

Implementation of Advanced Display Technologies on HP-UX Workstations

Todd M. Spencer

Paul M. Anderson

This article provides implementation and configuration information about the single logical screen (SLS) technology described in the article "Advanced Display Technologies on HP-UX Workstations."¹



Todd M. Spencer

A software engineer at the HP Workstation Systems Division, Todd

Spencer was responsible for development of the SAM component that allows users to set up multiscreen and single logical screen configurations. He came to HP in 1989 after receiving an MS degree in computer science from the University of Utah. Todd was born in Utah, is married and has four children. His outside interests include tropical fish, camping, woodworking, piano (playing classical music), and jogging.



Paul M. Anderson

Paul Anderson is a software engineer at the HP Workstation Systems Division. He joined HP in 1996 after receiving a

BS degree in computer science from the University of Minnesota. He is currently working on device drivers for new peripheral technologies. His professional interests include I/O drivers, operating systems, and networking. Paul was born in Edina, Minnesota. His outside interests include hiking, music, and mountain biking.

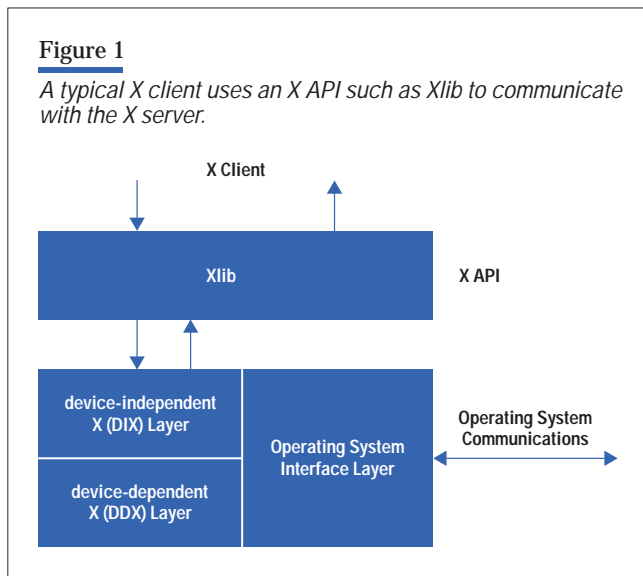
Understanding the implementation of SLS requires a little background on the X server architecture. The X server is divided into layers of functionality, each with a specific purpose and each interfacing with its neighbors. At the lowest level, which is closest to the physical hardware, is the device-dependent X (DDX) layer. This layer is responsible for controlling the frame buffer, including all rendering. Specifics of frame buffer control involve functions such as frame buffer initialization and state management, hardware locking, rendering, pointer control, hardware colormap management, and other functions such as direct memory access. Also included within the DDX layer is the detection and management of input.

At the highest layer is the device-independent X (DIX) layer. This layer contains code that is independent of the various hardware platforms. Think of this layer as the control layer. It includes the dispatch loop that manages the reception and processing of X protocol, event management, error detection, and other control, organization and bookkeeping operations. This layer also interfaces extensively with the operating system interface layer and with the DDX layers.

The operating system interface layer contains operating system calls, and it controls such things as memory management (allocation and deallocation), connections (socket creation), and kernel interactions. Although most of the operating system interface layer code is referenced from within the DIX layer, the DDX layer also invokes interface-layer-specific calls. Although there are other areas, such as font control, they have very little to do with SLS so they will be omitted from our discussion. **Figure 1** shows typical X Client using an X API such as Xlib to communicate with the X server.

Figure 1

A typical X client uses an X API such as Xlib to communicate with the X server.



The DDX, DIX, and operating system interface layers formulate the essence of the X server. User programs typically do not send X protocol directly to the X server. Rather, they use an API such as Xlib or one of the Motif toolkits that generate X protocol and communicate with the X server. For example, when a client issues an `XOpenDisplay()` call, a connection is established with the X server. A subsequent `XDrawRectangle()` request will then cause the Xlib library to generate the appropriate X protocol which eventually gets sent across the connection to the X server. The operating system interface layer procedure `WaitForSomething()` detects the arrival of each request. The `DIX Dispatch()` procedure decodes it and invokes the appropriate procedure. These high-level procedures perform error checking, ensure that the graphics context has been validated and possibly perform other internal bookkeeping before invoking the appropriate DDX entry point to draw the appropriate pixels on the screen.

In the HP-UX operating system, the X server executable does not contain static DDX modules. DDX modules are dynamically loaded and initialized via plug-and-play functionality within the `DIX InitOutput()` entry point. There is a collector mechanism in place that identifies which DDX modules can support a particular device and selects the best module available. Typically, when support for new hardware is added all that is required is a new DDX module and broker, which is another shared library that works with the collector to tell the X server what graphics device it supports and how it is to be loaded. With SLS,

we provided a new DDX module (and broker) and made some minor modifications to the DIX and input areas.

Initialization

During initialization of the HP-UX X server, screen configurations are set up by parsing the `X0screen` configuration file. The syntax for a normal screen is simply "Screen <device_name>" (for example, "Screen /dev/crt"). Whenever this syntax is parsed, a new screen is created and stored in the global `screenInfo` array.

Because SLS manages multiple devices, it cannot use the same `X0screen` configuration file parsing mechanism. It needs to know which devices to manage and how to organize them (for example, vertical, horizontal, or matrix layout). To accomplish this, the X server configuration syntax was extended. For SLS scenarios, the configuration file must specify the number of rows, columns, and device files that will be managed by the logical screen. The syntax is as follows*:

```
SingleLogicalScreen <nRows> <nCols> <device_name1> ...  
<device_nameN>
```

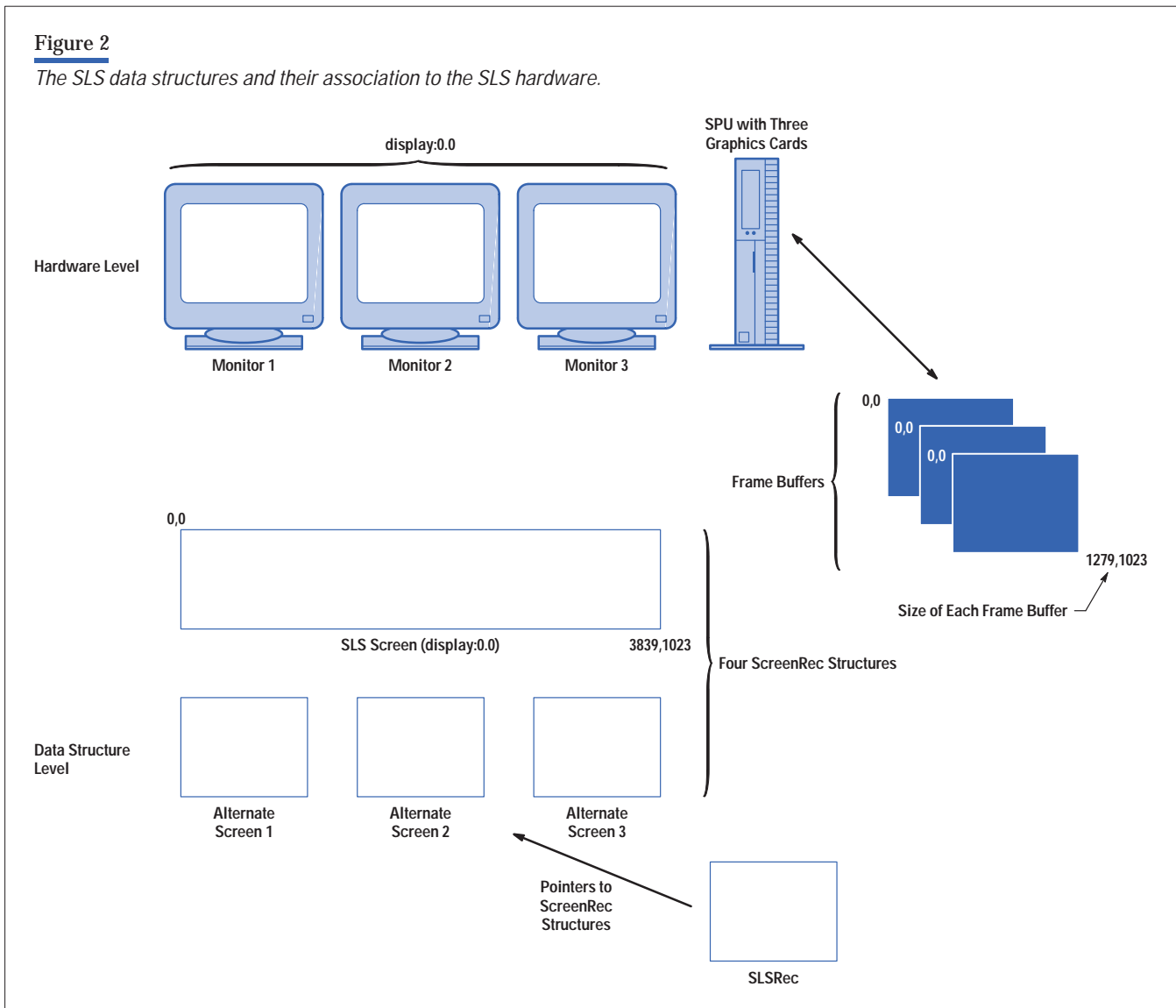
Whenever the parsing routines detect the `SingleLogicalScreen` token in the configuration file, a new procedure is called to parse the remaining SLS tokens, initialize the necessary global structures, and create the SLS screen. An interesting implementation detail to note is that we also create additional screen structures for each device being managed by the single logical screen. These screens are referred to as *alternate screens* and are hidden from the DIX layer in that they are not accessible by the global `screenInfo` array. There is another global data structure, called `SLSRec`, that points to these alternate screens and records information such as screen offsets. An instance of an `SLSRec` is created for each SLS screen. The structure and purpose of `SLSRec` is similar to the `screenInfo` global.

For each SLS screen and its components (alternate screens) an instance of the `ScreenRec` structure is created. `ScreenRec` is a high-level DIX structure that stores screen-specific information and function pointers for screen operations involving windows, pixmap, backing store, fonts, pointers, colormap, regions, and related operations. As shown in **Figure 2** there is one `ScreenRec` for

* This is transparent to users configuring the X server via the system administration manager (SAM) utility.

Figure 2

The SLS data structures and their association to the SLS hardware.



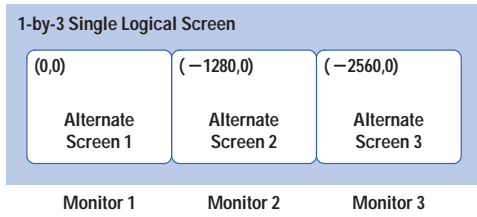
the SLS screen and one each for the alternate screens. This structure was modified to allow SLS to intercept protocol for mapping and unmapping windows, circulating windows, changing the window stacking order, and delivering events. Thus, when SLS intercepts one of these protocols, the SLS ScreenRec uses the function pointers in its jump table to invoke the functions associated with each alternate screen's ScreenRec. These functions use the information contained in their respective ScreenRecs to invoke other functions to perform the requested operations on their alternate screens. This architecture allows the DDX modules to drive the SLS screen in a device-independent fashion.

The reason for creating alternate screens is to leverage as much from the existing DDX drivers as possible. It would be unproductive to require a rewrite of all the existing code to support SLS. The SLS module is a controller that inserts itself into the DIX and DDX layers and performs the necessary control and manipulation to drive multiple DDX modules. In this way, we get the benefit of controlling multiple devices under a logical screen without having the development and maintenance overhead of creating and supporting SLS-specific drivers for all our graphics devices.

When each alternate screen is created, the origins are manipulated to account for the layout of the SLS screens.

Figure 3

Coordinate offsets in the alternate screens.



For example, in a 1-by-3 SLS configuration (see **Figure 3**), the first, or left-most screen, would have an offset of (0,0). The center screen would be offset by the width of the first screen (for example, -1280,0). In this case we ignore the height in horizontal configurations because each alternate screen begins at zero and extends to the height of the device. Finally, the last screen would have an x offset consisting of the width of the previous two screens (for example, -2560,0). The negative values are used because we need to compensate for the fact that the frame buffer begins at 0,0 but represents another location on the SLS screen (described next). This negative value adjusts for the SLS screen offset and allows the normal DDX driver to render to a location that is valid and visible for that device.

Request Duplication

In general, client requests for the SLS screen are duplicated across each of the alternate screens. For instance, when a window is created on an SLS screen, a window is also created on each of the alternate screens. These alternate windows have the same characteristics as the top-level SLS window. The same is true for other screen properties such as colormaps and graphics contexts. Although this creates some overhead, it enables SLS to operate transparently to the user and to the DDX modules. For example, creating a window with a specific geometry in a 1-by-3 SLS configuration, results in the creation of four windows, all with essentially the same geometry. One window is created for the SLS screen referenced by client applications via an identifier obtained with Xlib API calls to `XCreateWindow()` or `XCreateSimpleWindow()`. The other three windows are created on each alternate screen. Only

the SLS module in the X server knows about these alternate screen windows. When we create windows for alternate screens, we adjust the coordinates according to the offsets of each screen.

For example, in **Figure 4** the window defined by the coordinates $x=0$, $y=0$, $width=100$, $height=100$ is shifted to $x=0$, $y=0$ on alternate screen 1, to $x=-1280$, $y=0$ on alternate screen 2, and to $x=-2560$, $y=0$ on alternate screen 3. In this way, when a window overlaps monitors, only the correct portions are seen on each monitor. Out of the four windows created in this example, only the window on alternate screen 1 will be visible because it is completely contained within the frame buffer associated with alternate screen 1 because it is contained within region 0,0 and 1279,1023.

When rendering, an X Client must specify a target drawable, which is represented by a pixmap or a window. For rendering to a window, the SLS module must substitute the user-specified top-level window with the appropriate alternate windows before invoking the rendering procedures. A rendering procedure is called for each alternate window. Since this top-level SLS window has no frame buffer associated with it, all rendering must occur within the alternate windows.

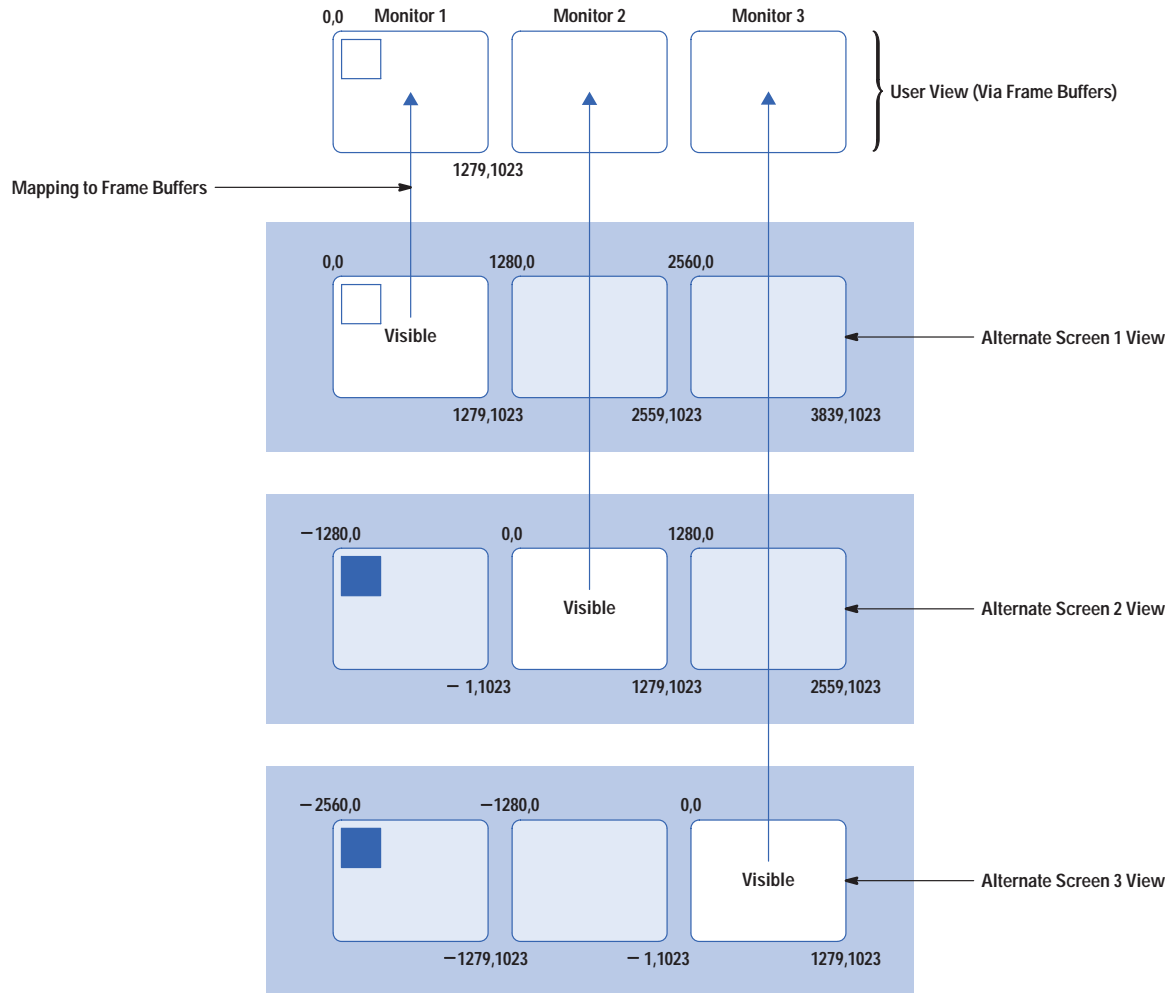
Event Processing

The implementation of the SLS module required some special considerations for event processing. If we had not made any changes to the SLS module, user applications would receive events in a different way than they receive them from a non-SLS environment, violating the design goal of application transparency and violating standard X protocol.

An event is a message generated by the X Server when a particular action occurs. Events are sent only to interested clients. A client application reads each event (in FIFO order) and performs the appropriate processing for that event. Events are commonly generated to notify the client of input events or state changes in the X server's window tree. In addition, pointer activities such as pressing mouse buttons and moving the mouse will cause the X server to generate events such as `ButtonPress`, `ButtonRelease`, and `MotionNotify`. Keyboard activity generates analogous

Figure 4

Mapping a window onto an SLS screen.



events such as `KeyPress` and `KeyRelease`. Examples of X server state change events include `EnterNotify` and `LeaveNotify` when the pointer enters or leaves a window or `DestroyNotify` when a window is destroyed. These are just a few examples of the various event types the X server can generate and clients can receive and process.

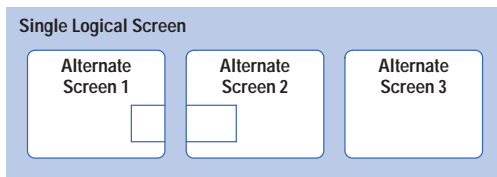
In describing the delivery of events to clients, we often use the term "interested client." We define a client as interested in an event if it has explicitly asked the X server to deliver that type of event. In this manner, the X server does not notify all clients about all events, which often reduces the complexity of client applications. This event

selection model also decreases network traffic by reducing the number of superfluous events traveling from the X server to its clients.

To illustrate the challenge posed by event processing in SLS, consider an example using a `VisibilityNotify` event. A `VisibilityNotify` event reports to an interested client that a window's visibility has changed. An interested client can then decide to begin or terminate rendering to that window if the event denotes that the window is partially obscured or fully obscured. Consider the SLS configuration shown in **Figure 5** that is composed of three screens with a window spanning screens 1 and 2.

Figure 5

A configuration in which a single state change could cause four VisibilityNotify events.



In this case, a single state change could cause four VisibilityNotify events (remember that four windows are created for this configuration: one for the SLS screen and one for each alternate screen). First, alternate screen 1 tells the client that the window is partially obscured. Second, alternate screen 2 gives the client the same message because each half of the window is only partially visible on each of the two physical screens it spans. As a result, an interested client will render to the window. The third event comes from alternate screen 3, which tells the client that the window is fully obscured so that the client will stop rendering to the window. Thus, the client gets several conflicting messages as to the visibility of the window. All three of these messages, however, do not give an accurate depiction of the window because another VisibilityNotify event will be generated, denoting that the window is fully visible on the SLS screen. Thus, our model of using alternate screens poses many problems in event handling.

Given the event processing complexity of the simple situation described above, what is the best mechanism to sort out a potentially conflicting series of events and give the client an accurate story of what is depicted on the screen? Since the SLS screen reflects the true composite state of all alternate screens, the X server can discard all events that do not originate from the SLS screen. Simply eliminating events that originate from all alternate screens solves much of the complexity of event processing. Because alternate screen events are never placed on the X server's outbound event queue, the potential for a client to receive conflicting event notifications is eliminated. Based on these assumptions, the primary event processing routine in the SLS module can be expressed in the following pseudocode:

```
for each potential "event" on server's event
queue
{
    if event's originating screen is the SLS
screen
    {
        Deliver event to interested clients
    }
else
    {
        Discard event
    }
}
```

The method of discarding events from alternate screens works well, except in the case of Expose events. The X server generates an Expose event when an obscured window becomes visible or when an obscured portion of a window becomes visible. Expose events, however, can be more complex than other events because a single window action, such as moving, raising, or destroying a window, can cause the X server to generate multiple Expose events. Consider **Figure 6**, which contains a single screen with three windows.

If the user destroyed window C, the X server would generate Expose events to tell interested clients that windows A and B have new regions exposed and that those exposed regions need to be redrawn. The X server must also ensure that these Expose events are delivered to interested clients in a continuous fashion. That is, the Expose events resulting from a single window action must be delivered in a continuous event stream. The XExposeEvent structure has a count field that tells the client how many subsequent contiguous Expose events remain.

The requirement that related Expose events must be delivered in a single series posed the greatest problem for the

Figure 6

A single screen with windows A, B, and C.

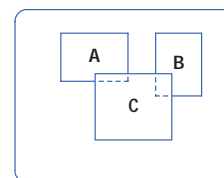
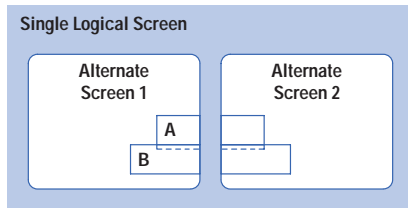


Figure 7

Window B obscures a portion of window A.



event handling component in the SLS module. To illustrate the problem, consider the two screens in **Figure 7** that are configured as a single logical screen.

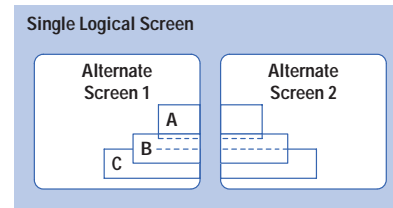
In this example, window B partially obscures window A. If window B is destroyed, the client needs to be notified so that it can issue requests to redraw the lower half of window A. There are two regions that need to be redrawn: the region of window A on alternate screen 1 and the region of window A on alternate screen 2. If we used the method of discarding events that originate from alternate screens, the client would not receive the exposures from each of the alternate screens.

This might lead one to conclude that the SLS module's event processing loop should ignore all events from alternate screens, except for Expose events, which should be passed straight through the client as if they were non-Expose events. However, this method would break the requirement that related Expose events be delivered contiguously.

If window C existed on alternate screens 1 and 2 and was also obscured by window B (**Figure 8**), the destruction of window B could cause Expose events to be generated in the sequence given in **Table I**.

Figure 8

Window B obscures portions of windows A and C.



This would result in four different Expose events referring to two different windows in an interleaving fashion. Instead, we would prefer to see something like the sequence in **Table II**.

As a result, the client receives two event bundles referencing windows A and C, and each bundle contains two Expose events denoting the regions on the particular window that need to be redrawn. This satisfies the continuity requirement for delivering Expose events.

Based on the need for a reordering method similar to that shown in Table II, the X server needs to use a different mechanism to process and deliver Expose events. If a typical user's environment only had two or three windows, the X server might have been able to function using hard-coded bookkeeping to track and reorder Expose events. Since most users commonly have substantially greater than a few windows (the Common Desk Environment [CDE] may use hundreds), a more general and robust method was needed. As a result, the SLS module uses a technique called *event coalescing* to reorder Expose events correctly into bundles corresponding to each window before the Expose events are delivered to the client.

Table I

Expose events generated from destroying window B in Figure 8.

Sequence Number	Alternate Screen	Window	Region (x1,y1,x2,y2)
0	1	A	(1000,600,1279,700)
1	1	C	(600,1000,1279,1100)
2	2	A	(1280,600,1580,700)
3	2	C	(1280,1000,1800,1100)

Table II

Desired Expose event sequence generated from destroying window B in Figure 8.

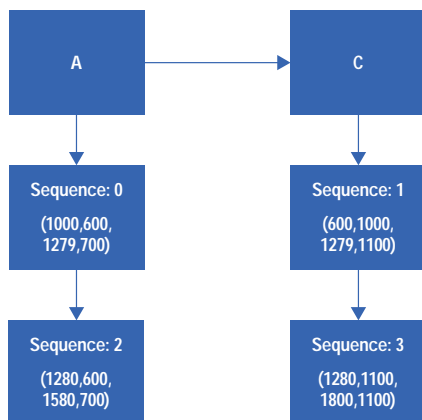
Sequence Number	Alternate Screen	Window	Region (x1,y1,x2,y2)
0	1	A	(1000,600,1279,700)
1	2	A	(1280,600,1580,700)
2	1	C	(600,1000,1279,1100)
3	2	C	(1280,1100,1800,1100)

Event coalescing is a generic method for temporarily removing events from the X server's event queue, reordering the events into related bundles, and then placing them back onto the event queue for later delivery to the client. To perform this reordering, the SLS module uses an internal linked list that corresponds to each window. When a newly Exposed region for a particular window is discovered, that region is simply added to the link corresponding to that window. From the configuration shown in **Figure 8** (and the event list in Table I), generating interleaved Expose events for windows A and C on alternate screens 1 and 2 would yield an event coalescing list like the one shown in **Figure 9**.

Therefore, at the appropriate time, we can easily arrange the Expose events for window A and send a two-event bundle corresponding to window A to the client. The same applies to window C.

Figure 9

A coalescing list for the exposed windows in Figure 8.



Now that we have rearranged the Expose events into easily deliverable bundles, the X server needs to determine the appropriate time to deliver these bundles to the client. As with other events, we deliver the repackaged Expose event when the SLS screen, rather than an alternate screen, receives the Expose event. When the SLS screen receives an Expose event for a particular window, we find the events packaged for that window by looking for that window in our list, delivering the event bundle to the client, and then deleting these events from our event coalescing linked list.

Using this information, we can extend our original event-processing component of the SLS module into the following pseudocode:

```
for each potential "event" on server's event queue
{
    if event is an Expose event
    {
        if event originated from an alternate screen
        {
            add event to the list for its corresponding window
        }
        else if event originated from SLS screen
        {
            Deliver "saved" events for that window to interested clients.
            Delete those events from internal list
        }
        continue;
    }
    if event's originating screen is the SLS screen
    {
        Deliver event to interested clients
    }
}
```

```

    }
    else
    {
        Discard event
    }
}

```

By using this event-processing mechanism, the SLS module can cleanly deliver events to clients in an expected fashion and without conflicting results.

Input Considerations

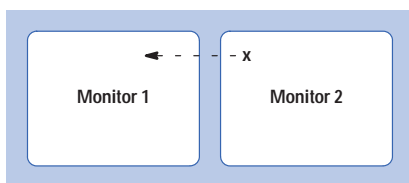
Although much of the complexity of the SLS module involves managing display output (that is, keeping track of which windows or portions thereof appear on which physical screens), there are also some special considerations that must be made for input. The most important aspect of input event handling is that the user sees a smooth motion when moving the pointer. The pointer needs to move from one screen to another when it crosses monitor boundaries, and this transition should be handled smoothly. For example, consider the configuration in **Figure 10**.

If the pointer is located in the upper left corner of monitor 2, and we move the pointer to the left, the pointer is moved from monitor 2 to monitor 1. The SLS module needs to ensure that the pointer moves smoothly across the monitor boundaries rather than jumping across monitors. This is especially important when the user drags a window from one monitor to another, since a jumping or pointer-snap effect will cause the entire window to jump.

In addition, if the pointer comes to rest at a location that corresponds to a physical monitor boundary, we want the pointer to be present on both screens, rather than choosing one screen or the other to display the pointer. To do this, we employ the concept of pointer sensitivity along

Figure 10

Moving the pointer between monitors.



the edges of the physical monitors. When the pointer is within this sensitive range along the edge of a monitor, the X server displays the pointer on the neighboring screen and the current screen. In **Figure 10**, as the pointer moves from monitor 2 to monitor 1, there will be a time when the head of the pointer is displayed on monitor 1, and the tail of the pointer appears on monitor 2. This maintains the look and feel of the single logical screen concept.

To implement this, we check the new location coordinates for our pointer on each screen and turn the pointer on or off for that screen, depending on whether the pointer should be visible on that screen. Since we use the DDX driver's pointer rendering routines, we let the individual DDX driver perform the work of clipping against physical screen boundaries. Therefore, the SLS module does not need to clip the pointer for the case in which the pointer falls on adjacent screens. Not only does this implementation method simplify pointer management code within the SLS module, but it also easily generalizes to the case of a four-headed 2-by-2 SLS configuration, in which some portion of the pointer could physically appear on all four screens. Based on this description, the following pseudo-code handles most of the code in the SLS module that tracks pointer position:

```

/* x & y are the x- and y-coordinates of
 * location to move the pointer.
 * sensitivity_x & sensitivity_y are the
 * x and y coordinates of our pointer
 * sensitivity bounding box.
 */
for each "alternate screen"
{
    if (x,y) is on "alternate screen" or is
    in the region formed by (sensitivity_x,
    sensitivity_y)
    {
        Displaypointer (screen, x, y);
    }
    else
    {
        TurnOffpointer (screen);
    }
}

```

The question comes up as to how the X server handles pointer movement for the case of a multiscreen configuration like that shown in **Figure 11**, which has four physical monitors configured as two SLS screens. In this scenario

Figure 11

Two 1-by-2 SLS configurations combined via multiscreen.



the SLS module handles the pointer movement, as previously described, for each of the two SLS screens. Therefore, pointer motion within each SLS screen will have a smooth and even movement across the physical screens. When moving from one SLS to the other SLS, however, the pointer motion will not be as smooth because we do not try to accomplish this smooth pointer motion in a multiscreen environment.

If the pointer in **Figure 11** moves from the left SLS screen to the right SLS screen, the pointer on the left SLS screen will be turned off, and it will be turned on in the right SLS screen. The user would not, however, see a case where the pointer's head appears on the left screen of the right SLS screen and the pointer's tail on the right screen of the left SLS screen. Since the X server simply turns off the pointer on the left SLS screen and turns it on in the right SLS screen, the pointer transition will not appear smooth. This occurs because pointer motion between (logical) screens in a multiscreen configuration is not handled in the SLS module but rather in the DDX input code.

Configuration

The HP-UX system administration manager (SAM) supports advanced display configuration on HP-UX 10.0 systems and beyond. This HP-UX GUI automates many different administration tasks and can run under X Windows or in terminal mode. Users are required to have root permission to execute the SAM command.

A prerequisite for an advanced display configuration is the availability of multiple graphics display devices. Although some utilities exist that will identify graphics devices, the best way to identify the number and type of graphics devices in a system is through SAM. To do this, invoke SAM and look for either the Display or X Server Configuration icons. On patched 10.20 systems and beyond,

the X Server Configuration icon has been moved to under the Display folder shown in **Figure 12**.

Clicking on the Display folder icon will produce the window shown in **Figure 13**. The Monitor Configuration icon allows changing the screen's (frame buffer) resolution, refresh rate, timing standard, hardware double buffering, and quad buffer stereo operation without requiring the user to reboot. The X Server Configuration icon is used to configure the HP-UX X server or simply identify which graphics devices have been installed and are available. It is this subarea within SAM that enables the advanced display configurations.

Note that on systems containing only one graphics device, it will be impossible to set up an advanced display configuration. It is possible, however, to install additional graphics devices and have an advanced display configuration. The graphics configuration tool GUI will provide insight into what graphics are supported on HP-UX systems and where they can be plugged in.

As shown in **Figure 14**, the X Server Configuration window graphically illustrates the current X server configuration. In this case system xtc112 contains two HP VISUALIZE-EG graphics devices. The first is an internal graphics device (slot 0) and is configured, meaning that the X server is currently using it. The second graphics device is plugged into slot 1 and is unconfigured, as represented by the grayed icon.

Just below the menu bar are two status lines. The first status line is used to convey critical screen information. The first two words identify if a user is configuring a print server or a video server. For most users this will always be a video server. The DISPLAY string is the environment variable users must set for X clients to talk to this X server. The X Configuration File string identifies which configuration

Figure 12

Main SAM window.

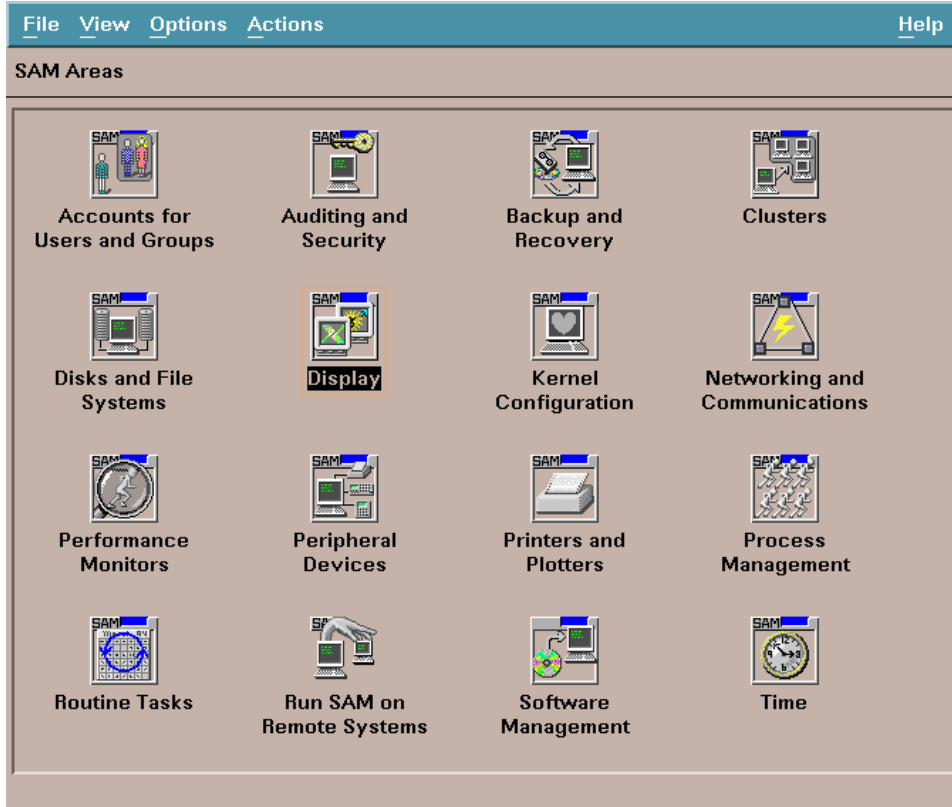


Figure 13

The Display window.

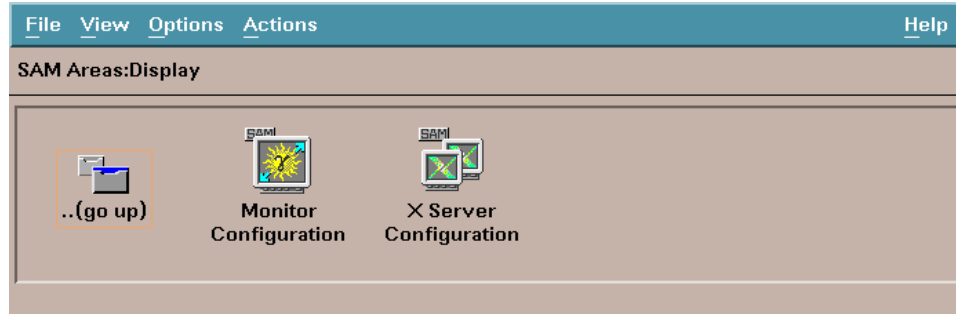
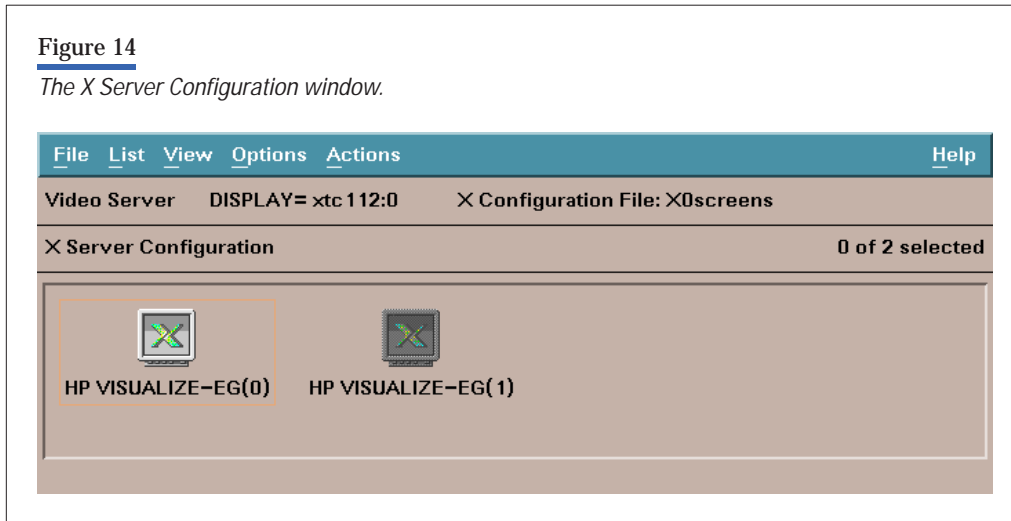


Figure 14

The X Server Configuration window.



file is being used. For example, if the default X0screens file is being used, the hostname is xtc112, and if the ksh shell is being used, then a user would need to type: export DISPLAY=xtc112:0 to run an X client on that display.

The major portion of the X Server Configuration window is used for screen icons. There are three types of screen icons: configured, unconfigured, and SLS (see **Figure 15**). Each icon represents one screen that maps directly to a physical graphics device (or devices) such as a graphics card or the internal onboard graphics devices (or a combination of devices for SLS). Each screen can be configured or unconfigured. The configured screens will always be sorted by screen position in a left-to-right fashion. For example, if there are two configured screens, the left-most icon is assigned position: 0.0, and the second screen is given position :0.1. Unconfigured screens are represented with grayed icons that indicate that the X server will not use them. The icon for a single logical screen shows the X spanning multiple monitors and is used to represent screens made up of multiple graphics devices.

Actions

The menu items in the Actions menu contain the functionality for changing the configuration of the X server.

Actions will be active or grayed out depending on which screens have been chosen. Users can select multiple screens via CTRL-select. Preselecting configured and unconfigured screens will result in only the global actions Describe Screen and Identify Screen being active. SLS icons are in the same classification as configured icons. Unconfiguring an SLS screen is similar to breaking the components into individual screens (which will remain individually configured). Removing a screen from configuration on an SLS icon will dissolve the SLS object into its components and return it to an unconfigured state.

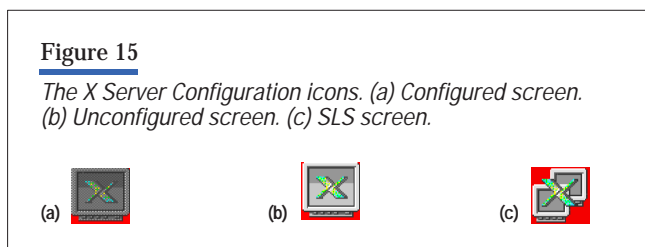
Multiscreen Configuration

There are several ways to create a multiscreen configuration. The simplest is to select an unconfigured screen and add it to the configuration. The Add action causes the selected screen to be appended to the existing list of configured screens.

To control screen positioning, users can invoke the Modify Multi-Screen Layout... action. When invoking this action, if any unconfigured screens have been preselected, these screens will become configured before displaying the layout dialog window. If fewer than two configured screens exist, an error message is displayed instructing the user in properly invoking the layout.

Figure 15

The X Server Configuration icons. (a) Configured screen. (b) Unconfigured screen. (c) SLS screen.



Any modifications to the X server configuration will take effect after the configuration has been saved and the X server has been restarted.

Identify Screen

There is no automated method of identifying whether the monitors are positioned left-to-right, top-to-bottom, or whether there is a monitor connected at all. This was the first real problem encountered in providing a GUI for advanced display configurations. For single-screen systems this is not much of a problem. Of course, if the monitor is not connected properly, nothing will be visible. On systems with multiple screens, users might not know which monitor they are looking at. How can a user identify which monitor is connected to which graphics device? When configuring an advanced display, it is important that users know where a screen is positioned, otherwise their layout will be incorrect.

On HP-UX systems a unique device file represents each graphics device. No physical relationship exists between the mapping of physical monitor placement to device files, nor is there any direct correlation between the position of the graphics device (for example, which slot it is located in) with the major and minor numbers of the device file. Thus, even if the user were to trace monitor cables from the graphics cards to the actual monitors, a cumbersome task, nothing would be learned about which device file maps to which monitor. Since there is no automated manner to identify this information, we provided the user with the ability to identify this information via a point and click interface that identifies a particular screen's monitor.

In the X Server Configuration window (see **Figure 14**), the Identify Screen action calls a procedure to repeatedly turn off and on the video output of the graphics cards for the selected screens. This action causes the monitor connected to the graphics device to blink. In this way users can easily identify which monitor maps to which physical graphics device.

The procedures that cause the screen to blink are located in the shared library of each DDX driver and are independent of our SAM component. In this way we have enabled SAM to support new drivers (or patched drivers) without changing the SAM executable. This is part of the plug-and-play functionality that enables the X server to support new hardware without changing existing libraries.

Single Logical Screen Configuration

The single logical screen setup is very similar to the multi-screen configuration. Users select those screens they want to combine into a single logical screen and then select the Create SLS... action from within the Single Logical Screen submenu.

An error message will appear if the user attempts to combine two or more incompatible screens. Warning messages are issued if the various screen options of the component screens are different in any way.

The Create SLS... action invokes the layout dialog. When a user accepts a single logical screen layout, the individual component screen icons are replaced with the SLS icon. The SLS screen will then assume the position of its lowest component. **Figure 16** shows what this icon would like if we combined the two HP VISUALIZE-EG screens in **Figure 14** into a single logical screen using a horizontal layout.

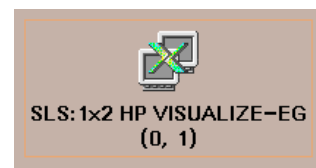
The SLS icon naming convention conveys information about its internal layout. The 1x2 in the icon signifies that the screens are horizontal, and the parenthetical expression (0,1) indicates the ordering with the internal card (slot 0) on the left and the external card (in slot 1) on the right. As with the multiscreen setup, any modifications will take effect after the configuration has been saved and the X server has been restarted.

SLSclone Configuration

The SLSclone configuration begins with an SLS configuration. After the SLS configuration is operational, a user then runs a program to change between SLS and SLSclone dynamically. This program is shipping with the latest X server patch (PHSS_12462) and is called SLSclone.

Figure 16

The icon that results from combining two HP VISUALIZE-EG screens.



Conclusion

This article has described the implementation of our advanced display technologies for HP-UX workstations. Initialization of a single logical screen (SLS) or SLSclone configuration includes the creation of screen structures for each of the component screens. These screens are called alternate screens. The SLS driver replicates appropriate requests directed at the SLS screen to these alternate screens, adjusting the offsets as appropriate to achieve the desired layout. Rendering to alternate screens is accomplished through existing DDX libraries, providing flexibility in allowing us to support DDX under SLS drivers without modifying our SLS driver.

We have also shown how event management was modified and discussed input considerations arising from our handling of events. Finally, we have briefly shown how the HP-UX SAM utility can be used to easily configure multiple graphics devices. For additional information on advanced display configurations and other X server features, visit our web site at:

<http://www.hp.com/go/xwindow>

References

1. T. Spencer, P. Anderson, and D. Sweetser, "Advanced Display Technologies on HP-UX Workstations," *Hewlett-Packard Journal*, Vol. 49, no. 2, May 1998.

-
- ▶ [Go to Subarticle 7b](#)
 - ▶ [Return to Article 7](#)
 - ▶ [Go to Next Article](#)
 - ▶ [Go to Journal Home Page](#)