1
XView and the X Window System

The XView Toolkit allows a programmer to build the interface to an application without having to learn many of the details of the underlying windowing system. However, it is valuable to have some understanding of X before attempting to build applications under XView. This chapter introduces many of the most important concepts on which the X Window System is based and describes the computing environment for X applications. It also describes the role of the XView Toolkit in the X Window System.

For the most part, this chapter assumes that you are new to programming the X Window System. This chapter describes the basics of the X Window System—further details will be described as necessary later in the manual. However, this book does not repeat the detailed description of Xlib programming found in Volume One, *Xlib Programming Manual*. If you already have some experience programming the X Window System, you might wish to begin at Chapter 2, *The XView Programmer's Model*.

1.1 The X Window System

X controls a bit-mapped display in which every pixel (dot on the screen) is individually controllable. This allows drawing of pictures in addition to text. Until recently, individual control of screen pixels was widely available only on personal computers (PCs) and high-priced technical workstations, while more general-purpose machines were limited to output on text-only terminals. X brings the same world of graphic output to both PCs and more powerful machines. Figure 1-1 shows an X application in comparison with a traditional text terminal.

Like other windowing systems, X divides the screen into multiple input and output areas called windows, each of which can act as an independent *virtual terminal*. Using a terminal emulator, windows can run ordinary text-based applications. However, windows can also run applications designed to take advantage of the graphic power of the bitmapped display.

X takes user input from a *pointer*, which is usually a mouse but could just as well be a trackball or tablet. The pointer allows the user to point at certain graphics on the screen and use the buttons on the mouse to control a program without using the keyboard. This method of using programs is often easier to learn than traditional keyboard control, because it is more intuitive. Figure 1-2 shows a typical pointer being used to select a menu item.

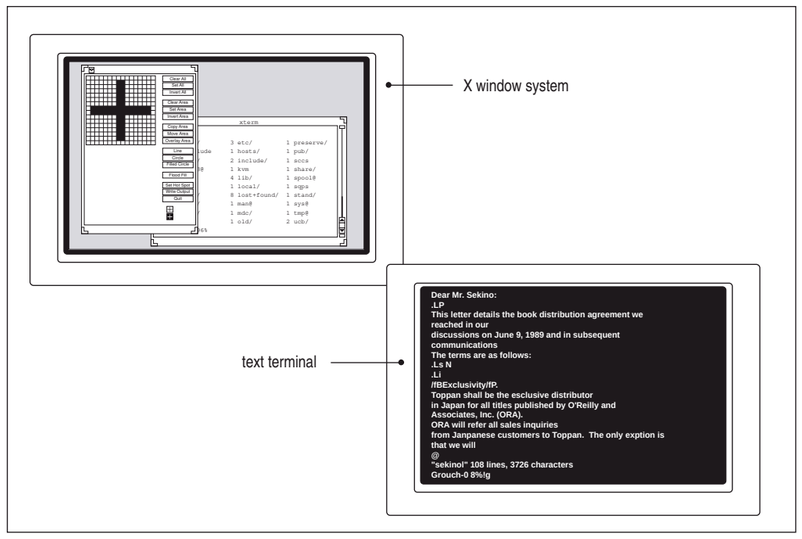

Figure 1-1. An X application and a traditional text terminal

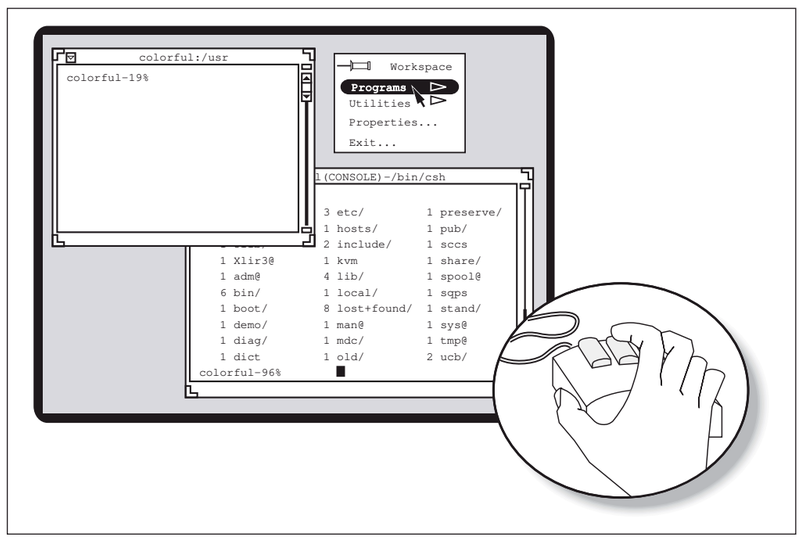

Figure 1-2. Selecting a menu item with the pointer

Of course, X also handles keyboard input. The pointer is used to direct the input focus of the keyboard (often called the *keyboard focus*) from window to window with only one application at a time able to receive keyboard input.

In X, as in many other window systems, each application need not (and in fact usually does not) consist of only a single window. Any part of an application can have its own separate window, simplifying the management of input and output within the application code. Such *child* windows are only visible within the confines of their parent window. Each window has its own coordinate system where the origin is the upper-left corner of the window inside its border. In basic X, windows are rectangular and oriented along the same axes as the edges of the display. The application or the user can change the dimensions of windows.

Many of the above characteristics are also true of several other window systems. What has made X a standard is that X is based on a network protocol—a predefined set of requests and replies—instead of system-specific procedure calls. The *X Protocol* can be implemented for different computer architectures and operating systems, making X device-independent. Another advantage of a network-based windowing system is that programs can run on one architecture while displaying on another. Because of this unique design, the X Window System can make a network of different computers cooperate. For example, a computationally-intensive application might run on a supercomputer but take input from and display output on a workstation across a network.

1.1.1 The Server and Client

To allow programs to be run on one machine and display on another, X was designed as a network protocol between two processes, one of which is an application program called a *client*, and the other, the *server*. The server is a resource server, controlling a user's resources (such as the display hardware, keyboard, and pointer) and making these resources available to user applications. In other words, the X server isolates the device-specific code from the application.

The server performs the following tasks:

- Allows access to the display by multiple clients. The server can deny access from clients running on certain machines.
- Interprets network messages from clients and acts on them. These messages are known as requests. Some requests command the server to do two-dimensional drawing or move windows, while others ask the server for information.
- Passes user input to clients sending network messages known as events, which represent key or button presses, pointer motion, and so forth. Events are generated asynchronously, and events from different devices might be intermingled. The server must de-multiplex the event stream and pass the appropriate events to each client.
- Maintains complex data structures, including windows and fonts, so that the server can perform its tasks efficiently. Clients refer to these abstractions by ID numbers. Server-maintained abstractions reduce the amount of data that has to be maintained by each client and the amount of data that has to be transferred over the network.

In X, the term *display* is often used as a synonym for server, as is the combined term *display server*. However, the terms *display* and *screen* are not synonymous. A screen is the actual piece of hardware on which the graphics are drawn. Both color and monochrome displays are supported. A server might control more than one screen. For example, one server might control a color screen and a monochrome screen for a user who wants to be able to debug an application on both types of screens without leaving his or her seat.

The communication path between a client and the server is called a *connection*. Several clients may be connected to a single server. Clients may run on the same machine as the server if that machine supports multitasking, or clients may run on other machines in the network. In either case, the X Protocol is used by the client to send requests to draw graphics or to query the server for information; it is used by the server to send user input or replies to requests back to the client. All communication from the client to the server and from the server to the client takes place using the X Protocol.*

It is common for a user to have programs running on several different hosts in the network, all invoked from and displaying their windows on a single screen (see Figure 1-3). Clients running remotely can be started from the remote machine or from the local machine using the network utilities *rlogin* or *rsh*.

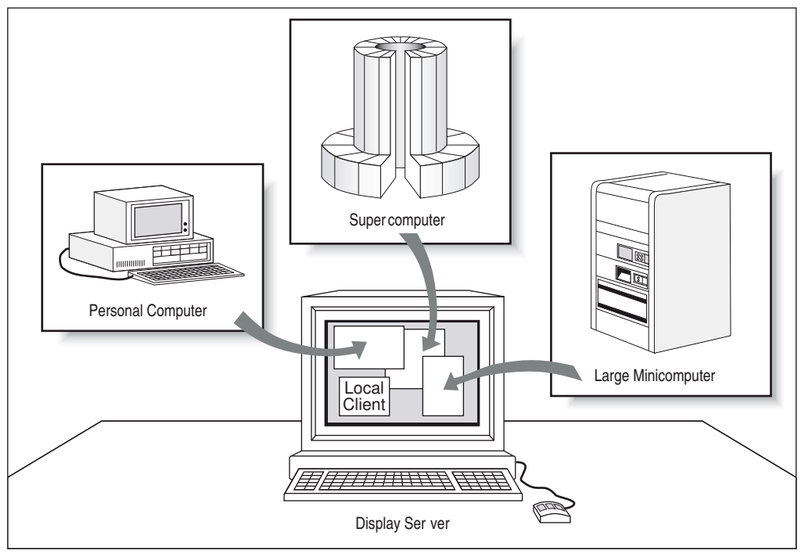

Figure 1-3. Applications can run on any system across the network

*The X Protocol is independent of the networking hardware and runs on top of any network that provides point-to-point packet communication. TCP/IP and DECnet are the only networks currently supported. For more information about the X Protocol, see Volume Zero, *The X Protocol Reference Manual*.

This use of the network is known as *distributed processing*. The most important application of this concept is to provide graphic output for powerful systems that cannot have built-in graphics capabilities. However, distributed processing can also help solve the problem of unbalanced system loads. When one host machine is overloaded, the users running clients on that machine can arrange for programs to run on other hosts.

1.2 The Software Hierarchy

There are many different ways to write X applications because X is not restricted to a single language, operating system, or user interface. The only requirement of an X application is that it generate and receive X Protocol messages.

Figure 1-4 shows the layering of software in an X application. Xlib is the lowest-level C language interface to X. The main task of Xlib is to translate C data structures and procedures into X Protocol events; it then sends them off and receives protocol packets in return that are unpacked into C data structures. Xlib provides full access to the capabilities of the X Protocol but does little to make programming easier. It handles the interface between an application and the network and includes some optimizations that encourage efficient network usage. The list of functions that Xlib performs is so extensive that if the programmer were responsible for handling all these pieces directly, application programs would be too large and prone to performance degradation and potential bugginess. For this reason, *toolkits* are used to modularize the more common functions that handle the user interface portion of an application.

XView is one of a half-dozen or so toolkits available for the X Window System. If you are familiar with other toolkits, you will recognize that the XView Toolkit is equivalent to the Xt Intrinsics and a widget set. Like the Intrinsics, XView is built upon Xlib. It is an object-oriented toolkit that provides reusable, configurable user interface components, equivalent to widgets.*

Toolkits handle many things for the programmer. They provide a framework for combining pre-built user interface components with application-specific code. For example, if the application needs to prompt the user for a filename, a toolkit should provide a component (a *command frame*) that is functionally capable of displaying the query to the user and providing the user's response to the application.

Any user interface component also needs to manage the interpretation of events delivered from the window system. When events are generated, the toolkit decides whether or not to propagate the event to the application or to use it for its own internal purposes. To continue the example, when the user types a filename in the command frame, events are generated in which the interface must decide whether that object should interpret the input or whether it should be sent to the application. A toolkit thus comprises a mechanism to dispatch events and a set of prebuilt interface objects that define the look and feel of an application.

*Widget sets are sometimes loosely referred to as toolkits. However, a toolkit comprises the functions of the Xt Intrinsics layer and one widget set (e.g., the Athena widget set). There are several different widget sets from various vendors that are designed to work with Xt. For more information on Xt Intrinsics-based toolkits, see Volume Four, *X Toolkit Intrinsics Programming Manual*.

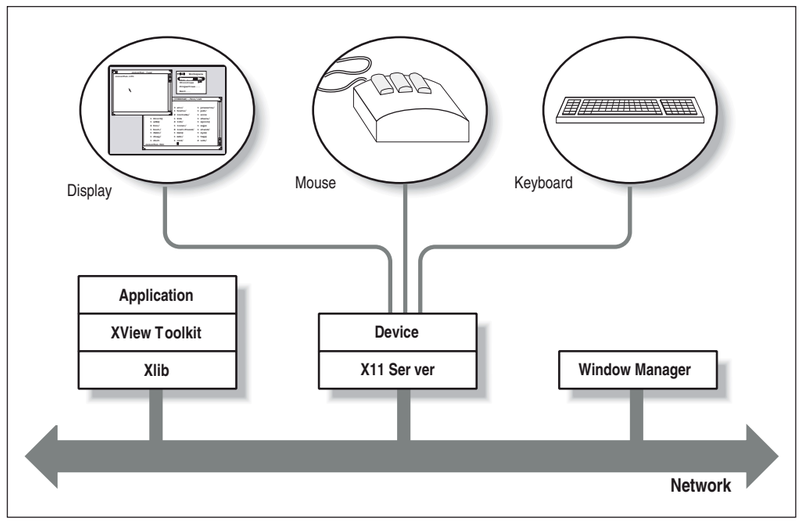

Figure 1-4. The software architecture of X applications

Note that using a toolkit does not preclude calling Xlib directly to accomplish certain tasks such as drawing. In XView, graphics rendering is done most efficiently by using Xlib drawing routines, for instance.

1.3 Extensions to X

Another thing to know about X is that it is *extensible*. The code includes a defined mechanism for incorporating extensions, so that vendors are not forced to modify the existing system in incompatible ways when adding features. An extension requires an additional piece of software on the server side and an additional library at the same level as Xlib on the client side. After an initial query to see whether the server portion of the extension software is installed, these extensions are used just like Xlib routines and perform at the same level.

Among the extensions currently being developed are support for 2D spline curves, for 3D graphics, and for Display PostScript™. These extensions can be used in toolkit applications just like Xlib can.

1.4 The Window Manager

Because multiple applications can be running simultaneously, rules must exist for arbitrating conflicting demands for input. For example, does keyboard input automatically go to whichever window the pointer is in, or must the user explicitly select a window for keyboard input?

Unlike most window systems, X itself makes no rules about this kind of thing. Instead, there is a special client called the *window manager* that manages the positions and sizes of the main windows of applications on a server's display. The window manager is just another client, but by convention, it is given special responsibility to mediate competing demands for the physical resources of a display including screen space, color resources, and the keyboard. The window manager allows the user to move windows around on the screen, resize them, and usually start new applications. The window manager also defines much of the visible behavior of the window system, such as whether windows are allowed to overlap or are tiled (side by side), and whether the keyboard focus simply follows the pointer from window to window or whether the user must click a pointer button in a window to change the keyboard focus (click-to-type).

Applications are required to give the window manager certain information that helps the window manager mediate competing demands for screen space or other resources. For example, an application specifies its preferred size and size increments. These are known as *window manager hints* because the window manager is not required to honor them. The XView Toolkit provides an easy way for applications to set the window manager hints.

The conventions for interaction with the window manager and with other clients have been standardized by the X Consortium as of July 1989 in a manual called the *Inter-Client Communications Conventions Manual* (ICCCM, for short). The ICCCM provides basic policy intentionally omitted from X itself, such as the rules for transferring data between applications (selections), transfer of keyboard focus, layout schemes, colormap installation, and so on. As long as applications and window managers follow the conventions set out in the ICCCM, applications created with different toolkits will be able to coexist and work together on the same server. Toolkit applications should be immune to the effects of the change from earlier conventions.

1.5 Handling Events

The window that X provides is the connection between the XView application and the X server. The reason X windows are important to XView is that these windows are the input targets for the user's focus. They are the actual objects that get events from the user and pass the events through to the XView world.

An X *event* is a data structure sent by the server that describes something that just happened that is of interest to the application. The sources of events are the user's input, the window system, the operating system, and the application programs. For example, the user's pressing a key on the keyboard or clicking a mouse button generates an event, and a window's being moved on the screen also generates events if it changes the visible portions of other applications. It is the server's job to distribute events to the various windows on the screen.

In XView, between the server and the application, there is an event dispatch mechanism called the Notifier, as shown in Figure 1-5.

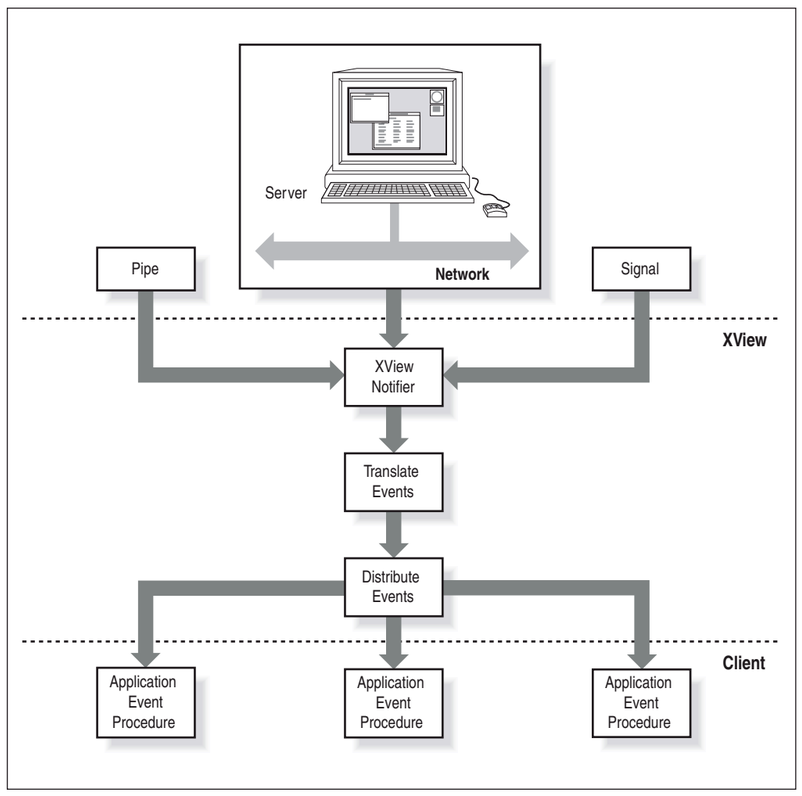

Figure 1-5. The Notifier exists between the server and the XView application

After the set-up phase of the application, where you create XView objects such as buttons or scrollbars and determine how they will interact with your application code, you have several choices for input distribution. The simplest method is to hand off control of your application to XView. From then on, the XView Notifier automatically distributes events to the objects created by the application. These objects process many events internally so that your application does not need to get involved.

The key point is that your application is only told about events for which it specifically requested to be notified. By responding to these events, the application can perform its tasks. For example, if the user types the letter *A* in a window, X will pass the event to XView, which

in turn can pass it to the application. An application's event handler can interpret this event and display the letter typed in the window. Finally, control returns to the top level so the next event can be read. This is a typical cycle of events that happens for each event generated by the user.

If the application created a scrollbar, then it would track certain events, such as when the scroll button is pressed. XView actually sends a request to the X server to create the X window that will become part of the scrollbar object. In the application's request, it can ask to be notified about or to ignore certain user events in the X window created.

The window system dictates which window gets an event. If the window currently has the keyboard focus and that window does not want to process the event, it has the option of throwing away the event or dropping it to the window below it. This is usually the parent of the window. For instance, if a panel item gets a keyboard press (the user typed an *A*) and if the panel item does not want to deal with the event, then the panel item might be configured so that the event is passed to the item's parent: the panel itself.

The Notifier does not just do input distribution; it also allows selection of different input sources. In addition to handling window system events, your application can also handle a number of interrupts that might be generated by the operating system. Your application can respond to signals, input on a file descriptor and interval timers. You could also pass events between clients in the same process, interpose on a Notifier client to change its behavior, and receive notification on the death of a child process which you have spawned. The use of some of these input sources requires you to call the Notifier directly. The Notifier is covered again in the next two chapters as well as in Chapter 20, *The Notifier*.

1.6 Development of the XView Toolkit

Over the years, Sun Microsystems, Inc. has developed several toolkits, each more well-defined, more functional, and aesthetically superior than the last. SunView 1 was perhaps the first well-accepted user interface toolkit Sun provided. It had all the basic elements necessary to make a functional user interface—including a well-defined API (application programmer's interface). It introduced the attribute-value interface, which we'll examine in more detail in Chapter 2, *The XView Programmer's Model*. It has very few procedure calls. For instance, you use a single function to create all user interface objects. You have the option of using default values, in which case the object is created with only a few lines of code, or of setting the values of specific attributes as required. These attributes can be set at the time the object is created, or later on, by using a different function.

The SunView 1 Toolkit was based on SunWindows™, a kernel-based window system. It required that a single computer control both the application and the user's display and keyboard. The X Window System represents a new generation of window systems. It is server-based, which means that the client application can run on a different system than the server that controls the display.

Today there are several thousand SunView applications, and one of the aims of XView is to make it easy to bring those applications to the X Window System marketplace. In addition,

Sun has made the source code to the XView Toolkit freely available. It will be shipped as part of the standard MIT X distribution as well as with UNIX System V Release 4.

XView provides a set of windows that include:

- *Canvases* on which programs can draw.
- *Text subwindows* with built-in editing capabilities.
- *Panels* containing items such as buttons, choice items, and sliders.
- *TTY subwindows* that emulate character-based terminals.

These windows are arranged as *subwindows* within *frames*, which are themselves windows. Frames can be transitory or permanent. Transient interactions with the user can also take place in *menus* which can pop up anywhere on the screen. We will look more at all XView objects when we cover the XView programming model in the next chapter.

1.7 Versions of the XView Toolkit

Since XView was first released, many applications have been developed for XView and many others have been ported from SunView, as well as from other software platforms. XView development has continued at Sun; new packages have been written, extensions have been added, and the existing packages have been improved based on user's needs. This manual is written for the latest release of XView, Version 3, and it includes descriptions for all the important improvements available in this XView release.

1.8 OPEN LOOK Graphical User Interface

An important feature of the XView Toolkit is that it implements the OPEN LOOK Graphical User Interface (GUI). The OPEN LOOK GUI aims to provide users with a simple, consistent, and efficient interface. An example of an OPEN LOOK workspace is shown in Figure 1-6. OPEN LOOK is supported by Sun and AT&T as the graphical user interface standard for System V Release 4. Users and developers benefit from a standard because it ensures consistent behavior across a number of diverse applications. Programmers can concentrate on the design of the application without having to "invent" a user interface.

A well-defined user interface should be generalized enough so that it can be implemented on any operating system, windowing system, or graphics display. Because OPEN LOOK is not bound by any of these constraints, XView was built based entirely on specifications that could be mapped easily into the X Window System.

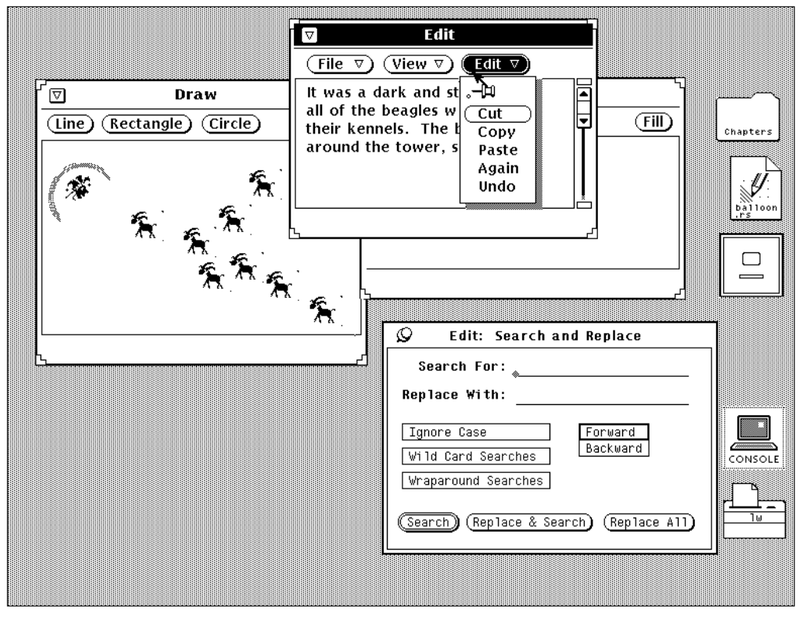

Figure 1-6. A sample OPEN LOOK workspace

The visual design of OPEN LOOK is restrained. The design of each component in the user interface is simple and uncomplicated. The interface is based on a few simple concepts that result in a system that is easy to learn initially. And an XView application is relatively simple and easy to implement because many of the default values of interface components are adequate for most applications.

We will attempt to summarize the OPEN LOOK interface where appropriate in this document. However, the definitive document on OPEN LOOK for application programmers is the *OPEN LOOK Graphical User Interface Style Guide*. This book provides guidelines for developers on using the elements of the OPEN LOOK GUI in applications. Where appropriate, we will show examples, such as Figure 1-6, which are taken from the *OPEN LOOK GUI Specification Guide*.