.bar
   ```

   Each installed app resides in its own directory under **/apps** on your system. The directory name looks something like this:

   ```
   package_name.testDev_system_generated_suffix/
   ```

   where *system_generated_suffix* is a combination of the last few characters of the package name and a generated identifier.

   For example, here's the directory name for the default HTML5 app created with the package name **com.example.hello**:

   ```
   com.example.hello.testDev_ample_hellodf4765a1/
   ```

You can now launch the app.

Launching an app on the target

After you've installed your app's **.bar** file, you can launch your app.

If you don't have an HMI running on your target, you can still run your app from the command line.

To launch your app:

- Run the **launch** script from the command line to start your app:

  ```
  launch  project_name
  ```

  For example, `launch BrowserLite`.

The **launch** script creates all the files and folders the app needs in the application sandbox, and then runs the app on the target. You can see the files in the sandbox by looking at the app data directory located at **/accounts/1000/appdata** on your target image.

To launch the app, the Application Launcher service echoes a command containing the app's Application ID to the appropriate Persistent Publish/Subscriber (PPS) object:

```
echo "msg::start\ndat::application_id\nid::id_number" >
 /pps/services/launcher/control
```

Here, *application_id* is the app's name in the **/apps** directory and *id_number* is an arbitary string that identifies the message. For example, if the BrowserLite sample application was started, the *application_id* would be `BrowserLite.testDev_BrowserLite353323d6` (you can determine this from the **/apps** folder) and an arbitary value identifying the message such as `1234` would be also sent. The full message would be:

```
echo "msg::start\ndat::BrowserLite.testDev_BrowserLite353323d6\n
 id::1234" > /pps/services/launcher/control
```

For information about Application Launcher and the PPS control object it uses, see `launcher` in the *System Services Reference* and **/pps/services/launcher/control** in the *PPS Objects Reference*.

Stopping all apps on the target

For testing purposes or if your system doesn't have an HMI, you can stop applications from the command line using the `stop-apps` script.

Under normal circumstances, however, your HMI stops applications by issuing a stop message to PPS (see "*Stopping an application* (p. 24)").

To stop all running apps on your system:

- Run the following from the command line:

  ```
  stop-apps
  ```

The **stop-apps** script echoes this stop command to the PPS object:

```
"msg::stop\ndat::" > /pps/services/launcher/control
```

The Application Launcher service shuts down all the running apps.

For information about Application Launcher and the PPS control object it uses, see `launcher` in the *System Services Reference* and **/pps/services/launcher/control** in the *PPS Objects Reference*.

Uninstalling Apps

You can use the **bar-uninstall** script to uninstall apps.

To uninstall an app:

- Run the following from the command line:

  ```
  bar-uninstall my_app_id
  ```

  where *my_app_id* is the ID of your app as given in the `<id>` element in your **bar-descriptor.xml** file (e.g., `<id>com.example.hello</id>`).

  You can run `bar-uninstall` without any arguments to see a list of all apps currently installed on your system.

Sample bar-descriptor.xml files

Here are the **bar-descriptor.xml** files from several apps. For information about the elements in the files, see *The application descriptor file DTD* in the BlackBerry 10 documentation.

HelloWorld: an HTML5 app

```
<?xml version='1.0' encoding='utf-8'?>
<qnx>
    <id>helloworld</id>
    <versionNumber>1.0.0</versionNumber>
    <author>QNX</author>
    <asset entry="true" type="qnx/elf">wwe</asset>
    <asset>README.txt</asset>
    <asset>config.xml</asset>
    <asset>cordova.js</asset>
    <asset>default-icon.png</asset>
    <asset>index.html</asset>
    <asset>wwe</asset>
    <asset>chrome/frameworkModules.js</asset>
    <asset>chrome/index.html</asset>
    <asset>chrome/require.js</asset>
    <asset>chrome/ui.html</asset>
    <asset>chrome/lib/exception.js</asset>
    <asset>chrome/lib/utils.js</asset>
    <asset>plugins/jnext/auth.txt</asset>
    <entryPointType>Qnx/WebKit</entryPointType>
    <cascadesTheme>default</cascadesTheme>
    <initialWindow>
        <systemChrome>none</systemChrome>
        <transparent>true</transparent>
        <autoOrients>true</autoOrients>
    </initialWindow>
    <env value="2.0.0" var="WEBWORKS_VERSION" />
    <env value="slog2" var="CONSOLE_MODE" />
    <permission system="true">run_native</permission>
    <permission system="false">access_internet</permission>
    <permission>access_shared</permission>
    <permission>access_internet</permission>
    <permission>run_native</permission>
    <name>HelloWorld</name>
    <description>Cordova Hello World</description>
    <icon>
        <image>default-icon.png</image>
    </icon>
    <splashScreens />
    <buildId>0</buildId>
</qnx>
```

RearviewCamera: a native app

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<qnx xmlns="http://www.qnx.com/schemas/application/1.0">
    <id>rearview_camera</id>
    <name>Camera</name>
    <versionNumber>1.0.0</versionNumber>
    <buildId>1</buildId>
    <description>Camera</description>
    <author>QNX</author>
    <initialWindow>
        <autoOrients>true</autoOrients>
        <systemChrome>none</systemChrome>
        <transparent>false</transparent>
    </initialWindow>
    <category>vehicle</category>
    <configuration name="arm">
    <asset path="display_black" entry="true" type="Qnx/Elf">
        nto/arm/o.le.v7/display_black
    </asset>
    </configuration>
    <configuration name="x86">
    <asset path="display_black" entry="true" type="Qnx/Elf">
        nto/x86/o/display_black
    </asset>
    </configuration>
    <icon>
       <image>resources/icon.png</image>
    </icon>
    <permission system="true">run_native</permission>
    <permission system="true">configure_system</permission>
</qnx>
```

IPCamera: a QT app

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<qnx xmlns="http://www.qnx.com/schemas/application/1.0">
    <id>com.example.ipcamera</id>
    <name>IP Camera</name>
    <versionNumber>1.0.0</versionNumber>
    <description>DESCRIPTION</description>
    <initialWindow>
        <systemChrome>none</systemChrome>
        <transparent>false</transparent>
        <autoOrients>true</autoOrients>
        <aspectRatio>landscape</aspectRatio>
    </initialWindow>
    <publisher>PUBLISHER</publisher>
    <copyright>COPYRIGHT</copyright>
    <env var="QQNX_PHYSICAL_SCREEN_SIZE" value="150,90"/>
    <icon><image>icon.png</image></icon>
    <action system="true">run_native</action>
    <permission>access_protected_media</permission>
    <permission>access_shared</permission>
    <!-- should point to the project binary, path can be relative -->
    <asset entry="true" path="IPCamera" type="Qnx/Elf">IPCamera</asset>
</qnx>
```

Chapter 2
Creating Your Own Application Window Manager

You can create your own application window manager to support an HMI built with a standard UI technology such as HTML5, Qt, or OpenGL ES.

What is an application window manager?

An application window manager is responsible for managing:

- interactions with the HMI
- the placement and appearance of application windows
- the starting and stopping of applications

Your window manager can support an HMI that was developed using any one of the industry-standard UI technologies (e.g., HTML5, Qt, OpenGL ES).

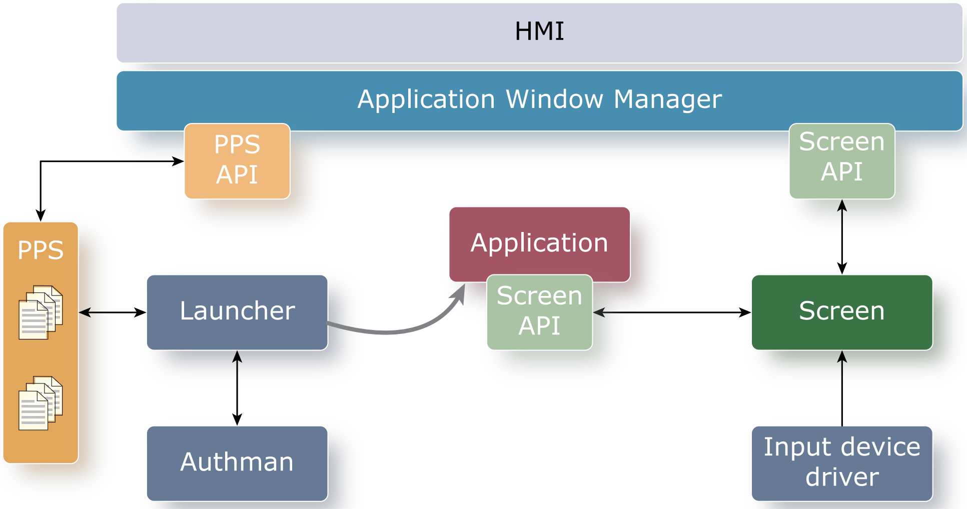

Figure 1: Overview of how window manager interacts with other components to launch an application

Application management

Managing the application life cycle (starting and stopping applications) is achieved by communicating with the **launcher** service through the Persistent Publish/Subscribe (PPS) service.

Your window manager implementation must, at a minimum, do the following:

- have access to the list of applications to be managed
- publish to the **/pps/services/launcher/control** object
- subscribe to the **/pps/services/launcher/control** object
- handle state changes appropriately

Your window manager needs to access the application ID and any data associated with each application. This application information is required to interact with the PPS control object for starting and stopping applications.

Application information can be retrieved from various means and depends on the design of your window manager. You can use anything from a simple configuration file to setting up your own PPS objects to track application information.

The examples in this guide illustrate the system calls required to open the **launcher** control object, read PPS messages to react to the object's creation or deletion or to changes in its attributes, and write commands to start and stop applications.

For information about Application Launcher and the PPS control object it uses, see `launcher` in the *System Services Reference* and **/pps/services/launcher/control** in the *PPS Objects Reference*.

Interacting with the `/pps/services/launcher/control` object

Your window manager interacts with the **/pps/services/launcher/control** object to start and stop applications.

To manage the interactions with this PPS object, you need to:

1. Create a thread to handle object interactions.
2. Open the object for publishing and subscribing.
3. Handle messages.

Create a thread to handle object interactions

You can create a thread in your window manager to manage your interactions with the PPS object:

```
window_manager_t window_manager;
memset(&window_manager, 0, sizeof(window_manager_t));
window_manager_t *winmgr = &window_manager;
...
int  pps_tid;                               /* PPS thread ID */
void *pps_thread(void *arg);
if (rc = pthread_create(NULL, NULL, pps_thread, (void*)win_mgr) < 0)
```

```
{ /* error-handling code goes here */ }
pthread_setname_np(screen_tid = pthread_self(), "pps");
```

Open the object for publishing and subscribing

Simply call *open()* with `O_RDWR` from your PPS thread to publish and subscribe to the **/pps/services/launcher/control** object.

Make sure to open the object in **delta** mode and **wait** mode. With **delta** mode, a subscriber receives only the changes to the object attributes. The **wait** mode means that any *read()* calls to the object block until the object changes or a delta appears.

```
int pps_fd;
pps_fd = open("/pps/services/launcher/control?wait,delta", O_RDWR);
```

Handle messages

In your window manager's PPS thread, parse the PPS message received and then handle each message type accordingly. Refer to “PPS API reference” in the *Persistent Publish/Subscribe Developer's Guide* for PPS API constants and functions.

Here's an example of how you could implement handling PPS messages:

```
char buf[1024];
int nread = -1;
while(1){
...
while (nread == -1){
...
    nread = read(fd, buf, sizeof(buf)-1);
    if (nread > 1){
        buf[nread] = '\0';
        /* Declare variables to store PPS API attribute and status*/
        pps_attrib_t    info;
        pps_status_t    rc;
        /* Call PPS API function ppsparse() to parse
         * the message received
         */
        while((rc = ppsparse(&buf, NULL, NULL, &info, 0)) != PPS_END) {
            /* Handle each PPS message type accordingly */
            switch(rc) {
            case PPS_OBJECT_CREATED:
                win_mgr->objname = info.obj_name;
                win_mgr->ptype = PPS_EVENT_OBJECT_CREATED;
            break;

            case PPS_OBJECT_TRUNCATED:
            case PPS_OBJECT_DELETED:
```

```
            nav->objname = info.obj_name;
            nav->ptype = PPS_EVENT_OBJECT_DELETED;
            break;

            case PPS_OBJECT:
            if(info.obj_name[0] != '@') {
                exit;
            }
            win_mgr->objname = info.obj_name;
            win_mgr->ptype = PPS_EVENT_OBJECT_CHANGED;
            break;

            case PPS_ATTRIBUTE_DELETED: {
            --info.attr_name;
            /* Handle when there is an attributed deleted */
            int err = pps_parse_attr(nav, &info);
            if(err != EOK)
                return;
            }

            case PPS_ATTRIBUTE: {
            /* Handle when there is an attributed updated */
            int err = pps_parse_attr(nav, &info);
            if(err != EOK)
                return;
            break;
            }

            case PPS_ERROR:
            default:
                SLOG_WARNING("We got a parsing error.");
                return;
            }
        }
        /* Update any necessary information such as appending
         * or deleting from your application list based on the
         * PPS message received and parsed. In this case, we
         * are calling a helper function, launcher_pps() to
         * perform any updates to the window manager.
         */
        launcher_pps(win_mgr);
    }
}
}
```

Starting an application

To start an application from your window manager, you need to issue a *write()* call to send a `start` command to the PPS launcher control object.

Publishing a `start` command to the PPS object

After opening the **/pps/services/launcher/control** object for publishing, call *write()* to modify the object's attributes. Use the appropriate message format with the `start` command.

```
#define CMD_START        "start"
...
int msgsize;
char msgbuf[4096];
int ret = 0;
char *id = strdup("101");
char *data = strdup("HelloWebWorks.testDev_lloWebWorks1fa80f60");
...
msgsize = snprintf(msgbuf,
                   sizeof(msgbuf),
                   "msg::%s\ndat::%s\nid::%s", CMD_START, data, id);
ret = write(pps_fd, msgbuf, (unsigned)msgsize);
```

The figure below shows the basic steps for launching an application:

1. The device driver writes an event letting Screen know that someone has tried to launch an application.
2. Window manager learns about the event through the Screen API.
3. Window manager publishes to a PPS object so that interested components can know about the request to launch an application.
4. The launcher reads the PPS object and begins launch procedures.
5. The launcher asks the authorization manager to check permissions to launch the application.
6. When it receives authorization, the launcher completes the application launch.
7. The application uses the Screen API to tell Screen that it is present and ready to be displayed.

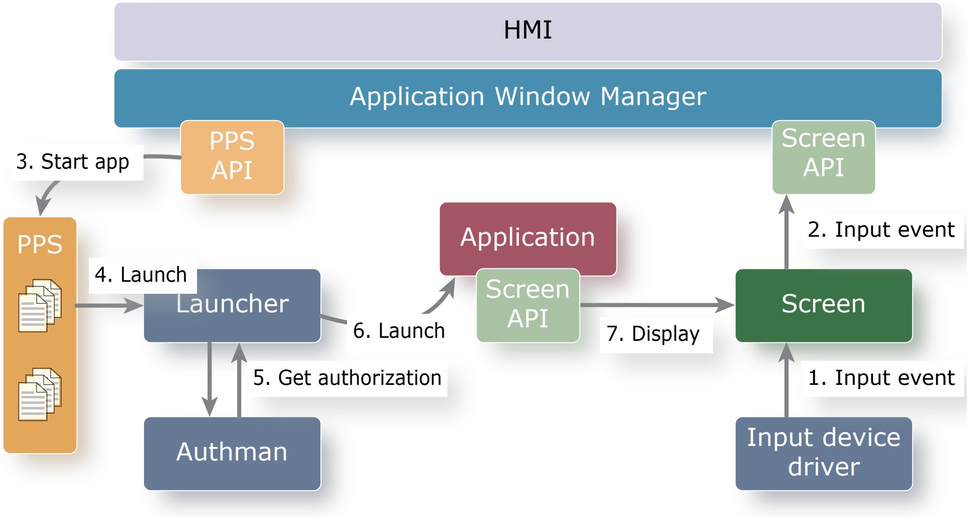

Figure 2: Step-by-step view of how window manager launches an application

For information about Application Launcher and the PPS control object it uses, see `launcher` in the *System Services Reference* and **/pps/services/launcher/control** in the *PPS Objects Reference*.

Stopping an application

To stop an application from your window manager, you need to issue a *write()* call to send a `stop` command to the PPS launcher control object.

Publish `stop` command to the PPS object

After opening the **/pps/services/launcher/control** object for publishing, call *write()* to modify the object's attributes. Use the appropriate message format with the `stop` command.

```
#define CMD_START        "stop"
...
int msgsize;
char msgbuf[4096];
int ret = 0;
char *id = strdup("101");
char *data = strdup("HelloWebWorks.testDev_lloWebWorks1fa80f60");
...
msgsize = snprintf(msgbuf,
                   sizeof(msgbuf),
                   "msg::%s\ndat::%s\nid::%s", CMD_STOP, data, id);
ret = write(pps_fd, msgbuf, (unsigned)msgsize);
```

For information about Application Launcher and the PPS control object it uses, see `launcher` in the *System Services Reference* and **/pps/services/launcher/control** in the *PPS Objects Reference*.

Window management

Your window manager is responsible for the placement and appearance of application windows.

Managing application windows is achieved through Screen. Through Screen API functions, your window manager deals with the application windows' z-order, transparency, positioning on the physical display, and scaling. For more information on Screen, see the *Screen Graphics Subsystem Developer's Guide*.

Once started, applications communicate directly with Screen from their own context. Applications manage their own windows through the Screen API.

Set up Screen

Window management requires a connection to Screen, configuration of a window, and creation of a window buffer.

Before you can manage application windows, you need to set up your window manager by doing the following:

1. Connect to Screen.
2. Create a window for your window manager.
3. Set the properties for your window.
4. Create a window buffer for your window.
5. Post your window and flush your context.

Connect to Screen

The first step is to establish a connection between your window manager and the underlying windowing system, Screen. To set up this connection, you need to create a Screen context.

There are different context types. A standard application would use SCREEN_APPLICATION_CONTEXT. Because you're writing a window manager, you need a context type that lets you modify all the windows in the system. Specifically, you need to use SCREEN_WINDOW_MANAGER_CONTEXT; this context type enables the receipt of events when application windows are created and destroyed and when applications change their window properties.

Note that **root** permission is required to use the SCREEN_WINDOW_MANAGER_CONTEXT context type.

```
int rc = 0;
screen_context_t screen_ctx; /* connection to Screen */
rc = screen_create_context(&screen_ctx, SCREEN_WINDOW_MANAGER_CONTEXT);
```

Create a window for your window manager

Without a window for your window manager, you can still receive SCREEN_EVENT_CREATE and SCREEN_EVENT_CLOSE window manager events, but you can't receive any *input events*.

You need to create a window for your window manager so that you can receive and handle these input events:

- SCREEN_EVENT_MTOUCH_TOUCH

- SCREEN_EVENT_MTOUCH_MOVE
- SCREEN_EVENT_MTOUCH_RELEASE
- SCREEN_EVENT_POINTER
- SCREEN_EVENT_KEYBOARD

```
screen_window_t screen_win;    /* native handle for our window */
rc = screen_create_window(&screen_win, screen_ctx);
```

Set the properties for your window

Although many window properties are available, you don't need to set them all because most have defaults that are appropriate. For a window manager, however, you need to set some particular window properties:

SCREEN_PROPERTY_USAGE

The intended usage for the buffers associated with the window. You need to ensure that these buffers can be written to, which means you need to set the SCREEN_USAGE_WRITE flag in the bitfield for this property.

SCREEN_PROPERTY_SIZE

The width and height, in pixels, of the window. By default, windows are fullscreen. You may not want your window manager's window to be fullscreen; for example, if you still want to see this window when you run multiple applications at the same time.

SCREEN_PROPERTY_POSITION

The window's display coordinates. You want to set this so that the position of your window manager's window isn't obscuring an application window's area of interest.

SCREEN_PROPERTY_ZORDER

This property indicates the level from the bottom, which is used to order window groups among each other. Your window manager needs to examine the z-order settings of its own window and of all application windows so that it can display them in the correct order.

```
int val = 0;
val = SCREEN_USAGE_WRITE;
rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &val);
...
int size[2] = { 64, 64 };      /* size of the window on screen */
rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SIZE, size);
...
int pos[2] = { 0, 0 };         /* position of the window on screen */
rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_POSITION, pos);
...
int zorder = 0;
rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_ZORDER, &zorder);
```

Create a window buffer for your window

You need at least one buffer to hold the contents of your window so that your window will be visible.

In the simplest case, you can fill your window with a solid color so that you can see the window. Before you can do this, you'll need to query some properties of the window buffer.

SCREEN_PROPERTY_RENDER_BUFFERS

The pointer to the window buffer available for rendering. It's best to first query SCREEN_PROPERTY_RENDER_BUFFER_COUNT to determine the number of window buffers you have. But in this case, there's only one, so you can simply query SCREEN_PROPERTY_RENDER_BUFFERS.

SCREEN_PROPERTY_POINTER

The pointer that can be used to read from and/or write to the window buffer. When you set the SCREEN_PROPERTY_USAGE to include SCREEN_USAGE_WRITE, you enable write access to this buffer. Therefore, this pointer will reference memory that you can write to.

SCREEN_PROPERTY_STRIDE

The size, in bytes, of each line of the window buffer. This value is the number of bytes between the same pixel on adjacent rows.

For the sake of simplicity, you can just fill the window buffer with a solid color pattern. To do this, you can use *memset()*:

```
screen_buffer_t screen_buf;    /* renderable buffer for the window */
rc = screen_create_window_buffers(screen_win, 1);
rc = screen_get_window_property_pv(screen_win,
                                   SCREEN_PROPERTY_RENDER_BUFFERS,
                                   (void **)&screen_buf);
rc = screen_get_buffer_property_pv(screen_buf, SCREEN_PROPERTY_POINTER, &pointer);
int stride;                    /* size of each window line in bytes */
memset(pointer, 0x80, stride * size[1]);
```

Post your window and flush your context

To make the content rendered on your window visible, you need to post your changes to Screen. Posting to Screen indicates that you have completed drawing to your render buffer and you wish to have the changes made visible. When you post, you need to specify which area of your buffer has changed so that Screen will redraw only the parts of the framebuffer that need updating. When posting your first frame, you must post the entire buffer.

To ensure that any delayed Screen commands are processed, flush the command queue of your context after you post:

```
int rect[4] = { 0, 0, size[0], size[1] };
rc = screen_post_window(screen_win, screen_buf, 1, rect, 0);
rc = screen_flush_context(screen_ctx, SCREEN_WAIT_IDLE);
```

Handle Screen events

The window manager needs to handle any events of interest.

Events that require action by your window manager include the creation and destruction of application windows as well as some input events.

To set up your window manager to handle Screen events, you need to:

1. Create a thread in your window manager.
2. Create a Screen event.
3. Handle any Screen event of interest to your window manager.

Create a thread in your window manager

You can create a thread in your window manager to handle Screen events.

```
window_manager_t window_manager;
memset(&window_manager, 0, sizeof(window_manager_t));
window_manager_t *winmgr = &window_manager;
...
int    screen_tid;                       /* Screen thread ID */
void *screen_thread(void *arg);
if (rc = pthread_create(NULL, NULL, pps_thread, (void*)win_mgr) < 0)
{ /* error-handling code goes here */ }
pthread_setname_np(screen_tid = pthread_self(), "screen_monitor");
```

Create a Screen event

Create a Screen event to store the event. After retrieving the event from the context's event queue, you can use Screen API functions to query the event's properties to determine whether additional action is required.

```
screen_event_t screen_ev;    /* handle used to retrieve events from our queue */
rc = screen_create_event(&screen_ev);
```

Handle any Screen event of interest to your window manager

Create an event-handling routine within your thread. This routine will retrieve the most recent event from the queue and then extract data from the event.

Use the Screen API function *screen_get_event()* to retrieve the event. Then, use the *screen_get_event_property_iv()* function to retrieve the event type, by querying the SCREEN_PROPERTY_TYPE property. Handle the events of interest to your window manager.

```
screen_window_t win;                 /* stores a window contained in an event */
int val;                             /* used for simple property queries */
screen_display_t *displays, disp;    /* used for display queries */
int display_count = 0, port;         /* used for display queries */
int pair[2];                         /* used to query pos, size */
```

```
void *ptr;                              /* used to query user handles */
char str[128];                          /* used to query string properties */
int size[2] = { 64, 64 };               /* size of the window on screen */

while (!screen_get_event(screen_ctx, screen_ev, ~0L)) {
    screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &val);
    switch (val) {
        case SCREEN_EVENT_DISPLAY:
            if ( screen_get_event_property_pv(screen_ev,
                                              SCREEN_PROPERTY_DISPLAY,
                                              (void *)&disp) == 0 ) {
            } else {
                break;
            }
            screen_get_display_property_iv(disp, SCREEN_PROPERTY_TYPE, &val);
            switch(val) {
               case SCREEN_DISPLAY_TYPE_HDMI:
                    screen_get_display_property_iv(disp, SCREEN_PROPERTY_ATTACHED, &val);
                    screen_get_display_property_iv(disp, SCREEN_PROPERTY_ID, &port);
                    break;
                default:
                    break;
            }
            break;
        case SCREEN_EVENT_IDLE:
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_IDLE_STATE, &val);
            screen_get_context_property_iv(screen_ctx, SCREEN_PROPERTY_IDLE_STATE, &val);
            screen_get_context_property_iv(screen_ctx,
                                           SCREEN_PROPERTY_DISPLAY_COUNT,
                                           &display_count);
            displays = malloc(display_count * sizeof(screen_display_t));
            screen_get_context_property_pv(screen_ctx,
                                           SCREEN_PROPERTY_DISPLAYS,
                                           (void *)displays);
            for (int i=0; i<display_count; i++) {
                screen_get_display_property_iv(displays[i], SCREEN_PROPERTY_KEEP_AWAKES, &val);
            }
            free(displays);
            break;
        case SCREEN_EVENT_CREATE:
            screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&win);
            screen_get_window_property_iv(win, SCREEN_PROPERTY_OWNER_PID, &val);
            screen_get_window_property_pv(win, SCREEN_PROPERTY_USER_HANDLE, &ptr);
            break;
        case SCREEN_EVENT_PROPERTY:
            screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&win);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_NAME, &val);
            break;
        case SCREEN_EVENT_CLOSE:
            screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&win);
            screen_get_window_property_pv(win, SCREEN_PROPERTY_USER_HANDLE, &ptr);
            screen_destroy_window(win);
            break;
        case SCREEN_EVENT_POST:
            screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&win);
            screen_get_window_property_pv(win, SCREEN_PROPERTY_USER_HANDLE, &ptr);
            set_window_properties(win);
            screen_flush_context(screen_ctx, 0);
            break;
        case SCREEN_EVENT_INPUT:
        case SCREEN_EVENT_JOG:
            break;
        case SCREEN_EVENT_POINTER:
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_DEVICE_INDEX, &val);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_POSITION, pair);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_BUTTONS, &val);
            if (val) {
                if (pair[0] >= size[0] - exit_area_size &&
                    pair[0] < size[0] &&
                    pair[1] >= 0 &&
                    pair[1] < exit_area_size) {
                    goto end;
                }
           }
           break;
        case SCREEN_EVENT_KEYBOARD:
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_DEVICE_INDEX, &val);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_CAP, &val);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_FLAGS, &val);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_MODIFIERS, &val);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_SCAN, &val);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_KEY_SYM, &val);
            switch (val) {
                case KEYCODE_ESCAPE:
                    goto end;
            }
            break;
        case SCREEN_EVENT_MTOUCH_TOUCH:
        case SCREEN_EVENT_MTOUCH_MOVE:
        case SCREEN_EVENT_MTOUCH_RELEASE:
            screen_get_event_property_pv(screen_ev, SCREEN_PROPERTY_WINDOW, (void **)&win);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TOUCH_ID, &val);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_SEQUENCE_ID, &val);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_POSITION, pair);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_SIZE, pair);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_SOURCE_POSITION, pair);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_SOURCE_SIZE, pair);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TOUCH_ORIENTATION, &val);
            screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TOUCH_PRESSURE, &val);
            break;
        case SCREEN_EVENT_USER:
            break;
    }
}
```

See the *Screen Graphics Subsystem Developer's Guide* for a complete list of all Screen event types.

An example of a simple application window manager

This example shows some essential application window manager functionality.

The following reference code implements a simple application window manager. There are several ways of designing a window manager; this example shows only the essential initialization and handling required. Your window manager implementation will likely involve more complicated handling of PPS and Screen events.

struct.h

Constants and function definitions for a simple window manager

```
/*
* $QNXLicenseC:
* Copyright 2012, 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.
* $
*/

#ifndef STRUCT_H_
#define STRUCT_H_

#include <errno.h>

#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 <time.h>           /* Header file for clock_gettime, timespec2nsec */
#include <screen/screen.h> /* Header file for all screen API calls */
#include <pthread.h>
#include <fcntl.h>

#include <sys/pps.h>

#include <sys/slog.h>
#include <sys/slogcodes.h>
```

```
#define WINMGR_SLOG_CODE _SLOG_SETCODE(_SLOGC_TEST,104)
#define SLOG_WARNING(...)  slogf(WINMGR_SLOG_CODE,_SLOG_WARNING, __VA_ARGS__)
#define SLOG_ERROR(...)    slogf(WINMGR_SLOG_CODE,_SLOG_ERROR, __VA_ARGS__)
#define SLOG_NOTICE(...)   slogf(WINMGR_SLOG_CODE,_SLOG_NOTICE, __VA_ARGS__)

#define KILO(n) ((n)*1024)
#define MEG(n)  ((n)*1024*1024)

#define MAX_REQSIZE KILO(32)
#define MAX_RESSIZE KILO(1)
#define MAX_ATTRS   KILO(1)

// launcher commands
#define CMD_START            "start"
#define CMD_DEBUG            "debug"
#define CMD_STOP             "stop"
#define CMD_FREEZE           "freeze"
#define CMD_THAW             "thaw"
#define CMD_LOWMEM           "lowmem"
#define CMD_STOPPED          "stopped"
#define CMD_ACTIVE           "active"
#define CMD_QUERY            "query"
#define CMD_HIDE             "hide"

enum {
    WINMGR_UPDATE = (1 << 0),
    WINMGR_TERMINATE = (1 << 1),
};

typedef enum {
    PPS_EVENT_OBJECT_UNKNOWN = 0x00,
    PPS_EVENT_OBJECT_CHANGED = 0x01,
    PPS_EVENT_OBJECT_CREATED = 0x02,
    PPS_EVENT_OBJECT_DELETED = 0x04,
    PPS_EVENT_ALL = 0x7,
    PPS_FLAG_CREDENTIALS = 1 << 15
} pps_event_type;

typedef struct {
    char    *id;
    char    *pid;
    char    *data;
} app_t;

typedef struct
{
    char            *name;
    char            *encoding;
    char            *value;
} pps_attr_t;

typedef struct {
    int                 state;
    int                 pps_fd;
    app_t               car_app;
    app_t               weather_app;
    int                 verbose;
    int                 background;

    // pps related
    int                 numattrs;
    char                *objname;
    pps_event_type      ptype;
    pps_attr_t          attrs[MAX_ATTRS];
    int                 pps_tid;

    // screen related
    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 */
    int                 screen_tid;
} window_manager_t;

// pps.c
extern void* pps_thread(void* arg);
int pps_write(int pps_fd, const char *msgbuf, int msgsize);
int pps_is_open(int pps_fd);
char* pps_lookup(window_manager_t *winmgr, char *name);
pps_attr_t* pps_lookup_attr(window_manager_t *winmgr, char *name);

// launcher.c
int launcher_pps(window_manager_t *winmgr);
void launcher_send(window_manager_t *winmgr, char *cmd, char *data, char *id);

// core.c
void core_app_started(window_manager_t *winmgr, char *id, char *data, int error, char *errstr);
void core_app_stopped(window_manager_t *winmgr, char *data);
void core_lowmem(window_manager_t *winmgr, char *data);

// screen.c
int screen_init(window_manager_t *winmgr, int argc, char **argv);
void* screen_thread(void *arg);

#endif /* STRUCT_H_ */
```

main.c

The main application for a simple window manager

```
/*
 * $QNXLicenseC:
 * Copyright 2012, 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 "struct.h"

static void main_setup_default(window_manager_t *winmgr)
{
    winmgr->background = 1;
    winmgr->pps_fd = -1;
    winmgr->state = WINMGR_UPDATE;
    winmgr->verbose = 1;

    winmgr->car_app.id = strdup("100");
    winmgr->car_app.data = strdup("carcontrol.testDev_carcontrol_21522f09,\
                               WIDTH=800,HEIGHT=395");

    winmgr->weather_app.id = strdup("101");
    winmgr->weather_app.data = strdup("sys.browser.gYABgJYFHAzbeFMPCCpYWBtHAm0,\
                                   WIDTH=800,HEIGHT=395");

    winmgr->screen_tid = -1;
}
```

```
int main(int argc, char **argv)
{
    int rc;

    window_manager_t window_manager;
    memset(&window_manager, 0, sizeof(window_manager_t));
    window_manager_t *winmgr = &window_manager;
    main_setup_default(winmgr);

    rc = screen_init(winmgr, argc, argv);
    if (rc != EOK) {
        exit(EXIT_FAILURE);
    }

    // create a pps thread and here
    if (pthread_create(NULL, NULL, pps_thread, (void*)winmgr) < 0) {
        SLOG_ERROR("Failed to create a pps thread (%d:%s)", errno, strerror(errno));
    }

    if (pthread_create(NULL, NULL, screen_thread, (void*)winmgr) < 0) {
        SLOG_ERROR("Failed to create a screen thread (%d:%s)", errno, strerror(errno));
    }

    // launcher apps
    while (winmgr->pps_fd == -1) {
        sleep(1);
    }
    launcher_send(winmgr, CMD_START, winmgr->car_app.data, winmgr->car_app.id);
    launcher_send(winmgr, CMD_START, winmgr->weather_app.data, winmgr->weather_app.id);

    sleep(30);
    launcher_send(winmgr, CMD_STOP, "", "");

    sleep(5);

    // create a self_detached thread
    pthread_cancel(winmgr->pps_tid);
    pthread_cancel(winmgr->screen_tid);

    return EXIT_SUCCESS;
}
```

screen.c

Window management of a simple window manager

```
/*
* $QNXLicenseC:
* Copyright 2012, 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 "struct.h"

static void set_window_properties(screen_window_t win)
{
    static int size[2] = {400, 240};
    static int pos[2] = {0, 0};
    static int order = 100;
    // if it is the video clip, set it to a fixed position
    static int i = 0;
    int r = i%4;

    if (r == 0) {
        pos[0] = 0;
        pos[1] = 0;
    } else if (r == 1) {
        pos[0] = 400;
        pos[1] = 0;
    } else if (r == 2) {
        pos[0] = 0;
        pos[1] = 240;
    } else if (r == 3) {
        pos[0] = 400;
        pos[1] = 240;
    }

    screen_set_window_property_iv(win, SCREEN_PROPERTY_SIZE, size);
    screen_set_window_property_iv(win, SCREEN_PROPERTY_POSITION, pos);
    screen_set_window_property_iv(win, SCREEN_PROPERTY_ZORDER, &order);
//    printf("application index: %d\tpos: %d,%d\n", r, pos[0], pos[1]);
    i++;
}

int screen_init(window_manager_t *winmgr, 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.
     **/

    screen_context_t screen_ctx;   /* connection to screen windowing system */
    screen_window_t screen_win;    /* native handle for our window */
    screen_buffer_t screen_buf;    /* renderable buffers for the window */
    screen_event_t screen_ev;      /* handle used to pop events from our queue */
    int size[2] = { 64, 64 };      /* size of the window on screen */
    int pos[2] = { 0, 0 };         /* position of the window on screen */
    int val;                       /* used for simple property queries */
    const char *tok;               /* used to process command line arguments */
    int rval = EXIT_FAILURE;       /* application exits with value stored here */
    int rc;                        /* store return value from functions */
    int i;                         /* loop/frame counter */
    int stride;                    /* size of each window line in bytes */
    void *pointer;                 /* virtual address of the window buffer */
    int zorder = 0;
    char *group_name = strdup("default-group");

    /**
     ** 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], "-size=", strlen("-size=")) == 0) {
            /**
             ** The syntax of the size option is -size=(width)x(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).
             **/
            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], "-order=", strlen("-order=")) == 0) {
            tok = argv[i] + strlen("-order=");
            zorder = atoi(tok);
        }
        else if (strncmp(argv[i], "-name=", strlen("-name=")) == 0) {
            tok = argv[i] + strlen("-name=");
            if (group_name) {
                free(group_name);
            }
            group_name = strdup(tok);
        }
        else {
            /**
             ** Make sure we say something instead of silently ignoring a
             ** command line option.
             **/
            fprintf(stderr, "invalid command line option: %s\n", argv[i]);
        }
    }

    /**
     ** The first step is to connect to the windowing system. A standard
     ** application would pass a 0 or SCREEN_APPLICATION_CONTEXT for the
     ** second argument. We are demonstration some of the features the
     ** screen windowing system provides to window managers, so in our case
     ** we want to pass SCREEN_WINDOW_MANAGER_CONTEXT. For security reasons,
     ** this can only succeed if the application is started as root.
     **/

    rc = screen_create_context(&screen_ctx, SCREEN_WINDOW_MANAGER_CONTEXT);
    if (rc) {
        perror("screen_context_create");
        goto fail1;
    }

    /**
     ** Now we create a window. It is not necessary to have a window to get window
     ** manager events like window creation and destruction notifications. We
     ** create a window so we can get input (e.g., pointer and keyboard) events.
     **/

    rc = screen_create_window(&screen_win, screen_ctx);
    if (rc) {
        perror("screen_create_window");
        goto fail2;
    }

    /**
     ** We are going to use a plain memset to fill our window with a solid
     ** color. In order to guarantee that we will get a pointer to the buffer,
     ** we must set the usage to SCREEN_USAGE_WRITE. The SCREEN_PROPERTY_USAGE
     ** requires an array of a single integer. We will use val as a generic
     ** variable to hold our usage temporarily.
     **/

    val = SCREEN_USAGE_WRITE;
    rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_USAGE, &val);
    if (rc) {
        perror("screen_set_window_property_iv(SCREEN_PROPERTY_USAGE)");
        goto fail3;
    }
//    val = SCREEN_FORMAT_RGB565;
//    rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_FORMAT, &val);
//    if (rc) {
//        perror("screen_set_window_property_iv(SCREEN_PROPERTY_FORMAT)");
//        goto fail3;
//    }

    /**
     ** By default, windows are full screen unless the application specifically
     ** sets the window size. We don't necessarily want to be full screen,
     ** since we'll want to run a couple of other applications to show that we
     ** are getting window manager events. The SCREEN_PROPERTY_SIZE requires
     ** two integers (the width and height), so we use the iv variant and pass
     ** size, which is an array of two.
     **/
    rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_SIZE, size);
    if (rc) {
        perror("screen_set_window_property_iv(SCREEN_PROPERTY_SIZE)");
        goto fail3;
    }

    /**
     ** We also allow the position of our window to be controlled by the -pos
     ** command line argument. This might be useful if the default position
     ** obscured an area of interest of another window we want to see while
     ** running this tutorial. The SCREEN_PROPERTY_POSITION also requires two
     ** integers (the x and y offsets), so again we use the iv variant and pass
     ** pos, which is an array of two integers.
     **/

    rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_POSITION, pos);
    if (rc) {
        perror("screen_set_window_property_iv(SCREEN_PROPERTY_POSITION)");
        goto fail3;
    }

    rc = screen_set_window_property_iv(screen_win, SCREEN_PROPERTY_ZORDER, &zorder);
    if (rc) {
        perror("screen_set_window_property_iv(SCREEN_PROPERTY_POSITION)");
        goto fail3;
    }

    /**
     ** A window will never be visible until at least one buffer was created
     ** to hold its contents and one frame was posted. Because we don't plan to
     ** change the contents of this window, we don't need more than one buffer.
     **/

    rc = screen_create_window_buffers(screen_win, 1);
    if (rc) {
        perror("screen_create_window_buffers");
        goto fail3;
    }

    /**
     ** We want to fill our window with a solid color so we can see it. To do
     ** that, we will need to get a pointer to the buffer. We get the buffer
     ** first, so we can later query the pointer and stride properties that we
     ** need. The SCREEN_PROPERTY_RENDER_BUFFERS returns up to n buffers, where
     ** n is the number of buffers created or attached to a window. We've
     ** created one, so we only need to pass in an array of one buffer handle.
     **/
```

```
rc = screen_get_window_property_pv(screen_win,
                                   SCREEN_PROPERTY_RENDER_BUFFERS,
                                   (void **)&screen_buf);
if (rc) {
    perror("screen_get_pixmap_property_pv(SCREEN_PROPERTY_RENDER_BUFFERS)");
    goto fail3;
}

/**
 ** Now we query the pointer from our buffer. Because we've set the usage
 ** to include write access, this should be a pointer to memory we can
 ** write to. The SCREEN_PROPERTY_POINTER returns a single pointer, so we
 ** pass the address of a void pointer variable, which is equivalent to
 ** passing an array of one pointer.
 **/

rc = screen_get_buffer_property_pv(screen_buf, SCREEN_PROPERTY_POINTER, &pointer);
if (rc) {
    perror("screen_get_buffer_property_pv(SCREEN_PROPERTY_POINTER)");
    goto fail3;
}

/**
 ** The last piece of information we need before we can fill our window
 ** with a solid color is the stride of the buffer. The stride is the size,
 ** in bytes, of each line of the buffer. This may or may not be the same
 ** as the width times the bit depth. The SCREEN_PROPERTY_STRIDE writes to
 ** an array of one integer, so we can pass in the address of our stride
 ** variable.
 **/

rc = screen_get_buffer_property_iv(screen_buf, SCREEN_PROPERTY_STRIDE, &stride);
if (rc) {
    perror("screen_get_buffer_property_iv(SCREEN_PROPERTY_STRIDE)");
    goto fail3;
}

/**
 ** The following line fills the window buffer with a solid color pattern.
 ** We don't really care about the color, as long as we see something on
 ** the screen, so we will simply use memset. The transparency will be off
 ** by default, and our format is RGBX8888, so we don't need to put a 255
 ** in the alpha channel.
 **/

memset(pointer, 0x80, stride * size[1]);
```

```
// to test window group
rc = screen_create_window_group(screen_win, group_name);
if (rc) {
    perror("screen_create_window_group");
    goto fail3;
}

/**
 ** Nothing is going to be visible on the screen until we post the changes.
 ** Posting will tell the windowing system that we're done drawing into our
 ** render buffer and that we want the changes to be made visible. When we
 ** post we must indicate which parts of the buffer have changed. This
 ** allows the composited windowing system to be smart and redraw only the
 ** parts of the frame buffer that need an update. Since this is our first
 ** frame we naturally put a full dirty rect.
 **/

int rect[4] = { 0, 0, size[0], size[1] };
rc = screen_post_window(screen_win, screen_buf, 1, rect, 0);
if (rc) {
    perror("screen_post_window");
    goto fail3;
}

rc = screen_flush_context(screen_ctx, SCREEN_WAIT_IDLE);
if (rc) {
    perror("screen_post_window");
    goto fail3;
}

rc = screen_create_event(&screen_ev);
if (rc) {
    perror("screen_create_event");
    goto fail3;
}

screen_display_t *displays;
int display_count = 0, port;

printf("checking displays\n");
screen_get_context_property_iv(screen_ctx, SCREEN_PROPERTY_DISPLAY_COUNT, &display_count);
printf("%d displays\n", display_count);

displays = malloc(display_count * sizeof(screen_display_t));
screen_get_context_property_pv(screen_ctx, SCREEN_PROPERTY_DISPLAYS, (void *)displays);
```

```
    for(i = 0; i < display_count; i++ ) {
        screen_get_display_property_iv(displays[i], SCREEN_PROPERTY_ATTACHED, &val);
        screen_get_display_property_iv(displays[i], SCREEN_PROPERTY_ID, &port);
        printf("display %d (port %d) is %stached\n", i, port, val?"at":"de");
    }

    winmgr->screen_ctx = screen_ctx;
    winmgr->screen_win = screen_win;
    winmgr->screen_ev = screen_ev;

    return EOK;

    fail3:
        screen_destroy_window(screen_win);
    fail2:
        screen_destroy_context(screen_ctx);
    fail1:
        return rval;
}

void *screen_thread(void *arg)
{
    window_manager_t *winmgr = (window_manager_t*)arg;
    const int exit_area_size = 20;

    screen_context_t screen_ctx = winmgr->screen_ctx; /* connection to screen windowing system */
    screen_window_t screen_win = winmgr->screen_win;  /* native handle for our window */
    screen_event_t screen_ev = winmgr->screen_ev; /* handle used to pop events from our queue */
    screen_window_t win;           /* stores a window contained in an event */
    int size[2] = { 64, 64 };      /* size of the window on screen */
    int val;                       /* used for simple property queries */
    int pair[2];                   /* used to query pos, size */
    void *ptr;                     /* used to query user handles */
    char str[128];                 /* used to query string properties */
    int i;                         /* loop/frame counter */
//    void *pointer;                 /* virtual address of the window buffer */
//    int zorder = 0;

    screen_display_t *displays, disp;
    int display_count = 0, port;

//    char *group_name = strdup("default-group");

    pthread_setname_np(winmgr->screen_tid = pthread_self(), "screen_monitor");

    while (winmgr->state & WINMGR_UPDATE)
    {
```

```
while (!screen_get_event(screen_ctx, screen_ev, ~0L))
{
    screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &val);

    switch (val) {
        case SCREEN_EVENT_DISPLAY:
            if ( screen_get_event_property_pv(screen_ev,
                                              SCREEN_PROPERTY_DISPLAY,
                                              (void *)&disp) == 0 ) {
                printf("SCREEN_EVENT_DISPLAY(display=%p)\n", disp);
            } else {
                perror("SCREEN_PROPERTY_DISPLAY");
                break;
            }
            screen_get_display_property_iv(disp, SCREEN_PROPERTY_TYPE, &val);

            switch (val) {
                case SCREEN_DISPLAY_TYPE_HDMI:
                    screen_get_display_property_iv(disp,
                                                   SCREEN_PROPERTY_ATTACHED,
                                                   &val);
                    port = 0;
                    screen_get_display_property_iv(disp,
                                                   SCREEN_PROPERTY_ID,
                                                   &port);
                    printf("HDMI display (port %d) is %stached\n",
                           port, val?"at":"de");
                    break;
                default:
                    printf("display %p is type %#x\n", disp, val );
                    break;
            }
            break;
        case SCREEN_EVENT_IDLE:
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_IDLE_STATE,
                                         &val);
            printf("SCREEN_EVENT_IDLE_STATE(state=%d)\n", val);
            screen_get_context_property_iv(screen_ctx,
                                           SCREEN_PROPERTY_IDLE_STATE,
                                           &val);
            printf("context idle state is %d\n", val);
            screen_get_context_property_iv(screen_ctx,
                                           SCREEN_PROPERTY_DISPLAY_COUNT,
                                           &display_count);
            printf("%d displays\n", display_count);
            displays = malloc(display_count * sizeof(screen_display_t));
            screen_get_context_property_pv(screen_ctx,
                                           SCREEN_PROPERTY_DISPLAYS,
                                           (void *)displays);

            for (i=0; i < display_count; i++) {
                screen_get_display_property_iv(displays[i],
                                               SCREEN_PROPERTY_KEEP_AWAKES,
                                               &val);
                printf("display %d has %d keep awake windows\n", i, val );
            }
            free(displays);
            break;
        case SCREEN_EVENT_CREATE:
            screen_get_event_property_pv(screen_ev,
                                         SCREEN_PROPERTY_WINDOW,
                                         (void **)&win);
            screen_get_window_property_iv(win,
                                          SCREEN_PROPERTY_OWNER_PID,
                                          &val);
            printf("SCREEN_EVENT_CREATE(window=0x%08x, pid=%d, handle=0x%08x)\n",
                   (size_t)win, val, (size_t)ptr);
            break;
        case SCREEN_EVENT_PROPERTY:
            screen_get_event_property_pv(screen_ev,
                                         SCREEN_PROPERTY_WINDOW,
                                         (void **)&win);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_NAME,
                                         &val);

            switch (val) {
                case SCREEN_PROPERTY_ALPHA_MODE:
                    screen_get_window_property_iv(win,
                                                  SCREEN_PROPERTY_ALPHA_MODE,
                                                  &val);
                    if (val) {
                        printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
                               pname=SCREEN_PROPERTY_ALPHA_MODE,
                               value=SCREEN_ALPHA_MODE_PREMULTIPLIED)\n",
                               (size_t)win);
                    } else {
                        printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
                               pname=SCREEN_PROPERTY_ALPHA_MODE,
                               value=SCREEN_ALPHA_MODE_NONPREMULTIPLIED)\n",
                               (size_t)win);
                    }
```

```
    break;
case SCREEN_PROPERTY_BRIGHTNESS:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_BRIGHTNESS,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_BRIGHTNESS, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_BUFFER_COUNT:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_BUFFER_COUNT,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_BUFFER_COUNT, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_BUFFER_SIZE:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_BUFFER_SIZE,
                                  pair);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_BUFFER_COUNT, value=%dx%d)\n",
           (size_t)win, pair[0], pair[1]);
    break;
case SCREEN_PROPERTY_CLASS:
    screen_get_window_property_cv(win,
                                  SCREEN_PROPERTY_CLASS,
                                  sizeof(str),
                                  str);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_CLASS, value=%s)\n",
           (size_t)win, str);
    break;
case SCREEN_PROPERTY_COLOR_SPACE:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_COLOR_SPACE,
                                  &val);
    if (val) {
        printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
               pname=SCREEN_PROPERTY_COLOR_SPACE,
               value=SCREEN_COLOR_SPACE_LINEAR)\n",
               (size_t)win);
    } else {
        printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
               pname=SCREEN_PROPERTY_COLOR_SPACE,
               value=SCREEN_COLOR_SPACE_sRGB)\n",
               (size_t)win);
    }
    break;
case SCREEN_PROPERTY_CONTRAST:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_CONTRAST,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_CONTRAST, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_FLIP:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_FLIP,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_FLIP, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_FORMAT:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_FORMAT,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_FORMAT, value=",
           (size_t)win);

    switch (val) {
        case SCREEN_FORMAT_BYTE:
            printf("SCREEN_FORMAT_BYTE)\n");
            break;
        case SCREEN_FORMAT_RGBA4444:
            printf("SCREEN_FORMAT_RGBA4444)\n");
            break;
        case SCREEN_FORMAT_RGBX4444:
            printf("SCREEN_FORMAT_RGBX4444)\n");
            break;
        case SCREEN_FORMAT_RGBA5551:
            printf("SCREEN_FORMAT_RGBA5551)\n");
            break;
        case SCREEN_FORMAT_RGBX5551:
            printf("SCREEN_FORMAT_RGBX5551)\n");
            break;
        case SCREEN_FORMAT_RGB565:
            printf("SCREEN_FORMAT_RGB565)\n");
            break;
        case SCREEN_FORMAT_RGB888:
```

```
                        printf("SCREEN_FORMAT_RGB888)\n");
                        break;
                    case SCREEN_FORMAT_RGBA8888:
                        printf("SCREEN_FORMAT_RGBA8888)\n");
                        break;
                    case SCREEN_FORMAT_RGBX8888:
                        printf("SCREEN_FORMAT_RGBX8888)\n");
                        break;
                    case SCREEN_FORMAT_YVU9:
                        printf("SCREEN_FORMAT_YVU9)\n");
                        break;
                    case SCREEN_FORMAT_YUV420:
                        printf("SCREEN_FORMAT_YUV420)\n");
                        break;
                    case SCREEN_FORMAT_NV12:
                        printf("SCREEN_FORMAT_NV12)\n");
                        break;
                    case SCREEN_FORMAT_YV12:
                        printf("SCREEN_FORMAT_YV12)\n");
                        break;
                    case SCREEN_FORMAT_UYVY:
                        printf("SCREEN_FORMAT_UYVY)\n");
                        break;
                    case SCREEN_FORMAT_YUY2:
                        printf("SCREEN_FORMAT_YUY2)\n");
                        break;
                    case SCREEN_FORMAT_YVYU:
                        printf("SCREEN_FORMAT_YUY2)\n");
                        break;
                    case SCREEN_FORMAT_V422:
                        printf("SCREEN_FORMAT_V422)\n");
                        break;
                    case SCREEN_FORMAT_AYUV:
                        printf("SCREEN_FORMAT_AYUV)\n");
                        break;
                    default:
                        printf("%d)\n", val);
                        break;
                }
                break;
            case SCREEN_PROPERTY_GLOBAL_ALPHA:
                screen_get_window_property_iv(win,
                                              SCREEN_PROPERTY_GLOBAL_ALPHA,
                                              &val);
                printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
                       pname=SCREEN_PROPERTY_GLOBAL_ALPHA, value=%d)\n",
                       (size_t)win, val);
```

```
    break;
case SCREEN_PROPERTY_HUE:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_HUE,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_HUE, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_ID_STRING:
    screen_get_window_property_cv(win,
                                  SCREEN_PROPERTY_ID_STRING,
                                  sizeof(str),
                                  str);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_ID_STRING, value=%s)\n",
           (size_t)win, str);
    break;
case SCREEN_PROPERTY_MIRROR:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_MIRROR,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_MIRROR, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_POSITION:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_POSITION,
                                  pair);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_POSITION, value=%d,%d)\n",
           (size_t)win, pair[0], pair[1]);
    break;
case SCREEN_PROPERTY_ROTATION:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_ROTATION,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_ROTATION, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_SATURATION:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_SATURATION,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_SATURATION, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_SIZE:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_SIZE,
                                  pair);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_SIZE, value=%dx%d)\n",
           (size_t)win, pair[0], pair[1]);
    break;
case SCREEN_PROPERTY_SOURCE_POSITION:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_SOURCE_POSITION,
                                  pair);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_SOURCE_POSITION, value=%d,%d)\n",
           (size_t)win, pair[0], pair[1]);
    break;
case SCREEN_PROPERTY_SOURCE_SIZE:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_SOURCE_SIZE,
                                  pair);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_SOURCE_SIZE, value=%dx%d)\n",
           (size_t)win, pair[0], pair[1]);
    break;
case SCREEN_PROPERTY_STATIC:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_STATIC,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_STATIC, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_SWAP_INTERVAL:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_SWAP_INTERVAL,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_SWAP_INTERVAL, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_TRANSPARENCY:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_TRANSPARENCY,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_TRANSPARENCY, value=",
           (size_t)win);

    switch (val) {
        case SCREEN_TRANSPARENCY_NONE:
            printf("SCREEN_TRANSPARENCY_NONE)\n");
            break;
        case SCREEN_TRANSPARENCY_TEST:
            printf("SCREEN_TRANSPARENCY_TEST)\n");
            break;
        case SCREEN_TRANSPARENCY_SOURCE_COLOR:
            printf("SCREEN_TRANSPARENCY_SOURCE_COLOR)\n");
            break;
        case SCREEN_TRANSPARENCY_SOURCE_OVER:
            printf("SCREEN_TRANSPARENCY_SOURCE_OVER)\n");
            break;
        default:
            printf("%d)\n", val);
            break;
    }
    break;
case SCREEN_PROPERTY_USAGE:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_USAGE,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_USAGE, value=0x%04x)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_VISIBLE:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_VISIBLE,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_VISIBLE, value=%d)\n",
           (size_t)win, val);
    break;
case SCREEN_PROPERTY_ZORDER:
    screen_get_window_property_iv(win,
                                  SCREEN_PROPERTY_ZORDER,
                                  &val);
    printf("SCREEN_EVENT_PROPERTY(window=0x%08x,
           pname=SCREEN_PROPERTY_ZORDER, value=%d)\n",
           (size_t)win, val);
    break;
default:
```

```
                    printf("SCREEN_EVENT_PROPERTY(window=0x%08x, pname=%d)\n",
                           (size_t)win, val);
                    break;
            }
            break;
        case SCREEN_EVENT_CLOSE:
            screen_get_event_property_pv(screen_ev,
                                         SCREEN_PROPERTY_WINDOW,
                                         (void **)&win);
            screen_get_window_property_pv(win,
                                          SCREEN_PROPERTY_USER_HANDLE,
                                          &ptr);
            printf("SCREEN_EVENT_CLOSE(window=0x%08x, handle=0x%08x)\n",
                   (size_t)win, (size_t)ptr);
            screen_destroy_window(win);
            break;
        case SCREEN_EVENT_POST:
            screen_get_event_property_pv(screen_ev,
                                         SCREEN_PROPERTY_WINDOW,
                                         (void **)&win);
            screen_get_window_property_pv(win,
                                          SCREEN_PROPERTY_USER_HANDLE,
                                          &ptr);
            printf("SCREEN_EVENT_POST(window=0x%08x, handle=0x%08x)\n",
                   (size_t)win, (size_t)ptr);

            set_window_properties(win);
            screen_flush_context(screen_ctx, 0);
            break;
        case SCREEN_EVENT_INPUT:
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_DEVICE_INDEX,
                                         &val);
            printf("SCREEN_EVENT_INPUT(index=%d, ", val);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_INPUT_VALUE,
                                         &val);
            printf("value=%d)\n", val);
            break;
        case SCREEN_EVENT_JOG:
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_DEVICE_INDEX,
                                         &val);
            printf("SCREEN_EVENT_JOG(index=%d, ", val);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_JOG_COUNT,
                                         &val);
            printf("count=%d)\n", val);
            break;
        case SCREEN_EVENT_POINTER:
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_DEVICE_INDEX,
                                         &val);
            printf("SCREEN_EVENT_POINTER(index=%d, ", val);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_POSITION,
                                         pair);
            printf("pos=[%d,%d], ", pair[0], pair[1]);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_BUTTONS,
                                         &val);
            printf("buttons=0x%04x)\n", val);

            if (val) {
                if (pair[0] >= size[0] - exit_area_size &&
                    pair[0] < size[0] &&
                    pair[1] >= 0 &&
                    pair[1] < exit_area_size) {
                    goto end;
                }
            }
            break;
        case SCREEN_EVENT_KEYBOARD:
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_DEVICE_INDEX,
                                         &val);
            printf("SCREEN_EVENT_KEYBOARD(index=%d, ", val);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_KEY_CAP,
                                         &val);
            printf("cap=%d, ", val);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_KEY_FLAGS,
                                         &val);
            printf("flags=%d, ", val);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_KEY_MODIFIERS,
                                         &val);
            printf("modifiers=%d, ", val);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_KEY_SCAN,
                                         &val);
            printf("scan=%d, ", val);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_KEY_SYM,
                                         &val);
            printf("sym=%d)\n", val);

            switch (val) {
                case KEYCODE_ESCAPE:
                    goto end;
            }
            break;
        case SCREEN_EVENT_MTOUCH_TOUCH:
        case SCREEN_EVENT_MTOUCH_MOVE:
        case SCREEN_EVENT_MTOUCH_RELEASE:
            switch (val) {
                case SCREEN_EVENT_MTOUCH_TOUCH:
                    printf("SCREEN_EVENT_MTOUCH_TOUCH(");
                    break;
                case SCREEN_EVENT_MTOUCH_MOVE:
                    printf("SCREEN_EVENT_MTOUCH_MOVE(");
                    break;
                case SCREEN_EVENT_MTOUCH_RELEASE:
                    printf("SCREEN_EVENT_MTOUCH_RELEASE(");
                    break;
            }
            screen_get_event_property_pv(screen_ev,
                                         SCREEN_PROPERTY_WINDOW,
                                         (void **)&win);
            printf("window=0x%08x, ", (size_t)win);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_TOUCH_ID,
                                         &val);
            printf("id=%d, ", val);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_SEQUENCE_ID,
                                         &val);
            printf("sequence=%d, ", val);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_POSITION,
                                         pair);
            printf("pos=[%d,%d], ", pair[0], pair[1]);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_SIZE,
                                         pair);
            printf("size=[%d,%d], ", pair[0], pair[1]);
            screen_get_event_property_iv(screen_ev,
                                         SCREEN_PROPERTY_SOURCE_POSITION,
                                         pair);
            printf("source pos=[%d,%d], ", pair[0], pair[1]);
```

```
                    screen_get_event_property_iv(screen_ev,
                                                 SCREEN_PROPERTY_SOURCE_SIZE,
                                                 pair);
                    printf("source size=[%d,%d], ", pair[0], pair[1]);
                    screen_get_event_property_iv(screen_ev,
                                                 SCREEN_PROPERTY_TOUCH_ORIENTATION,
                                                 &val);
                    printf("orientation=%d, ", val);
                    screen_get_event_property_iv(screen_ev,
                                                 SCREEN_PROPERTY_TOUCH_PRESSURE,
                                                 &val);
                    printf("pressure=%d)\n", val);
                    break;
                case SCREEN_EVENT_USER:
                    break;
            }
        }
    }

end:
    screen_destroy_event(screen_ev);
    screen_destroy_window(screen_win);
    screen_destroy_context(screen_ctx);

    return 0;
}
```

launcher.c

Interaction with the `launcher` *of a simple window manager*

```
/*
* $QNXLicenseC:
* Copyright 2012, 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 "struct.h"

int launcher_pps(window_manager_t *winmgr)
{
    char *msg, *dat, *res, *err, *pkg, *errstr;

    // There must be a msg and a dat attribute
    msg = pps_lookup(winmgr, "msg");
    res = pps_lookup(winmgr, "res");
    err = pps_lookup(winmgr, "err");
    dat = pps_lookup(winmgr, "dat");
    pkg = pps_lookup(winmgr, "id");
    errstr = pps_lookup(winmgr, "errstr");

    if ((msg == NULL && res == NULL) || (dat == NULL && err == NULL))
        return EINVAL;

    if (winmgr->verbose) {
        if (msg)
            SLOG_NOTICE("launcher - msg:%s dat:%s", msg, dat);
        else
            SLOG_NOTICE("launcher - res:%s dat/err:%s", res, dat ? dat : err);
    }

    if (msg) {
        if (strcmp(msg, CMD_STOPPED) == 0) {
            core_app_stopped(winmgr, dat);
        } else if (strcmp(msg, CMD_LOWMEM) == 0) {
            core_lowmem(winmgr, dat);
        }
    } else {
        if (strcmp(res, CMD_START) == 0) {
            core_app_started(winmgr, pkg, dat ? dat : err, dat ? 0 : 1, errstr);
        }
        if (strcmp(res, CMD_DEBUG) == 0) {
            core_app_started(winmgr, pkg, dat ? dat : err, dat ? 0 : 1, errstr);
        }
    }
    return EOK;
}

void launcher_send(window_manager_t *winmgr, char *cmd, char *data, char *id)
{
    int msgsize;
    char msgbuf[4096];

    if (strcmp(cmd, CMD_START) == 0 || strcmp(cmd, CMD_DEBUG) == 0) {
        msgsize = snprintf(msgbuf, sizeof(msgbuf), "msg::%s\ndat::%s\nid::%s", cmd, data, id);
    } else {
        msgsize = snprintf(msgbuf, sizeof(msgbuf), "msg::%s\ndat::%s", cmd, data);
    }
    if (winmgr->verbose)
        SLOG_NOTICE("launch send - msg:%s dat:%s id:%s", cmd, data ? data : "", id ? id : "");

    if (pps_write(winmgr->pps_fd, msgbuf, msgsize) == -1 ) {
        SLOG_NOTICE("unable to write to launcher fd: %s", strerror(errno));
    }
}
```

pps.c

Interaction with PPS control object of a simple window manager

```
/*
* $QNXLicenseC:
* Copyright 2012, 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 "struct.h"

static int pps_parse(window_manager_t *winmgr, char* reqbuf);

/**
 * pps thread loop
 */
void* pps_thread(void *arg)
{
    window_manager_t *winmgr = (window_manager_t*)arg;

    char buf[1024];
    int fd = -1;
    int nread = -1;

    pthread_setname_np(winmgr->pps_tid = pthread_self(), "pps");

    while (1) {
        while (fd == -1) {
            if (winmgr->state & NAV_TERMINATE) {
                goto terminate;
            }

            if ((fd = open ("/pps/services/launcher/control?wait,delta", O_RDWR)) == -1 ) {
                sleep(1);
            } else {
                winmgr->pps_fd = fd;
            }
        }

        while (nread == -1) {

            if (winmgr->state & NAV_TERMINATE) {
                goto terminate;
            }

            nread = read(fd, buf, sizeof(buf)-1);
            if (nread > 1) {
                buf[nread] = '\0';

                // process received message
                if (pps_parse(winmgr, buf) == EOK
                        && !(winmgr->ptype == PPS_EVENT_OBJECT_CHANGED && !winmgr->numattrs)
                        && winmgr->ptype != PPS_EVENT_OBJECT_UNKNOWN) {
                    launcher_pps(winmgr);
                }
            }
        }

        if (nread == -1 && errno == EBADF) {
            fd = -1;
        }

        nread = -1;
    }

terminate:
    close(fd);
    pthread_exit(NULL);

    return NULL;
}

int pps_write(int pps_fd, const char *msgbuf, int msgsize)
{
    int ret = 0;
    ret = write(pps_fd, msgbuf, (unsigned)msgsize);

    return ret;
}

int pps_is_open(int pps_fd)
{
    int ret = 1;

    if (pps_fd == 0 || pps_fd == -1) {
        ret = 0;
    }

    return ret;
}

pps_attr_t* pps_lookup_attr(window_manager_t *winmgr, char *name)
{
    int i;

    for (i = 0; i < winmgr->numattrs; ++i)
        if (strcmp(winmgr->attrs[i].name, name) == 0)
            return &(winmgr->attrs[i]);
    return NULL;
}

char* pps_lookup(window_manager_t *winmgr, char *name)
{
    pps_attr_t* attr = pps_lookup_attr(winmgr, name);

    if (!attr)
        return NULL;

    if (attr->value)
        return attr->value;
    else
        return attr->name;
}

static int pps_parse_attr(window_manager_t *winmgr, const pps_attrib_t* info)
{
    if (winmgr->numattrs >= MAX_ATTRS) {
        SLOG_NOTICE("Too many attributes.");
        return E2BIG;
    } else {
        winmgr->attrs[winmgr->numattrs].name = info->attr_name;
        winmgr->attrs[winmgr->numattrs].encoding = info->encoding;
        winmgr->attrs[winmgr->numattrs++].value = info->value;
        return EOK;
    }
}

static int pps_parse(window_manager_t *winmgr, char* reqbuf)
{
    pps_attrib_t    info;
    pps_status_t    rc;

    // Clear the request structure
    winmgr->numattrs = 0;
    winmgr->objname = NULL;
    winmgr->ptype = PPS_EVENT_OBJECT_UNKNOWN;

    while ((rc = ppsparse(&reqbuf, NULL, NULL, &info, 0)) != PPS_END)
    {
        switch (rc) {
        case PPS_OBJECT_CREATED:
            winmgr->objname = info.obj_name;
            winmgr->ptype = PPS_EVENT_OBJECT_CREATED;
            break;

        case PPS_OBJECT_TRUNCATED:
        case PPS_OBJECT_DELETED:
            winmgr->objname = info.obj_name;
            winmgr->ptype = PPS_EVENT_OBJECT_DELETED;
            break;

        case PPS_OBJECT:
            if (info.obj_name[0] != '@') {
                return EOK;
            }
            winmgr->objname = info.obj_name;
            winmgr->ptype = PPS_EVENT_OBJECT_CHANGED;
            break;

        case PPS_ATTRIBUTE_DELETED: {
            --info.attr_name;
            int err = pps_parse_attr(winmgr, &info);
            if (err != EOK)
                return err;
            break;
        }

        case PPS_ATTRIBUTE: {
            int err = pps_parse_attr(winmgr, &info);
            if (err != EOK)
                return err;
            break;
        }

        case PPS_ERROR:
        default:
            SLOG_WARNING("We got a parsing error.");
            return EINVAL;
        }
    }

    return EOK;
}
```

core.c

Core functionality of a simple window manager

```
/*
* $QNXLicenseC:
* Copyright 2012, 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 "struct.h"

void core_app_stopped(window_manager_t *winmgr, char *data)
{
    /* Delete from the application list */
    if (winmgr->car_app.pid && !strcmp(data, winmgr->car_app.pid)) {
        free(winmgr->car_app.pid);
        winmgr->car_app.pid = NULL;
    }

    if (winmgr->weather_app.pid && !strcmp(data, winmgr->weather_app.pid)) {
        free(winmgr->weather_app.pid);
        winmgr->weather_app.pid = NULL;
    }
}

void
core_app_started(window_manager_t *winmgr, char *id, char *data, int error, char *errstr)
{
    /* Append to the application list */
    if (error) {
        SLOG_WARNING("failed to launch app: %s", errstr);
        return;
    }

    if (!strcmp(id, winmgr->car_app.id)) {
        winmgr->car_app.pid = strdup(data);
    } else if (!strcmp(id, winmgr->car_app.id)) {
        winmgr->weather_app.pid = strdup(data);
    }
}

void core_lowmem(window_manager_t *winmgr, char *data)
{
}
```

Index

/pps/services/launcher/control object *20*

A

application management *20*
apps *9*, *11–14*, *19*, *23–25*
- installing *9*, *11–12*
- launching *9*
- packaging *9*
- running *12*
- starting *9*, *12*, *23*
- stopping *13*, *24*
- uninstalling *14*
- window manager *23–25*
- window manager can start and stop *19*

E

events *28*

L

launching applications *12*

P

PPS *12–13*, *20*
- launcher control object *12–13*, *20*
- messages *20*

S

Screen *25*
`start` command *23*
`stop` command *24*

T

Technical support *8*
Typographical conventions *6*

U

uninstalling apps *14*

W

window management *23*, *25*
window manager *19–20*, *23*, *25*, *28*, *32*
- handling events *28*
- interacting with PPS *20*
- reference code for *32*
- requirements of a *20*
- responsible for application windows *25*
- setting up *25*
- simple example of *32*

write() *23–24*