20
The Notifier

In this chapter, we look at the Notifier in greater detail, discussing its role in processing events for an XView application. This chapter serves as an introduction to the Notifier and covers its use for most applications. You should be familiar with the topics covered in Chapter 6, *Handling Input*, before you read this chapter.

20.1 Basic Concepts

The Notifier maintains the flow of control in an application. To understand the basic concepts of the Notifier, we must distinguish between two different styles of input handling, *mainline* and *event-driven* input, and consider how they affect where the flow of control resides within a program.

20.1.1 Mainline Input Handling

The traditional type of input handling of most text-based applications is mainline-based and input-driven. The flow of control resides in the main routine and the program *blocks* when it expects input. That is to say, no other portion of the program may be executed while the program is waiting for input. For example, in a mainline-driven application, a C programmer will use `fgets()` or `getchar()` to wait for characters that the user types. Based on the user's input, the program chooses an action to take. Sometimes, that action requires more input, so the application calls `getchar()` again. The program does not return to the main routine until the processing for the current input is done.

The tight control represented by this form of input handling is the easiest to program since you have control at all times over what to expect from the user and you can control the direction that the application takes. There is only one source of input—the keyboard—and the user can only respond to one interface element at a time. A user's responses are predictable in the sense that you know that the user is going to type *something*, even if you do not know what it is.

20.1.2 Event-driven Input Handling

Windowing systems are designed such that many sources of input are available to the user at any given time. In addition to the keyboard, there are other input devices, such as the mouse. Each keystroke and mouse movement causes an *event* that the application might consider. These keystroke and mouse events are generated from the window system. Further, there are other types of events that are generated from the window system itself and from other processes. Another aspect of event-driven input handling is that you are not guaranteed to have any predictable sequence of events from the user. That is, a user can position the mouse on an object that receives text as input. Before the user is done typing, the user can move the mouse to another window and select a panel button of some sort. The application cannot (and should not) expect the user to type in window *A* first, then move to window *B* and select the button. A well-written program should expect input from any window to happen at any time.

20.2 Functions of the Notifier

The Notifier can do any of the following:

- Handle *software interrupts*—specifically, UNIX signals such as `SIGINT` or `SIGCONT`.
- Notice state changes in processes that your process has spawned (e.g., a child process that has died).
- Read and write through file descriptors (e.g., files, pipes, and sockets).
- Receive notification of the expiration of timers so that you can regularly flash a caret or display animation.
- Extend, modify, or monitor XView Notifier clients (e.g., noticing when a frame is opened, closed, or about to be destroyed.)
- Use a non-notification-based control structure while running under XView (e.g., porting programs to XView).

The Notifier also has provisions, to a limited degree, to allow programs to run in the Notifier environment without inverting their control structure.

20.3 How the Notifier Works

Up until now, we have been saying that you should register an event handler for objects when they want to be notified of certain events such as mouse motion or selection or keyboard input. What you may not have been aware of is that you are indirectly registering these event handlers with the Notifier. When you specify *callbacks* or *notify procedures*, the XView object specified is said to be the *client* of the Notifier. Look at the following code:

```
extern void my_event_handler();

xv_set(canvas,
    CANVAS_PAINTWINDOW_ATTRS,
        WIN_CONSUME_X_EVENT_MASK,  ButtonPressMask | KeyPressMask,
        WIN_EVENT_PROC,            my_event_handler,
        NULL,
    NULL);
```

In the above code, each paint window of the canvas becomes a client of the Notifier.*

Generally stated, the Notifier detects events in which its clients have expressed an interest and dispatches these events to the proper clients in a predictable order. In the X Window System, events are delivered to the application by the X server. In XView, it is the Notifier that receives the events from the server and dispatches them to its clients. After the client's notify procedure processes the event, control is returned to the Notifier.

20.3.1 Restrictions

The Notifier imposes some restrictions on its clients. Designers should be aware of these restrictions when developing software to work in the Notifier environment. These restrictions exist so that the application and the Notifier do not interfere with each other. More precisely, since the Notifier is multiplexing access to user process resources, the application needs to respect this effort so as not to violate the sharing mechanism.

For example, a client should not call `signal` (3). The Notifier is catching signals on behalf of its clients. If a client sets up its own signal handler, then the Notifier will never notice the signal. The program should call `notify_set_signal_func()` instead of `signal` (3) (see Section 20.5, "Signal Handling").

*`CANVAS_PAINTWINDOW_ATTRS` tells the `CANVAS` package to register the input mask and callback routine for each of its paint windows.

20.3.1.1 System calls to avoid

Assuming an environment with multiple clients and an unknown Notifier usage pattern, you should not use any of the following system calls or C library routines:

`signal` (3) — The Notifier is catching signals on behalf of its clients. If you set up your own signal handler over the one that the Notifier has set up, then the Notifier will never notice the signal.

`sigvec` (2) — The same applies for `sigvec` (2) as for `signal` (3) above.

`sigaction` (2) — The same applies for `sigaction` (2) as for `signal` (3) above.

`setitimer` (2) — The Notifier is managing two of the process's interval timers on behalf of its many clients. If you access an interval timer directly, the Notifier could miss a timeout. Use `notify_set_itimer_func()` instead of `setitimer` (2).

`alarm` (3) — Because `alarm` (3) sets the process's interval timer directly, the same applies here as for `setitimer` (2) above.

`getitimer` (2) — When using a Notifier-managed interval timer, you should call `notify_itimer_value()` to get its current status. Otherwise, you can get inaccurate results.

`wait3` (2) — The Notifier notices child process state changes on behalf of its clients. If you do your own `wait3` (2), then the Notifier may never notice the change in a child process or you may get a change of state for a child process in which you have no interest. Use `notify_set_wait3_func()` instead of `wait3` (2).

`wait` (2) — The same applies for `wait` (2) as does for `wait3` (2) above.

`ioctl(2) (..., FIONBIO, ...)` — This call sets the blocking status of a file descriptor. The Notifier needs to know the blocking status of a file descriptor in order to determine if there is activity on it. `fcntl` (2) has an analogous request that should be used instead of `ioctl` (2).

`ioctl(2) (..., FIOASYNC, ...)` — This call controls a file descriptor's asynchronous IO (input/output) mode setting. The Notifier needs to know this mode in order to determine if there is activity on it. `fcntl` (2) has an analogous request that should be used instead of `ioctl` (2).

`popen and pclose` (2) — In the SunOS, these functions call `wait` (2). Hence, you should avoid using these for the reasons mentioned above.

`system` (3) — In the SunOS, this function calls `signal` (3) and `wait` (2). Hence, you should avoid using this for the reasons mentioned above.

20.4 What is a Notifier Client?

Notifier

A *client* of the Notifier is anything that has registered a callback routine with it. In XView, a client is an object such as a canvas or panel that you have created using `xv_create()`. Typically, most of the event registration happens at the time such clients are created. However, to the Notifier, a client is nothing more than an ID that distinguishes it from all other Notifier clients. Thus, you could identify a client using a number such as "43" or the address of an object as in `&foo`. XView objects are commonly used as Notifier clients because `xv_create()` returns a unique handle to an object that has been allocated dynamically.

The client ID is of type `Notify_client` as declared in <*xview/notify.h*>.

Certain `notify_*` functions create Notifier clients. For example:

```
Notify_client  client = (Notify_client)10101; /* arbitrary */
Notify_func     destroy_func();

notify_set_destroy_func(client, destroy_func);
```

In this code fragment, up until the call to `notify_set_destroy_func()`, the `client` may not have been registered as a client to the Notifier. When you call `notify_set_destroy_func()`, internally, the Notifier package looks up in its table of clients whether there is any other client using that identifier (the client handle is the identifier, in this case this is an arbitrary value). If there is a client using the identifier, then the destroy function specified is set for that client's destroy handling. Otherwise, a new Notifier client is allocated and the client handle is also used as a handle to the new Notifier client.

Notifier clients are *not* XView objects for you to create or manipulate using `xv_create()` or `xv_set()`. In fact, the Notifier is completely independent of XView and can be used in applications that do not even use XView objects.

20.4.1 Types of Interaction

Client interaction with the Notifier falls into two general categories:

- Event Handling – A client may receive events and respond to them using event handlers. As you have seen, event handlers (callbacks and notify procedures) do much of the work in the Notifier environment.
- Interposition – A client may request that the Notifier install a special type of event handler, supplied by the client, to be *interposed* ahead of the current event handler for a given type of event and client. This allows clients to screen incoming events and redirect them and to monitor and change the status of other clients. A thorough discussion of interposition is presented later in this chapter.

A client establishes an interest in a certain type of event by registering an event handler to respond to it. The following sections cover registration procedures for special signal and UNIX-related event handlers, for application-defined client event handlers, and for interposing event handlers. All event handlers, including interposers, return a value of `NOTIFY_DONE`, `NOTIFY_IGNORED`, or `NOTIFY_UNEXPECTED`, depending on how the event was handled. These three return values are sometimes used to convey important information about the status of the event in the Notifier (usually the function that called `notify_next_*_func()` looks at the return value). These return values and their significance are covered in Section 20.9.4, "Invoking the Next Function."

20.5 Signal Handling

Signals are UNIX software interrupts. The Notifier multiplexes access to the UNIX signal mechanism. A Notifier client may ask to be notified that a UNIX signal occurred either when it is received (asynchronously) and/or later during normal processing (synchronously).

Clients may define and register a signal event handler to respond to any UNIX signal desired. However, some of the signals that you might catch in a traditional UNIX program should be caught instead by the Notifier.

CAUTION

> Clients of the Notifier should not directly catch any UNIX signals using `signal` (3), `sigvec` (2), or `sigaction` (2). There are critical stages of event reading and dispatching that, if interrupted, could cause the program to jump to another location and interrupt the communication protocol between the X server and the application.

Exceptions to this are noted later in this section.

20.5.1 Signals to Avoid

Clients should not have to catch any of the following signals (even via `notify_set_signal_func()` described below). If they are, you are probably utilizing the Notifier inappropriately. The Notifier catches these signals itself under a variety of circumstances and handles them appropriately.

`SIGALRM` Caught by the Notifier's interval timer manager. Use `notify_set_itimer_func()` instead.

`SIGVTALRM` The same applies for `SIGVTALRM` as for `SIGALRM` above.

`SIGTERM` Caught by the Notifier so that it can tell its clients that the process is going away. Use `notify_set_destroy_func()` if that is why you are catching `SIGTERM`.

`SIGCHLD` Caught by the Notifier so that it can do child process management. Use `notify_set_wait3_func()` instead.

SIGIO Caught by the Notifier so that it can manage its file descriptors that are running in asynchronous I/O mode. Use `notify_set_input_func()` or `notify_set_output_func()` if you want to know when there is activity on your file descriptor.

SIGURG Caught by the Notifier so that it can dispatch exception activity on a file descriptor to its clients. Use `notify_set_exception_func()` if you are looking for out-of-band communications when using a socket.

The last two signals in the list are considered advanced topics and are not covered in this manual.

Notifier

20.5.2 A Replacement for signal()

Instead of using `signal()` to catch signals delivered by UNIX, you should register a signal event handler by calling `notify_set_signal_func()`. This function allows you to call another function when a specified signal is generated. Its form is as follows:

```
Notify_func
notify_set_signal_func(client, signal_func, sig, when)
    Notify_client      client;
    Notify_func        signal_func;
    int                sig;
    Notify_signal_mode when;
```

`signal_func` is the function to call when the signal described by `sig` occurs.*

The `when` parameter is either `NOTIFY_SYNC` or `NOTIFY_ASYNC`. `NOTIFY_SYNC` causes notification during normal processing. In other words, the delivery of the signal is delayed to avoid interrupting Xlib Protocol communication between the X server and the application. When it is safe to do so, your `signal_func` function is called and you can display a notice, exit, or jump to another place in the program. Typically, there is a very short time between signal delivery and notification to your callback routine. It is only a little slower than when you use `signal()`.

`NOTIFY_ASYNC` causes notification as soon as the signal is received. This mode mimics the UNIX `signal` (3) semantics.

CAUTION

When using asynchronous signals, your routine may set a variable, a condition, or a flag indicating that the signal was received, or it may change any internal state to your program. Do *not* make any XView, Xlib, or Notifier calls or call any function that might manipulate any XView data structures. Also, do not call `longjmp()` or `setjmp()`; they can cause a condition that can interfere with the X11 Protocol.

*See <*signal.h*> for a list of signals and definitions.

`notify_set_signal_func()` returns a pointer to the function that was installed before you set the new function. You should use this to reset the function if you want to unregister your installed function.

When the specified signal occurs, your `signal_func` is called:

```
Notify_value
signal_func(client, sig, when)
    Notify_client      client;
    int                sig;
    Notify_signal_mode when;
```

The parameters to the signal handler are not the same as the ones given to a signal handler in the call to `signal` (3). However, it is not advisable to use either one, except in unusual circumstances. As a general rule, you should use `notify_set_signal_func()` for all signal handling. If you want the signal *code* or *context* from the signal that was generated (two parameters that are passed to a normal UNIX signal handler), you can use:

```
int
notify_get_signal_code()

struct sigcontext *
notify_get_signal_context()
```

These two functions take no parameters—they return the signal code and context (respectively) of the last signal generated. If you wish to save these values, you can copy them.

Using the Notifier, you can catch any signal except `SIGKILL` and `SIGSTOP`, which cannot be caught by a UNIX application. Attempting to do so generates an error.

An example of common signal handling is shown below:

```
#include <xview/notify.h>

...
extern Notify_value sigint_func();

notify_set_signal_func(frame, sigint_func, SIGINT, NOTIFY_SYNC);
...

Notify_value
sigint_func(client, sig, when)
Notify_client client;
int sig;
{
    puts("received interrupt -- exiting");
    xv_destroy_safe(frame);
    return NOTIFY_DONE;
}
```

The return value of your signal handler tells whether you handled the signal or ignored it.*

*Currently, the return value is ignored.

20.5.3 Timers

Notifier

One specific type of signal is a *timer*. Timers can be set up to call a routine after the passage of a specified amount of time.* Such a routine may cause a caret to flash at regular intervals or allow a clock application to change the time display. The timer is handled differently from other signals in that multiple timers can be installed for various clients (say a flashing caret and a clock in the same application). There can only be one timer function (`timer_func`) per client.

Because the Notifier handles timers, you should not make calls to such routines as `sleep()` or `setitimer()` (see Section 20.11, "Emulating a Sleep Call," for information on how to "sleep" in XView). As pointed out above, you should not attempt to use `signal()` to catch `SIGALRM` or `SIGVTALRM` to trap timer signals. To set timers and be notified when they expire, use `notify_set_itimer_func()`. The form of the call is:

```
Notify_func
notify_set_itimer_func(client, timer_func, which, value, ovalue)
    Notify_client   client;
    Notify_func     timer_func;
    int             which;
    struct itimerval *value, *ovalue;
```

The parameter `which` indicates which type of timer you want to use. Its value is either `ITIMER_REAL` or `ITIMER_VIRTUAL`.

The `value` parameter is a pointer to an `itimerval` which indicates the initial timeout and an interval timeout. The interval timeout is what is used after the initial timeout times out. If the initial timeout is 0, then the timer is not called, regardless of the value of the interval timer. The granularity of the timer is dependent on your hardware architecture and operating system. It is perhaps unwise to assume that you will be notified as frequently as 30 milliseconds or less. Some cases may have higher minimum limits—your mileage may vary.

The `ovalue` is also a pointer to an `itimerval` structure. If there was a previous timeout that has yet to expire, `itimerval` will have that time filled in. You may pass a `NULL` as `ovalue` if you are not interested in this value.

The `timer_func` parameter is the function to call when the timer expires. `notify_set_itimer_func()` returns the function that was previously set for this client (see `notify_set_signal_func` above). The form of the `timer_func` is:

```
Notify_value
timer_func(client, which)
    Notify_client client;
    int           which;
```

Example 20-1 demonstrates one possible use of `notify_set_itimer_func()`. This program displays an animation of several icons stored in the *icon* font. When animating, the next icon in a sequence is displayed using `XDrawString()` (since the icon is actually part of the font). The slider controls the rate at which the next icon is drawn. If the slider is set to 0, the animation stops. The slider is a panel item whose callback routine makes calls to

*Event processing still takes place during this time segment.

notify_set_itimer_func() to set the timer to the new value or to turn it off by setting the function to NOTIFY_FUNC_NULL.

Example 20-1. The animate.c program

```
/*
 * animate.c -- use glyphs from the "icon" font distributed with XView
 * to do frame-by-frame animation.
 */
#include <stdio.h>
#include <ctype.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xos.h>            /* for <sys/time.h> */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/font.h>
#include <xview/notify.h>

Frame           frame;
Display         *dpy;
GC              gc;
Window          canvas_win;
Notify_value    animate();
struct itimerval timer;

#define ArraySize(x)  (sizeof(x)/sizeof(x[0]))
char *horses[ ] = { "N", "O", "P", "Q", "R" };
char *boys[ ] = { "\007", "\005", "\007", "\010" };
char *men[ ] = { "\\", "]", "Y", "Z", "[" };
char *eyes[ ] = {
    "2", "5", "4", "3", "4", "5",
    "2", "1", "0", "/", "0", "1"
};

int max_images = ArraySize(horses);
char **images = horses;
int cnt;

main(argc, argv)
int     argc;
char    *argv[ ];
{
    Panel       panel;
    Canvas      canvas;
    XGCValues   gcvalues;
    Xv_Font     _font;
    XFontStruct *font;
    void        adjust_speed(), change_glyph();
    extern void exit();

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        NULL);
```

Example 20-1. The animate.c program (continued)

```
    panel = (Panel)xv_create(frame, PANEL,
        PANEL_LAYOUT,           PANEL_VERTICAL,
        NULL);
    xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    xv_create(panel, PANEL_SLIDER,
        PANEL_LABEL_STRING,     "Millisecs Between Frames",
        PANEL_VALUE,            0,
        PANEL_MAX_VALUE,        120,
        PANEL_NOTIFY_PROC,      adjust_speed,
        NULL);
    xv_create(panel, PANEL_CHOICE,
        PANEL_LABEL_STRING,     "Glyphs",
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        PANEL_DISPLAY_LEVEL,    PANEL_ALL,
        PANEL_CHOICE_STRINGS,   "Horse", "Man", "Boy", "Eye", NULL,
        PANEL_NOTIFY_PROC,      change_glyph,
        NULL);
    window_fit(panel);

    canvas = (Canvas)xv_create(frame, CANVAS,
        XV_WIDTH,               64,
        XV_HEIGHT,              64,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        NULL);
    canvas_win = (Window)xv_get(canvas_paint_window(canvas), XV_XID);

    window_fit(frame);

    dpy = (Display *)xv_get(frame, XV_DISPLAY);
    _font = (Xv_Font)xv_find(frame, FONT,
        FONT_NAME,      "icon",
        NULL);
    font = (XFontStruct *)xv_get(_font, FONT_INFO);

    gcvalues.font = font->fid;
    gcvalues.foreground = BlackPixel(dpy, DefaultScreen(dpy));
    gcvalues.background = WhitePixel(dpy, DefaultScreen(dpy));
    gcvalues.graphics_exposures = False;
    gc = XCreateGC(dpy, RootWindow(dpy, DefaultScreen(dpy)),
        GCForeground | GCBackground | GCFont | GCGraphicsExposures,
        &gcvalues);

    xv_main_loop(frame);
}

void
change_glyph(item, value)
Panel_item item;
int value;
{
    cnt = 0;
    if (value == 0) {
        max_images = ArraySize(horses);
```

Example 20-1. The animate.c program (continued)

```
        images = horses;
    } else if (value == 1) {
        max_images = ArraySize(men);
        images = men;
    } else if (value == 2) {
        max_images = ArraySize(boys);
        images = boys;
    } else if (value == 3) {
        max_images = ArraySize(eyes);
        images = eyes;
    }
    XClearWindow(dpy, canvas_win);
}

/*ARGSUSED*/
Notify_value
animate(client, which)
Notify_client  client;
int            which;
{
    XDrawImageString(dpy, canvas_win, gc, 5, 40, images[cnt], 1);
    cnt = (cnt + 1) % max_images;

    return NOTIFY_DONE;
}

void
adjust_speed(item, value)
Panel_item item;
int value;
{
    if (value > 0) {
        timer.it_value.tv_usec = (value + 20) * 1000;
        timer.it_interval.tv_usec = (value + 20) * 1000;
        notify_set_itimer_func(frame, animate,
            ITIMER_REAL, &timer, NULL);
    } else
        /* turn it off */
        notify_set_itimer_func(frame, NOTIFY_FUNC_NULL,
            ITIMER_REAL, NULL, NULL);
}
```

Figure 20-1 shows one frame of an animated horse sequence produced by *animate.c*.

20.5.4 Handling SIGTERM

The `SIGTERM` signal is a software *terminate* signal. If you receive a `SIGTERM` signal, another process is telling your application to terminate. Rather than handling this signal with `notify_set_signal_func()`, you could use `notify_set_destroy_func()`. It takes the form:

```
Notify_func
notify_set_destroy_func(client, destroy_func)
```

```
Notify_client  client;
Notify_func    destroy_func;
```

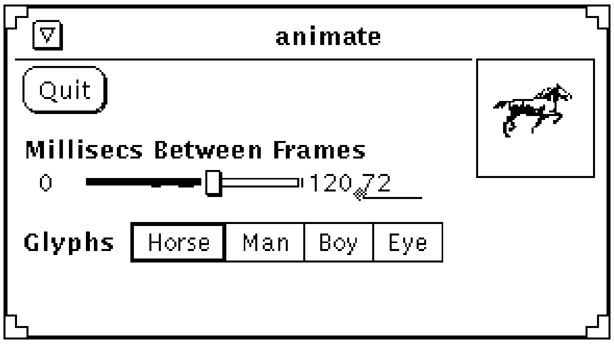

Figure 20-1. Output of animate.c

Like the other Notifier functions, this one also returns the previously set function that was handling this signal. Note that it only interprets SIGTERM—it does not get called if the application chooses to kill itself by calling either `exit()` or `xv_destroy()` on the base frame or if the user selects the *quit* option from the title bar (provided by the OPEN LOOK window manager).

20.5.5 Handling SIGCHLD

Let's say that you want to fork a process to run another program. UNIX requires that you perform some housekeeping on that process. The minimum housekeeping required is to notice when that process dies and to reap it. Normally, the system call `wait()` is used to do this. By default, `wait()` will block until a spawned process has terminated. When `wait()` returns, it has information about the process that died. Rather than have the application sit and wait for a spawned (child) process to die, it should continue processing events and be notified automatically when the process dies.

To handle this, you can register a `wait3` event handler* that the Notifier will call whenever a child process changes state (dies) by calling the following:

```
Notify_func
notify_set_wait3_func(client, wait3_func, pid)
    static Notify_client client;
    Notify_func          wait3_func;
    int                  pid;
```

In the above call, the `pid` identifies the particular child process that the client wants to wait for. The `wait3_func` is the function to call when that child has died. Another reason that an application should use `notify_set_wait3_func()` is the semantics of the `wait3`

*The name *wait3 event* originates from the `wait3` (2) system call. There are other forms of *wait* including `wait()`, `wait2()`, and `wait4()`. Since `wait4()` is not generally available, `wait3()` is the most efficient form of the *wait* functionality.

Notifier

(2) system call. `wait3` (2) will return with status about *any* process that has changed state. If two clients are managing different child processes, they should hear only about their own process. The Notifier keeps straight which client is managing which process.

20.5.5.1 Reaping dead processes

Many clients using child process control simply need to perform the required reaping after a child process dies. These clients can use the predefined `notify_default_wait3()` as their `wait3` event handler. This default handler does nothing but return—the fact that it handled the event is good enough for UNIX. The Notifier automatically removes a dead process's `wait3` event handler.

The code segment in Example 20-2 demonstrates how a *wait3* handler can be set up.

Example 20-2. Demonstrating a wait3 handler

```
#include <xview/notify.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

/* canvas created in another part of the program */
extern Canvas canvas;

/* declare our own wait3 handler */
Notify_value my_wait3_handler();

fork_it(argv)
char *argv[ ];
{
    int pid;

    /* here's the fork -- two processes are going to execute code
     * from this point down.
     */
    switch (pid = fork()) {
        case -1:
            perror("fork");
            return;
        case 0: /* this is the child of the fork -- the new process */
            /* execute the specified command */
            execvp(*argv, argv); /* execvp doesn't return unless failed */
            perror("execvp");
            _exit(0); /* don't call exit() -- man exec() for info */
        default: /* this is the parent -- the original process */
            /* Register a wait3 event handler */
            (void) notify_set_wait3_func(canvas, my_wait3_handler, pid);
    }
    /* parent returns -- child is happily processing away */
}

static Notify_value
my_wait3_handler(me, pid, status, rusage)
    Notify_client  me;
    int            pid;
```

Example 20-2. Demonstrating a wait3 handler (continued)

```
    union wait    *status;
    struct rusage *rusage;
{
    if (WIFEXITED(*status)) {
        /* Child process exited with return code */
        printf("child exited with status %d\n", status->w_retcode);
        /* Tell the Notifier that you handled this event */
        return (NOTIFY_DONE);
    }
    /* Tell the Notifier that you ignored this event */
    return (NOTIFY_IGNORED);
}
```

Example 20-2 is a simple example for demonstration only. There are other things to consider for a complete and proper method for forking new processes. See Section 20.8, "Reading and Writing through File Descriptors," for a full example program that uses `notify_set_wait3_func()`.

20.6 Interaction with RPC

XView provides a function that allows XView and RPC to easily work together. RPC provides for a client and an RPC server. The client makes a remote procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

The RPC server is similar to the XView notifier; it waits for requests and dispatches them to a procedure. The RPC server typically does some initialization (using the function `svc_register`...) and then calls `svc_run()`, which is similar to `xv_main_loop()`. XView works with RPC by incorporating `svc_run()`'s functionality into the notifier.

If you use an RPC server that also needs to work with XView, you should place the XView function `notify_enable_rpc_svc()` in your `main()` program, and do not call `svc_run()`. This function takes an **int** that tells the notifier whether it should handle RPC requests.

```
void
notify_enable_rpc_svc(bool);
    int bool;
```

Using this approach, `xv_main_loop()` handles incoming RPC requests; dispatching them just as if `svc_run()` had been called.*

If `notify_enable_rpc_svc()` is enabled, performance will be affected by the additional call to `svc_getreqset()`.

*Internally XView uses `svc_getreqset(&ibits)` which acts the same way as `notify_dispatch()`. It checks `svc_fdset` and calls `svc_getreqset()` if a `svc_fdset` descriptor is readable, meaning a request is coming in.

20.7 Client Events

Client events are used by an application to communicate with clients of the Notifier. With the client event mechanism, you can have any portion of your application send an event. The Notifier dispatches the event to any of its clients that have expressed interest in that particular client event; essentially the event is sent, via the notifier, to another portion of the application, where it is handled. Events are *posted* to specific clients of the Notifier.

Client events are important and are frequently used internally by XView; they can also be used in an application. For example, if you want an application to notify an interested Notifier client that input has been received from a pipe, once the data has been read, the application can *post* the data to the Notifier as a client event (see Section 20.8, "Reading and Writing through File Descriptors").

From the Notifier's point of view, client events are defined and generated by the application. Client events are not interpreted by the Notifier in any way; the Notifier does not *detect* client events. An X event is also a client event. Internally, the window package reads the X event from the server and posts it to a client via `notify_post_event()`. When client events are *posted* to the Notifier, it dispatches these events to a receiving client's event handler. The receiving client then interprets the client events.

The process for the delivery of client events is similar to that of signals. However, because the entire process happens in the application, more control can be maintained by the Notifier. This additional level of control uses optional parameters that may be passed to client event handlers. These optional parameters include values for *safe* and *immediate* handling. With safe handling the Notifier sends event handlers notification when it is safe to do so. This may involve some delay between when an event is posted and when it is delivered. Alternatively, for immediate handling, an event handler may ask to be immediately notified when a client event is posted. A client event handler may have *both* a safe *and* an immediate event handler. Safe and immediate notification are covered in more detail later on in this section.

The remainder of this section describes how to register client event handlers and how to post client events.

NOTE

> XView internally registers event handlers for its internally defined client events when window-based objects are created. You will almost never need to register your own client event handler. It is more common that you will want to interpose in front of one of XView's client event handlers. Refer to Section 20.9, "Interposition," for details on interposition.

20.7.1 Receiving Client Events

Notifier

To register a client event handler, use:

```
Notify_func
notify_set_event_func(client, event_func, when)
    Notify_client      client;
    Notify_func        event_func;
    Notify_event_type  when;

typedef enum notify_event_type {
    NOTIFY_SAFE         = 0,
    NOTIFY_IMMEDIATE    = 1,
} Notify_event_type;
```

The `when` parameter indicates whether the event handler will accept notifications only when it is safe (`NOTIFY_SAFE`) or for immediate notification and when it is safe (`NOTIFY_IMMEDIATE`). You may register two client event handlers to handle these cases individually, or just one client event handler to cover both.

The calling sequence of a client event handler is:

```
Notify_value
event_func(client, event, arg, when)
    Notify_client      client;
    Notify_event       event;
    Notify_arg         arg;
    Notify_event_type  when;
```

The `client` is the one that was passed to `notify_set_event_func()`. The `event` is passed through from `notify_post_event()`, which is described in the following section. The `arg` and `event` parameters are completely defined by the client. The types `Notify_arg` and `Notify_event` are of type `caddr_t`, a generic pointer type. The `when` parameter is the actual situation in which `event` is being delivered. It can have the value `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`, and can be different from `when_hint` of `notify_post_event()`.

The return value `Notify_value` may be one of `NOTIFY_DONE`, `NOTIFY_IGNORED`, or `NOTIFY_UNEXPECTED`. `NOTIFY_DONE` indicates that the event was acted on in some way. This implies that no further action is required by the client. If the safe event handler returns `NOTIFY_IGNORED`, this indicates that the event failed to provoke any action. If the immediate client event handler returns `NOTIFY_IGNORED`, then the same notification will be delivered to the safe client event handler when it is safe. A value of `NOTIFY_UNEXPECTED` indicates the event was not handled and not recognized. This return value may indicate an error condition.

20.7.2 Posting Client Events

A client event may be posted to the Notifier at any time. The poster of a client event may suggest to the Notifier when to deliver the event, but this is only a hint. The Notifier will see to it that it is delivered at an appropriate time (more on this below). The call to post a client event is:

```
typedef char * Notify_event;

Notify_error
notify_post_event(client, event, when_hint)
    Notify_client      client;
    Notify_event       event;
    Notify_event_type  when_hint;
```

The `client` handle from `notify_set_event_func()` is passed to `notify_post_event()`. `event` is defined and interpreted solely by the `client`. A return code of `NOTIFY_OK` indicates that the notification has been posted. Other values indicate an error condition. `NOTIFY_UNKNOWN_CLIENT` indicates that `client` is unknown to the Notifier. `NOTIFY_NO_CONDITION` indicates that `client` has no client event handler registered with the Notifier.

Usually it is during the call to `notify_post_event()` that the client event handler is called. Sometimes, however, the notification is queued up for later delivery. The Notifier chooses between these two possibilities by noting which kinds of client event handlers `client` has registered, whether it is safe, and what the value of `when_hint` is. Here are the cases broken down by the kind of client event handlers `client` has registered:

Immediate only — If `when_hint` is `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`, the event is delivered immediately.

Safe only — If `when_hint` is `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`, the event is delivered when it is safe.

Both safe and immediate — A client may have both an immediate client event handler as well as a safe client event handler. If `when_hint` is `NOTIFY_SAFE`, then the notification is delivered to the safe client event handler when it is safe. If `when_hint` is `NOTIFY_IMMEDIATE`, then the notification is delivered to the immediate client event handler right away. If the immediate client event handler returns `NOTIFY_IGNORED`, then the same notification will be delivered to the safe client event handler when it is safe.

20.7.2.1 Actual delivery time

For client events, other than knowing which event handler to call, the main function of the Notifier is to know when to make the call. The Notifier defines when it is safe to make a client notification. If it is not safe, then the event is queued up for later delivery. Here are the conventions:

- A client that has registered an immediate client event handler is sent a notification as soon as it is received. The client has complete responsibility for handling the event safely. It is rarely safe to do much of anything when an event is received asynchronously. Usually, just setting a flag that indicates that the event has been received is about the safest thing that can be done.
- A client that has registered a safe client event handler will have a notification queued up for later delivery when the notification was posted during an asynchronous signal notification. Immediate delivery is not safe because your process, just before receiving the signal, may have been executing code at any arbitrary place.
- A client that has registered a safe client event handler will have a notification queued up for later delivery if the client's safe client event handler hasn't returned from processing a previous event. This convention is mainly to prevent the cycle: Notifier notifies *A*, who notifies *B*, who notifies *A*. *A* could have had its data structures torn up when it notified *B* and was not in a state to be re-entered.

These conventions imply that a safe client event handler is called immediately from other signal handlers. The following sequence shows how the notification might progress.

- A client's input pending event handler is called by the Notifier.
- Two characters are read by the client's input pending event handler.
- The first character is given to the Notifier to deliver to the client's save event handler (the delivery is accomplished with `notify_post_event()`).
- Returning back to the input pending event handler, the second character is sent. This character is also delivered immediately.

20.7.3 Posting with an Argument

XView posts a fixed-field structure with each event. Sometimes additional data must be passed with an event. For instance, when the scrollbar posts an event to its owner to do a scroll, the scrollbar's handle is passed as an argument along with the event. The function `notify_post_event_and_arg()` provides this argument-passing mechanism (see below).

When posting a client event, there is the possibility of delivery being delayed. In the case of XView, the event being posted is a pointer to a structure. The Notifier avoids an invalid (dangling) pointer reference by copying the event if delivery is delayed. It calls routines the client supplies to copy the event information and later to free up the storage the copy uses.

notify_post_event_and_arg() provides this storage management mechanism.

```
Notify_error
notify_post_event_and_arg(client, event, when_hint, arg,
                          copy_func, release_func)
    Notify_client      client;
    Notify_event       event;
    Notify_event_type  when_hint;
    Notify_arg         arg;
    Notify_copy        copy_func;
    Notify_release     release_func;

typedef caddr_t Notify_arg;

typedef Notify_arg (*Notify_copy)();
#define NOTIFY_COPY_NULL   ((Notify_copy)0)

typedef void (*Notify_release)();
#define NOTIFY_RELEASE_NULL  ((Notify_release)0)
```

copy_func() is called to copy arg (and, optionally, event) when event and arg need to be queued for later delivery. release_func() is called to release the storage allocated during the copy call when event and arg were no longer needed by the Notifier.

Any of arg, copy_func(), or release_func() may be NULL. If copy_func is not NOTIFY_COPY_NULL and arg is NULL, then copy_func() is called anyway. This allows event the opportunity to be copied because copy_func() takes a pointer to event. The event pointed to may be replaced as a side effect of the copy call. The same applies to a NOTIFY_RELEASE_NULL release function with a NULL arg argument.

The copy() and release() routines are client-dependent, so you must write them yourself. Their calling sequences are the following:

```
Notify_arg
copy_func(client, arg, event_ptr)
    Notify_client  client;
    Notify_arg     arg;
    Notify_event  *event_ptr;

void
release_func(client, arg, event)
    Notify_client  client;
    Notify_arg     arg;
    Notify_event   event;
```

20.7.4 Posting Destroy Events

When a destroy notification is set, the Notifier also sets up a synchronous signal condition for SIGTERM that will generate a DESTROY_PROCESS_DEATH destroy notification. Otherwise, a destroy function will not be called automatically by the Notifier. One or two (depending on whether the client can veto your notification) explicit calls to notify_post_destroy() need to be made.

```
Notify_error
notify_post_destroy(client, status, when)
    Notify_client      client;
    Destroy_status     status;
    Notify_event_type  when;
```

NOTIFY_INVAL is returned if status or when is not defined. After notifying a client to destroy itself, all references to client are purged from the Notifier.

20.7.5 Delivery Time of Destroy Events

Unlike a client event notification, the Notifier does not try to detect when it is safe to post a destroy notification. Thus, a destroy notification can come at any time. It is up to the good judgement of a caller of notify_post_destroy() or notify_die() (described in Section 20.10, “Notifier Control”) to make the call when a client is not likely to be in the middle of accessing its data structures.

If status is DESTROY_CHECKING and the argument when is NOTIFY_IMMEDIATE, then notify_post_destroy() may return NOTIFY_DESTROY_VETOED, if the client does not want to go away. See Section 20.9.5, “Modifying an Objects’ Destruction,” for details on these values.

Often you want to tell a client to go away at a safe time. This implies that delivery of the destroy event will be delayed, in which case the return value of notify_post_destroy() cannot be NOTIFY_DESTROY_VETOED because the client has not been asked yet. To get around this problem, the Notifier will flush the destroy event of a checking/destroy pair of events if the checking phase is vetoed. Thus, a common idiom is:

```
(void) notify_post_destroy(client, DESTROY_CHECKING, NOTIFY_SAFE);
(void) notify_post_destroy(client, DESTROY_CLEANUP, NOTIFY_SAFE);
```

20.8 Reading and Writing Through File Descriptors

The Notifier is set up for testing whether or not there is input pending on file descriptors and whether file descriptors are available to accept data for writing. The file descriptors can represent files, pipes, and sockets. System calls such as read() or write() can be used on any of these file descriptors.

In an event-driven system, the application cannot wait for data to be read. Instead it is better for the system to notify the application when data can be read. System calls such as read() will *block* if there is no input to be read. That is, read() waits until there is something to read before it returns. If your application is blocking on a read, then it cannot process events that the user may be generating, such as selecting a panel button. Rather than blocking on a call to read(), the Notifier can inform you when there is input ready on that file descriptor so that when you finally call read(), it returns immediately.

To handle this, the Notifier provides functions such as `notify_set_input_func()` and `notify_set_output_func()` to test whether a file descriptor has data to be read or is ready for writing. These functions inform you of the status of file descriptors, whether they are files, pipes, or sockets.*

```
Notify_func
notify_set_input_func(client, input_func, fd)
    Notify_client   client;
    Notify_func     input_func;
    int             fd;
Notify_func
notify_set_output_func(client, output_func, fd)
    Notify_client   client;
    Notify_func     output_func;
    int             fd;
```

`input_func()` is called whenever the file descriptor (`fd`) has data to be read.† `output_func()` is called whenever the file descriptor (`fd`) is ready to receive data.

Generally, file descriptors open for writing are always ready to receive data, so this function may not be as widely used as the input function case. However, it is important if you are writing to a pipe where another process is probably on the other side of the pipe. Pipes have buffers which, when full, will not accept any more data on them until the process on the other side of the pipe reads the data written so far (thus emptying the buffer). If the process on the other side of the pipe is slow in reading the data you have written to it, then you may need to use `notify_set_output_func()` so that you can be notified when the pipe is ready to have more data written to it.

`input_func()` and `output_func()` take the following form:

```
Notify_value
func(client, fd)
    Notify_client  client;
    int            fd;
```

20.8.1 Reading Files

Our first example demonstrates how `notify_set_input_func()` can be used to read data from a file. The program is simple; it does not bother using any XView objects such as frames or canvases. Because of this, it makes use of `notify_start()`, which is covered in Section 20.10, “Notifier Control.”

*`notify_set_exception_func()` is also available for determining if out-of-band data is available on a socket. Its format is described in the *XView Reference Manual.*
†Exceptions to this are discussed in Section 20.8.1, “Reading Files.”

Example 20-3. The notify_input.c program

```
/*
 * notify_input.c -- use notify_set_input_func to monitor the state of
 * a file.  The Notifier is running and checking the file descriptors
 * of the opened files associated with the command line args.  The
 * routine installed by notify_set_input_func() is called whenever
 * there is data to be read.  When there is no more data to be read
 * for that file, the input function is unregistered.  When all files
 * have been read, notify_start() returns and the program exits.
 */
#include <stdio.h>
#include <sys/ioctl.h>
#include <xview/notify.h>

main(argc, argv)
char *argv[];
{
    Notify_value   read_it();
    Notify_client  client = (Notify_client)10101; /* arbitrary */
    FILE           *fp;

    while (*++argv)
        if (!(fp = fopen(*argv, "r")))
            perror(*argv);
        else {
            (void) notify_set_input_func(client, read_it, fileno(fp));
            client++; /* next client is new/unique */
        }

    /* loops continuously */
    notify_start();
}

/*
 * read_it() is called whenever there is input to be read.  Actually,
 * it is called continuously, so check to see if there is input to be
 * read first.
 */
Notify_value
read_it(client, fd)
Notify_client   client;
int fd;
{
    char buf[BUFSIZ];
    int bytes, i;

    if (ioctl(fd, FIONREAD, &bytes) == -1 || bytes == 0)
        (void) notify_set_input_func(client, NOTIFY_FUNC_NULL, fd);
    else
        do
            if ((i = read(fd, buf, sizeof buf)) > 0)
                (void) write(1, buf, i);
        while (i > 0 && (bytes -= i) > 0);
    return NOTIFY_DONE;
}
```

Notifier

The comments in this sample program describe what is going on up front. However, behind the scenes, there are interesting things happening.

The first thing to notice is that the clients used in the Notifier start at `10101` and, for each file on the command line, the client is incremented by one. Remember, the client is an arbitrary identifier and can be any unique value. Since there are no other clients of the Notifier, we know that `10101` is going to be unique.

The function `read_it()` is installed to read data from the files given on the command line. The first thing that `read_it()` does is check if there is data to be read. The `ioctl()` call returns the number of bytes to read in the `bytes` variable. If there is data to be read, then `read()` is used to read buffers (of size BUFSIZ) until it has read all the bytes pending. This continues until the program has read the entire file. Once this happens, the `ioctl()` call will return that there are no bytes to read. Normally, it would return -1 indicating that the end of file has been reached. However, the Notifier has modified the state of the file descriptor for internal purposes (to set nonblocking mode). When we have reached the end of the file, we unregister the client by calling `notify_set_input_func()` with a `notify_func` of NOTIFY_FUNC_NULL.

notify_input.c is not terribly interesting because you do not need to be notified of data to be read on a *file*; you can just open the file, read it till `EOF`, and then close it. But if you know that the file is going to have data continuously added to it, you might want to be informed when there is new data to be read and then print it out.

This can be accomplished by not unregistering the input function with the Notifier when there is no data to be read. Unfortunately, your function is going to be called continuously whether there is new data to be read or not. This is true *only for file descriptors that represent files* and is not the case for pipes.

20.8.2 Reading and Writing on Pipes

A more interesting and likely problem occurs when an application has to execute another program, send data to it, and read output from it all at the same time. The best way to handle such a situation is as follows:

- Set up a pipe for each stream (such as `stdin` and `stdout`). This is done using the `pipe()` system call.
- `fork()` a new process. The child will execute the program (using `execvp()`) and have its input and output redirected to the appropriate ends of the pipes (using `dup2()`).
- The parent registers *input* and *output* functions to read and write from the "other" ends of the pipes (using `notify_set_input_func()`). Also, the parent calls `notify_set_wait3_func()`, as discussed earlier.

The example program that demonstrates this capability is a little longer, but it is still simple. It follows the steps outlined above and is the minimum needed to spawn a new process. Again, to keep the code short and simple, no XView objects are created. The existence of XView objects would not affect the program in any way.

The way to run this program is to specify a command on the command line of the program. For example:

```
% ntfy_pipe cat
```

Remember, there are *two* processes involved here: the parent process (`ntfy_pipe`) and the child process (`cat`). Both programs read from their respective `stdin` and write to their own `stdout`. However, because the child process (`cat`) was spawned off from the parent, we need to handle the IO (input/output) of the new process. After the call to `fork()`, its `stdin` and `stdout` must be redirected to the pipe that was set up before the `fork`. The parent, however, retains its IO—its `stdin` and `stdout` remain directed towards the window in which you typed the command.

Example 20-4. The ntfy_pipe.c program

```
/*
 * ntfy_pipe.c -- fork and set up a pipe to read the IO from the
 * forked process.  The program to run is specified on the command
 * line.  The functions notify_set_input_func() and
 * notify_set_output_func() are used to install functions which read
 * and write to the process' stdin and stdout.
 * The program does not use any XView code -- just the Notifier.
 */
#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/resource.h>
#include <sys/ioctl.h>
#include <xview/notify.h>

Notify_client client1 = (Notify_client)10;
Notify_client client2 = (Notify_client)11;

int pipe_io[2][2]; /* see diagram */
/*
 *                 [0]                           [1]
 *    child reads:  |========= pipe_io[0] ========| <- parent writes
 *   pipe_io[0][0]                                     pipe_io[0][1]
 *
 *    parent reads: |========= pipe_io[1] ========| <- child writes
 *   pipe_io[1][0]                                     pipe_io[1][1]
 *
 * The parent process reads the output of the child process by reading
 * pipe_io[1][0] because the child is writing to pipe_io[1][1].
 * The child process gets its input from pipe_io[0][0] because the
 * parent writes to pipe_io[0][1].  Thus, one process is reading from
 * one end of the pipe while the other is writing at the other end.
 */
main(argc, argv)
char *argv[];
{
    Notify_value        read_it(), write_it(), sigchldcatcher();
    int                 i, pid;
    FILE                *fp;
```

Example 20-4. The ntfy_pipe.c program (continued)

```
    if (!*++argv)
        puts("specify a program [w/args]"), exit(1);

    pipe(pipe_io[0]); /* set up input pipe */
    pipe(pipe_io[1]); /* set up output pipe */
    switch (pid = fork()) {
        case -1:
            close(pipe_io[0][0]);
            close(pipe_io[0][1]);
            close(pipe_io[1][0]);
            close(pipe_io[1][1]);
            perror("fork failed");
            exit(1);
        case  0: /* child */
            /* redirect child's stdin (0), stdout (1) and stderr(2) */
            dup2(pipe_io[0][0], 0);
            dup2(pipe_io[1][1], 1);
            dup2(pipe_io[1][1], 2);
            for (i = getdtablesize(); i > 2; i--)
                (void) close(i);
            for (i = 0; i < NSIG; i++)
                (void) signal(i, SIG_DFL);
            execvp(*argv, argv);
            if (errno == ENOENT)
                printf("%s: command not found.0, *argv);
            else
                perror(*argv);
            perror("execvp");
            _exit(-1);
        default: /* parent */
            close(pipe_io[0][0]); /* close unused portions of pipes */
            close(pipe_io[1][1]);
        }

    /* when the process outputs data, read it */
    notify_set_input_func(client1, read_it, pipe_io[1][0]);
    notify_set_wait3_func(client1, sigchldcatcher, pid);

    /* wait for user input -- then write data to pipe */
    notify_set_input_func(client2, write_it, 0);
    notify_set_wait3_func(client2, sigchldcatcher, pid);

    notify_start();
}

/*
 * callback routine for when there is data on the parent's stdin to
 * read.  Read it and then write the data to the child process via
 * the pipe.
 */
Notify_value
write_it(client, fd)
Notify_client   client;
int fd;
{
```

Example 20-4. The ntfy_pipe.c program (continued)

```
    char buf[BUFSIZ];
    int bytes, i;

    /* only write to pipe (child's stdin) if user typed anything */
    if (ioctl(fd, FIONREAD, &bytes) == -1 || bytes == 0) {
        notify_set_input_func(client, NOTIFY_FUNC_NULL, pipe_io[0][1]);
        close(pipe_io[0][1]);
    } else
        while (bytes > 0) {
            if ((i = read(fd, buf, sizeof buf)) > 0) {
                printf("[Sending %d bytes to pipe (fd=%d)]0,
                    i, pipe_io[0][1]);
                write(pipe_io[0][1], buf, i);
            } else if (i == -1)
                break;
            bytes -= i;
        }
    return NOTIFY_DONE;
}

/*
 * callback routine for when there is data on the child's stdout to
 * read.  Read, then write the data to stdout (owned by the parent).
 */
Notify_value
read_it(client, fd)
Notify_client   client;
register int fd;
{
    char buf[BUFSIZ];
    int bytes, i;

    if (ioctl(fd, FIONREAD, &bytes) == 0)
        while (bytes > 0) {
            if ((i = read(fd, buf, sizeof buf)) > 0) {
                printf("[Reading %d bytes from pipe (fd=%d)]0,
                    i, fd);
                (void) write(1, buf, i);
                bytes -= i;
            }
        }
    return NOTIFY_DONE;
}

/*
 * handle the death of the child.  If the process dies, the child
 * dies and generates a SIGCHLD signal.  Capture it and disable the
 * functions that talk to the pipes.
 */
Notify_value
sigchldcatcher(client, pid, status, rusage)
Notify_client client; /* the client noted in main() */
int pid; /* the pid that died */
union wait *status; /* the status of the process (unused here) */
struct rusage *rusage; /* resources used by this process (unused) */
{
```

Example 20-4. The ntfy_pipe.c program (continued)

```
    if (WIFEXITED(*status)) {
        printf("Process termined with status %d0, status->w_retcode);
        /* unregister input func with appropriate file descriptor */
        notify_set_input_func(client, NOTIFY_FUNC_NULL,
            (client == client1)? pipe_io[1][0] : 0);
        return NOTIFY_DONE;
    }
    puts("SIGCHLD not handled");
    return NOTIFY_IGNORED;
}
```

Assuming that the program was run with `cat` (1) as described earlier, input to the child process comes from the function `write_it()`. That is, `write_it()` is the function that *writes* to the other side of the pipe that the child is reading from. This function was set up as the *input_func* for the *parent's* `stdin`. The function gets its input from whatever you type in the window in which you ran the program. It could have gotten its input from a text panel item or a selection using the selection service. It reads whatever data is sent through the pipe and ends up as the `stdin` for the child process.

Similarly, when the child process writes anything to its `stdout`, the data is redirected through the pipe. The parent reads from the other end of the pipe in `read_it()`. Whatever the parent reads is written to the parent's `stdout`, which is the user's tty window. It does this by calling `write(1, buf, i)`. This line of the program can easily be replaced by:

```
textsw_insert(textsw, buf, i);
```

This shows how you can redirect the output (including `stderr` output) from a child process to a text subwindow.

The last thing to note about the program is its treatment of extraneous file descriptors and of signals handled by the child process after forking. When using `fork()`, the child process inherits all the file descriptors and signal handling set up by the parent.* This could seriously affect your program if the child process gets certain signals. In this case the best thing for the child to do is loop through all the signals that are dealt with by the parent and reset them to `SIG_DFL` using `signal` (3). Then it can close all file descriptors except for the child's `stdin` (0), `stdout` (1), and `stderr` (2) descriptors.

We use the `signal()` system call here because it is assumed that the child is not going to execute any of the XView code in the parent's program. Therefore, the child does not need to use the `notify_set_signal_func()` routine to unregister the signals. Releasing these signals in this way completely disassociates the program from the parent.

This very general program is used as an example, but it is useful in all applications that need to run external processes. The only modifications it needs are alternate sources for the input and output of the parent. `main()` could be replaced by a general function that simply gets an `argv` parameter containing the program to execute, including arguments.

*Unless the file descriptors were set to *close on exec* using `ioctl(fd, FIOCLEX, 0)`.

The program could be modified to be used as a replacement for the `system` (3) call. Even if you do not expect to read the child's `stdout` or send data to its `stdin`, the child process still needs to be handled differently from `system` (3) because that function calls `wait()` and attempts to do its own signal handling.

20.8.3 Exception Occurred Events

Exception occurred notifications are similar to input pending notifications. The only known devices that generate exceptions are stream-based socket connections when an out-of-band byte is available. Thus a `SIGURG` signal catcher is set up by the Notifier, much like a `SIGIO` for asynchronous input.

```
Notify_func
notify_set_exception_func(nclient, func, fd)
    Notify_client   nclient;
    Notify_func     func;
    int             fd;
```

20.8.4 Getting an Event Handler

This section contains a list of the `notify_get_*_func()` routines. These functions allow you to retrieve the value of a client's event handler. Once you have the value of a client's event handler, you could modify or replace the event handler. It is recommended that you use the Notifier's interposition mechanism instead of using these functions. The arguments for these functions parallel the associated `notify_set_*_func()` functions described previously, except for the absence of the event handler function pointer. Refer to the *XView Reference Manual* for a description of the arguments.

For all of the `notify_get_*_func()` functions in the following list, a return value of `NOTIFY_FUNC_NULL` indicates an error. If the client is unknown, then `notify_errno` is set to `NOTIFY_UKNOWN_CLIENT`. If no event handler is registered for the specified event, then `notify_errno` is set to `NOTIFY_NO_CONDITION`. Other values on `notify_errno` are possible, depending on the event; for example, `NOTIFY_BAD_FD` if an invalid file descriptor is specified.

Here is a list of the event handler retrieval routines:

- `notify_get_input_func(client, fd)`
- `notify_get_event_func(client, when)`
- `notify_get_output_func(client, fd)`
- `notify_get_exception_func(client, fd)`
- `notify_get_signal_func(client, which)`
- `notify_get_itimer_func(client, signal, mode)`

- `notify_get_wait3_func(client, pid)`
- `notify_get_destroy_func(client)`

20.9 Interposition

The Notifier provides a mechanism called *interposition,* with which you can intercept control of the internal communications within XView. Interposition is a powerful way to both monitor and modify window behavior in ways that extend the functionality of a window object.

Interposition allows a client to intercept an event before it reaches the *base event handler*. The base event handler is the one set originally by a client. The client can call the base event handler before and/or after its own handling of the event or not at all. This allows the application to override the handling provided by XView's base event handler, or to add special event handling. The notifier supports interposition by keeping track of how interposition functions are ordered for each type of event for each client.

There may be more than one interposer for a Notifier client. As each interposer is added, it is inserted ahead of the last interposer installed. Interposes may also be removed from the interposer list. When an event arrives, the Notifier calls the function at the top of the interposer list for that client.

Keeping track of which function to call when running down the list of interposers is done by maintaining an *interposition stack*. The number of interpositions allowed on the stack is limited to a maximum size (about six levels deep). Since the base event handler is placed on the interposition stack, there are five usable levels, although few applications should require more than two levels of interposition. Figure 20-2 illustrates the flow of control with interposition. Note that the interposer could have stopped the flow of control to the next event handler.

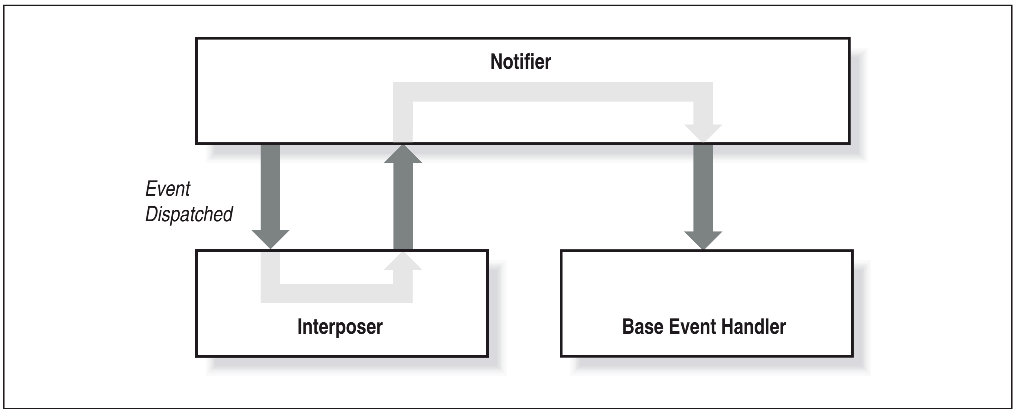

Figure 20-2. Flow of control in interposition

20.9.1 Uses of Interposition

Typically, it is application-level code that uses interposition. But, in general, any client's creator may want to use interposition. There are many reasons why an application might want to interpose a function in the call path to a client's event handler.

- An application may want to use the fact that a client has received a particular event as a trigger for some application-specific processing.
- An application may want to filter the events to a client, thus modifying the client's behavior.
- An application may want to extend the functionality of a client by handling events that the client is not programmed to handle.

XView window objects utilize the Notifier for much of their communication and cooperation. Thus, if an application wanted to monitor the user actions directed to a particular window, the application would use interposition to get into the flow of control.

20.9.2 Interface to Interposition

The Notifier supports interposition by keeping track of how interposition functions are ordered for each type of event for each client. Here is a typical example of interposition:

- An application creates a client. The client has set up its own client event handler using `notify_set_event_func()`. XView does this internally, using a default event handler, when you create a window based object.
- The application tells the Notifier that it wants to interpose a function in front of the client's event handler by calling `notify_interpose_event_func()`, which uses the same calling sequence as `notify_set_event_func()`.
- When the application's interposed function is called, it tells the Notifier to call the next function, i.e., the client's function, via a call to `notify_next_event_func()`, which uses the same calling sequence as that passed to the interposer function.

Note that you can only interpose if a base event handler has been set with `notify_set_*_func` (one of the `notify_set` functions). If no function is set, `notify_interpose_*_func()` will return an error.

20.9.3 Registering an Interposer

This section lists the routines that allow you to interpose your own function in front of an event handler. The arguments to each `notify_interpose_*_func()` function like the associated `notify_set_*_func()` function described in the previous sections on the Notifier. Refer to the associated `notify_set_*_func()` description, or to the *XView Reference Manual* for details on the various arguments.

- notify_interpose_destroy_func()
- notify_interpose_exception_func()
- notify_interpose_event_func()
- notify_interpose_input_func()
- notify_interpose_itimer_func()
- notify_interpose_output_func()
- notify_interpose_signal_func()
- notify_interpose_wait3_func()

The return values from these functions may be NOTIFY_OK, NOTIFY_UNKNOWN_CLIENT, NOTIFY_NO_CONDITION, NOTIFY_UNEXPECTED, or NOTIFY_FUNC_LIMIT. For NOTIFY_OK the interposition was successful. For NOTIFY_UNKNOWN_CLIENT the client is not known to the notifier. For NOTIFY_NO_CONDITION there is no event handler of the type specified. For NOTIFY_FUNC_LIMIT the level of interposition was exceeded. NOTIFY_FUNC_LIMIT means you are trying to use more than the maximum allowed levels of interposition.

If the return value is something other than NOTIFY_OK, then notify_errno contains the error code.

20.9.4 Invoking the Next Function

This section lists the routines that you call from your interposed function in order to invoke the next function in the interposition sequence (if you want to override the normal sequence, do not use these routines). The arguments to each notify_next_*_func() function are the same as the arguments passed to the interposer function. Refer to the *XView Reference Manual*, for details on the arguments.

- notify_next_destroy_func()
- notify_next_exception_func()
- notify_next_event_func()
- notify_next_input_func()
- notify_next_itimer_func()
- notify_next_output_func()
- notify_next_signal_func()
- notify_next_wait3_func()

The return value for these functions may be one of the following: NOTIFY_DONE, NOTIFY_IGNORED, or NOTIFY_UNEXPECTED. NOTIFY_DONE indicates that the event was acted on in some way. This implies that no further action is required by the client. If the safe event handler returns NOTIFY_IGNORED, this indicates that the event failed to provoke any action. If the immediate client event handler returns NOTIFY_IGNORED, then the same notification will be delivered to the safe client event handler when it is safe. A value of NOTIFY_UNEXPECTED indicates the event was not handled and not recognized. This return value may indicate an error condition.

Notifier

20.9.5 Removing an Interposed Function

This section presents a list of routines that allow you to remove the interposer function that you installed using a `notify_interpose_*_func()` call. The arguments to each `notify_remove_*_func()` function are the same as the arguments passed to the associated `notify_set_*_func()` function described in previous sections. Refer to the previous section, or to the *XView Reference Manual*, for details on these arguments. Note the one exception to this rule is that the arguments to `notify_remove_itimer_func()` are a subset of the arguments to `notify_set_itimer_func()`.

- `notify_remove_destroy_func()`
- `notify_remove_exception_func()`
- `notify_remove_event_func()`
- `notify_remove_input_func()`
- `notify_remove_itimer_func()`
- `notify_remove_output_func()`
- `notify_remove_signal_func()`
- `notify_remove_wait3_func()`

If the function returns successfully, the return value will be NOTIFY_OK. Otherwise, the error codes are the same as those associated with the `notify_interpose_*_func()` calls.

20.9.6 An Interposition Example

You can notice when a frame opens or closes by interposing in front of the frame's *client event handler*. The client event handler that you want to interpose in front of is the default client event handler supplied by XView. To register an interposer, the following routine is used:

```
Notify_error
notify_interpose_event_func(client, event_func, type)
    Notify_client     client;
    Notify_func       event_func;
    Notify_event_type type;
```

The `client` must be the handle of the Notifier client in front of which you are interposing. In XView, this is the handle returned from `xv_create()`, for a `Tty` subwindow or a

Textsw, the handle is the OPENWIN_NTH_VIEW of the window. For a Canvas, the handle returned from is CANVAS_NTH_PAINT_WINDOW.

Let's say that the application is displaying some animation and wants to do the necessary computation only when the frame is open. It can use interposition to notice when the frame opens or closes.

In Example 20-5, note the call to notify_next_event_func(). This function transfers control to the frame's client event handler through the Notifier. The routine notify_next_event_func() takes the same arguments as the interposer.

Example 20-5. Transferring control through the Notifier

```
#include <xview/xview.h>

main()
{
    Frame frame;
    Notify_value my_frame_interposer();

    /* Create the frame */
    frame = xv_create(NULL, FRAME, NULL);

    /* Interpose in front of the frame's event handler */
    (void) notify_interpose_event_func(frame,
           my_frame_interposer, NOTIFY_SAFE);
    ...

    /* Show frame and start dispatching events */
    xv_main_loop(frame);
}

Notify_value
my_frame_interposer(frame, event, arg, type)
Frame              frame;
Event             *event;
Notify_arg         arg;
Notify_event_type  type;
{
    int closed_initial, closed_current;
    Notify_value value;

    /* Determine initial state of frame */
    closed_initial = (int) xv_get(frame, FRAME_CLOSED);

    /* Let frame operate on the event */
    value = notify_next_event_func(frame, (Notify_event)event, arg, type);

    /* Determine current state of frame */
    closed_current = (int) xv_get(frame, FRAME_CLOSED);

    /* Change animation if states differ */
    if (closed_initial != closed_current) {
        if (closed_current) {
            /* Turn off animation because closed */
            (void) notify_set_itimer_func(me, my_animation,
                ITIMER_REAL, ITIMER_NULL, ITIMER_NULL);
```

Example 20-5. Transferring control through the Notifier (continued)

```
        } else {
            /* Turn on animation because opened */
            (void) notify_set_itimer_func(me, my_animation,
                ITIMER_REAL, &NOTIFY_POLLING_ITIMER, ITIMER_NULL);
        }
    }
    return (value);
}
```


In Example 20-5, the base event handler is intended to handle the event (so that the frame gets closed/opened). If the interposed function replaces the base event handler and you do not want the base event handler to be called at all, your interposed procedure should not call `notify_next_event_func()`.

20.9.7 Interposing on Resize Events

Another use of interposition is to give your application more control over the layout of its subwindows. To do this, you set up an interpose event handler which checks if the event type is `WIN_RESIZE`. If so, rather than calling `notify_next_event_func()` to dispatch the event to the normal handler for resizing, you call your own resize routine:

```
Notify_value
my_frame_interposer(frame, event, arg, type)
Frame             frame;
Event            *event;
Notify_arg        arg;
Notify_event_type  type;
{
    Notify_value value;

    if (event_action(event) == WIN_RESIZE)
        value = resize(frame);
    else
        value = notify_next_event_func(frame, (Notify_event)event, arg, type);

    return(value);
}
```

20.9.8 Modifying an Object's Destruction

Suppose an application must detect when the user selects the "Quit" menu item in order to perform some application-specific confirmation. To accomplish this, the application should interpose a new function in front of the object's *client-destroy event* handler using the following routine:

```
Notify_error
notify_interpose_destroy_func(client, destroy_func)
    Notify_client client;
    Notify_func   destroy_func;
```

For an XView object, destruction may originate from several sources:

- The application may be terminated by a software interrupt from the calling process.
- The connection to the server has been lost and the Notifier is informing the application that it is dying.
- `xv_destroy(`*`object`*`)` was called from within the application.
- The user may select the "Quit" menu item from the window manager's pulldown menu.

Each of these scenarios results in a different destruction method. When the object is going to be destroyed, the process happens in two phases. First, the object's destroy-interposer is called, informing it of the impending destruction. At this point, the interposer can *veto* the destruction or it can allow it to take place—at which time the Notifier proceeds to phase two, the actual object destruction.

Destroy event handlers use a status parameter to determine which phase of destruction the Notifier is in—whether the Notifier is requesting if it is feasible for the client to be terminated at present (phase one, DESTROY_CHECKING) or if it is making a request to terminate (phase two, DESTROY_CLEANUP or DESTROY_PROCESS_DEATH).

The destroy-interpose function takes the following form:

```
Notify_value
destroy_func(client, status)
    Notify_client client;
    Destroy_status status;

typedef enum destroy_status {
    DESTROY_PROCESS_DEATH,
    DESTROY_CHECKING,
    DESTROY_CLEANUP,
    DESTROY_SAVE_YOURSELF,
} Destroy_status;
```

If the `status` argument is DESTROY_CHECKING and the client cannot terminate at present, the destroy event handler should call `notify_veto_destroy()`, indicating that termination would not be advisable at this time, and return normally. If the client can terminate at present, then the destroy handler should do nothing; a subsequent call will tell the client to actually destroy itself. This veto option is used, for example, to give a text subwindow the chance to ask the user to confirm the saving of any editing changes when quitting a tool.

If `status` is DESTROY_PROCESS_DEATH, then the client can count on the entire process dying and should do whatever it needs to do to clean up its outside entanglements, such as updating a file used by a text subwindow. Since the process is about to die, it need not free allocated memory which is implicitly freed by the process's termination.

Since the entire process is dying, you cannot display a notice or do any sort of prompting with the user to try to veto this request.

However, if `status` is `DESTROY_CLEANUP` then the client is asked to destroy itself and to be very tidy about cleaning up all the process internal resources that it is using, as well as its outside entanglements. This may be called on frames which are not the sole frames used by the application. If a frame is dismissed but the application is still running (e.g., dismissing a pinned up menu), then the frame should clean up—including freeing allocated memory.

If the `status` is set to `DESTROY_SAVE_YOURSELF` when the window manager has sent a `WM_SAVE_YOURSELF` message to all the clients on the desktop, this means the user may have selected the "Save Workspace" option from an OPEN LOOK window manager's menu. Basically, this means that the application should save its current state in such a way that it could be resumed in the same state at a later time. For example, a text editing program would update the file being edited, a graphics program would output its image display to a file, or whatever.

When the user selects "Save Workspace" from an OPEN LOOK window manager, XView sets its status to `DESTROY_SAVE_YOURSELF` and then checks to see if the attribute `FRAME_WM_COMMAND_ARGC_ARGV` is set. XView informs the window manager of the option and the window manager writes the application's default command-line options to `~/.openwin-init`. If `FRAME_WM_COMMAND_ARGC_ARGV` is set, the window manager appends any user-specified command-line options to the default values written to `~/.openwin-init`.

Normally `FRAME_WM_COMMAND_ARGC_ARGV` is set when an application is initialized with `xv_init()`. In the special case when there are two base frames for an application, `FRAME_WM_COMMAND_ARGC_ARGV` should be set to -1 for one of the base frames (and for additional subsequent base frames for the application). This prevents the application from appending two sets of command-line options to `~/.openwin-init`.

20.9.8.1 Interposing a client destroy handler

We present an example of interposing in front of the frame's client-destroy event handler. The following program displays a frame, a panel, and a panel button labeled "Quit." When the user chooses the panel button or the frame's "Quit" menu selection, the interposer is called and indicates the fact that the frame is about to go away.

Example 20-6. The interpose.c program

```
/*
 * interpose.c -- shows how to use an interpose destroy function
 */
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/notice.h>

Frame frame;

Notify_value
destroy_func(client, status)
```

Example 20-6. The interpose.c program (continued)

```
Notify_client client;
Destroy_status status;
{
    if (status == DESTROY_CHECKING) {
        int answer = notice_prompt(client, NULL,
            NOTICE_MESSAGE_STRINGS, "Really Quit?", NULL,
            NOTICE_BUTTON_YES,  "No",
            NOTICE_BUTTON_NO,   "Yes",
            NULL);
        if (answer == NOTICE_YES)
            notify_veto_destroy(client);
    } else if (status == DESTROY_CLEANUP) {
        puts("cleaning up");
        /* allow frame to be destroyed */
        return notify_next_destroy_func(client, status);
    } else if (status == DESTROY_SAVE_YOURSELF)
        puts("save yourself?");
    else
        puts("process death");
    return NOTIFY_DONE;
}

main (argc, argv)
int argc;
char *argv[ ];
{
    Panel panel;
    int   quit();

    xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create (NULL, FRAME,
        FRAME_LABEL,    argv[0],
        XV_WIDTH,       200,
        XV_HEIGHT,      100,
        NULL);
    notify_interpose_destroy_func(frame, destroy_func);

    panel = (Panel)xv_create (frame, PANEL, NULL);
    (void) xv_create (panel, PANEL_BUTTON,
            PANEL_LABEL_STRING,         "Quit",
            PANEL_NOTIFY_PROC,          quit,
            NULL);
    xv_main_loop(frame);
}

int
quit()
{
    xv_destroy_safe(frame);
    return XV_OK;
}
```

The first time the interposer is called, the status is DESTROY_CHECKING. Here, we display a notice prompting the user to confirm the quit. We want the *default* action to be "No," so we make the NOTICE_YES button have the label "No" and the NOTICE_NO button have the label "Yes." This is because the notice's *default* button is the NOTICE_YES button.

If the user selects the default "No" choice, then notify_veto_client() is called, vetoing the destruction request. Otherwise, the routine returns and the destruction sequence continues as usual. The routine is then called again, notifying that it is being destroyed this time. It calls notify_next_destroy_func() to allow the process to continue. This is where any process cleaning up should take place.

Notifier

20.9.8.2 Enabling panel item interposition

The attribute PANEL_POST_EVENTS saves the application from the burden of installing a base event handler on a panel item. This replaces the default event handler, which for performance reasons, does not use the notifier. Setting attribute PANEL_POST_EVENTS to TRUE replaces the default panel item event handler with another one that calls notify_post_event(). Note that setting this attribute alone makes no functional difference between the panel item and any other panel items (besides the difference in performance due to events being dealt through the notifier rather than being sent directly).

If PANEL_POST_EVENTS is TRUE, a client may then proceed to call notify_interpose_event_func() on a panel item. All interposers on panel items set up this way should be of type NOTIFY_IMMEDIATE. The interposer reaches the next or default event handler via the notify_next_event_func() call.

This attribute has no bearing whatsoever on the PANEL_EVENT_PROC mechanism. The client may still replace the default event handler for a particular event by using PANEL_EVENT_PROC.

The following example illustrates interposing on a panel item:

```
ptxt = xv_create(panel, PANEL_TEXT,
              PANEL_LABEL_STRING,  "Text:",
              PANEL_POST_EVENTS,   TRUE,
              NULL );

/* make textfield all upper case */
notify_interpose_event_func(ptxt, all_upper, NOTIFY_IMMEDIATE);

     [...............]

Notify_value
all_upper ( client, event, arg, type )
   Notify_client client;
   Notify_event event;
   Notify_arg arg;
   Notify_event_type type;
{
   int id = event_action((Event *) event);

   if ( isascii(id) && islower(id) )
     event_set_action((Event *) event, toupper(id));
```

```
    else if ( id == ACTION_PASTE )
        /* whatever, ..... */
/*
* panel_default_handle_event, text_handle_event,
* next interposer, .....
*/
return notify_next_event_func(client, event, arg, type);
}
```

20.10 Notifier Control

The Notifier is started automatically by calling `xv_main_loop()`. The function `xv_main_loop()` takes a window object (typically the base frame of the application) and sets `XV_SHOW` to `TRUE`. Then it calls `notify_start()`, and the Notifier is running. The function `notify_start()` loops through the Notifier's processing loop, waiting for events in which its clients have expressed interest. This continues until the application calls `notify_stop()` or there are no more clients registered. At this time, `notify_start()` returns and the application continues or finishes. Essentially, the Notifer enters a loop, blocks until there is input (which it dispatches), and then blocks to wait for more input.

When `notify_start()` is called, the Notifier takes over and none of the application code is executed unless it is directly or indirectly called by *callback* procedure. If the callback routine calls `notify_stop()`, when that callback function returns, `notify_start()` returns and, as a result, `xv_main_loop()` returns. The Notifier can be stopped by using this method despite that there are still clients registered with the Notifier. At this point, the application may call `notify_start()` again, or it may attempt to do either *implicit* or *explicit* dispatching of events.

Explicit dispatching is done by calling `notify_dispatch()`. Here, the Notifier steps through one iteration of its normal control loop. This allows you to monitor events during the execution of a time consuming or computationally complex portion of the program. Say your application generates a complex fractal image. The process is initiated from the selection of a panel button. Because the generation of fractals is very time consuming, you may wish to check every once in a while to see if the user has generated any events that need to be processed—like selecting a panel button labeled `Stop`. In this case, the callback procedure for the panel button should call `notify_stop()` and return. The Notifier returns to the top-level, `notify_start()` returns (and thus, `xv_main_loop()` returns), and your application begins to generate the fractal image. During each iteration of the loop (or more often, if necessary), `notify_dispatch()` is called to ensure processing of any pending events that the user might have generated.

Implicit dispatching indicates that you are going to make calls to `read()` or `select()`, system calls that *block*. If `notify_do_dispatch()` is called, then events the user generates (such as moving the mouse or selecting a panel button) will continue to be processed even though the `read()` has not yet returned.

Implicit and explicit dispatching may be used before or after the call to `xv_main_loop()` or `notify_start()`, but it is not permitted to call these functions while the Notifier is looping on its own (from within a call to `notify_start()`). Therefore, you should never attempt to do direct dispatching from within a callback routine or any function that has been called indirectly by the Notifier.

These two methods of dispatching make porting programs that do not use the Notifier much easier. Thus, building an XView interface on top of a typical mainline-based, input-driven program is also much easier. Programs written from scratch should try to follow the event-driven input style of program design and try to avoid using implicit or explicit dispatching whenever possible.

20.10.1 Mass Destruction

The following routine causes all the client destruction routines to be called immediately with a destroy status set to `status`:

```
Notify_error
notify_die(status)
   Destroy_status   status;
```

This routine causes all the client destruction functions to be called immediately with `status` as the reason. The return values are `NOTIFY_OK` or `NOTIFY_DESTROY_VETOED`; the latter indicates that someone called `notify_veto_destroy()` and status was `DESTROY_CHECKING`. It is then the responsibility of the caller of `notify_die()` to exit the process, if so desired. Refer to the discussion on `notify_post_destroy` for more information.

20.10.2 Implicit Dispatching

Implicit dispatching is used whenever you wish to loop on a call that might block, such as `read` (2). Before calling `read()`, you should first call the function: `notify_do_dispatch();` This tells the Notifier that you are going to do implicit dispatching and that it should use its own version of `read()` rather than using the standard system call as the function. The two are equivalent with one exception: `read()` will return 0 on `EOF` rather than -1, as you might expect.

After `notify_do_dispatch()` has been called, you can call `notify_dispatch()` directly (to process events you know have already been delivered), call `read()`, or both.

The following example program demonstrates how this can be done. The program creates our usual frame, panel, and “Quit” panel button, but instead of calling `xv_main_loop()`, we create a small loop which reads `stdin` waiting for typed input.

Example 20-7. The ntfy_do_dis.c program

```
/*
 * ntfy_do_dis.c -- show an example of implicit notifier dispatching
 * by calling notify_do_dispatch().  Create a frame, panel and "Quit"
 * button, and then loop on calls to read() from stdin.  Event
 * processing is still maintained because the Notifier uses its own
 * non-blocking read().
 */
#include <stdio.h>
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>

Frame frame;

main (argc, argv)
int argc;
char *argv[ ];
{
    Panel panel;
    char  buf[BUFSIZ];
    int   n, quit();

    xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create (NULL, FRAME,
        FRAME_LABEL,    argv[0],
        XV_WIDTH,       200,
        XV_HEIGHT,      100,
        XV_SHOW,        TRUE,
        NULL);

    panel = (Panel)xv_create (frame, PANEL, NULL);

    (void) xv_create (panel, PANEL_BUTTON,
            PANEL_LABEL_STRING,         "Quit",
            PANEL_NOTIFY_PROC,          quit,
            NULL);

    /* Force the frame to be displayed by flushing the server */
    XFlush(xv_get(frame, XV_DISPLAY));

    /* tell the Notifier that it should use its own read() so that it
     * can also detect and dispatch events.  This allows us to loop
     * in this code segment and still process events.
     */
    notify_do_dispatch();

    puts("Frame being displayed -- type away.");
    while ((n = read(0, buf, sizeof buf)) >= 0)
        printf("read %d bytes0, n);

    printf("read() returned %d0, n);
}

int
quit()
```

Example 20-7. The ntfy_do_dis.c program (continued)

```
{
    xv_destroy_safe(frame);
    return XV_OK;
}
```

There are several things to note here. First, because `xv_main_loop()` is not called, we need to explicitly set the attribute `XV_SHOW` to `TRUE` for the base frame. Otherwise, it will never be displayed. Also, because of the nature of event dispatching, we must *flush* the connection between the X server and the application to make sure that the frame is displayed by the time the first `read()` call returns. If this is not done, the frame is not displayed until after the first `read()` returns.

20.10.3 Explicit Dispatching

Frequently, the programmer is plagued with the following problem: A great deal of processing has to be done to process something that the user has initiated. As mentioned before, the programmer might want to generate a fractal image or compute the value of *pi* when the user selects a panel button. Any time-consuming process that requires this type of functionality should utilize explicit dispatching. It seems like an easy solution to `fork()` and let the background process perform the operation, but this is frequently an expensive operation and should be avoided if explicit dispatching is sufficient. If forking is required, then refer to Section 20.5.5, "Handling SIGCHLD," and Section 20.8.2, "Reading and Writing on Pipes."

The call `notify_dispatch()` does explicit event dispatching immediately to service an event that you know is waiting to be read and dispatched. Typically, this is called at particular locations within a control loop that is processing time consuming tasks. It is assumed that this control loop does *not* call `read()` or `select()`. Rather, explicit dispatching is used within loops that might do heavy computation such as graphics processing or number crunching. This is necessary because while the computations are busy computing, the user might be attempting to interact with the application by selecting a panel button.

If `notify_dispatch()` is called frequently enough, then performance of the user interface may still be perceived as acceptable even though the program is very busy with its computations. Architecturally, such a program is designed in a similar way with *ntfy_do_dis.c* in Section 20.10.1, "Implicit Dispatching." That is, there is a central processing loop which is doing the main work of the program. In this case, `notify_do_dispatch()` is not called.

```
#include <xview/xview.h>
 ...
int finished;

main(argc, argv)
char *argv[ ];
{
    Display *dpy;
    Frame frame;

      ...

      xv_create(NULL, FRAME, NULL);
```

```
        ...

        dpy = (Display *)xv_get(frame, XV_DISPLAY);
        /* flush everything before starting loop */
        XFlush(dpy);
        while (!finished) {
            notify_dispatch();
            XFlush(dpy);
            /* compute PI to the next place */
            process_pi();
        }
    }
```

Notice that after `notify_dispatch()`, `XFlush()` is called. It is imperative that this happen; otherwise, any Xlib calls (which result from many `xv_create()` or `xv_set()` calls) may not be displayed. You only need to call `XFlush()` once after one or more calls to a sequence of calls to `notify_dispatch()`. The rule of thumb is to flush the server whenever you want to see the latest display.

20.11 Emulating a sleep() Call

XView allows you to emulate the system call `sleep()`. You cannot simply make the call since this would cause problems for the Notifier. Example 20-8 demonstrates how XView applications can implement a "sleep."

Example 20-8. Emulating a sleep

```
#include <xview/xview.h>
#include <xview/panel.h>
#include <sys/time.h>

Frame   frame;
Panel   panel;

main(argc,argv)
int     argc;
char    *argv[ ];
{
        int     sleep_for_awhile();

        xv_init(XV_INIT_ARGS,   argc,argv, 0);

        frame = xv_create(0,FRAME, 0);

        panel = xv_create(frame,PANEL, 0);

        xv_create(panel,PANEL_BUTTON,
                PANEL_NOTIFY_PROC,      sleep_for_awhile,
                PANEL_LABEL_IMAGE,
                  panel_button_image(panel,"Sleep For Awhile",0,0),
                NULL);

        window_fit(panel);
        window_fit(frame);
```

Example 20-8. Emulating a sleep (continued)

```
        xv_main_loop(frame);
}

sleep_for_awhile()
{
        sleep(5);
}

sleep(sec)
int     sec;
{
        our_sleep(sec,0);
}

usleep(usec)
int     usec;
{
        our_sleep(0,usec);
}

our_sleep(sec,usec)
int     sec,usec;
{
        int     oldmask,mask;
        struct  timeval tv;

        tv.tv_sec = sec;
        tv.tv_usec = usec;

        mask = sigmask(SIGIO);
        mask |= sigmask(SIGALRM); /* change = to |= JLM */
        oldmask = sigblock(mask);

        if ((select(0,0,0,0,&tv)) == -1) {
                perror("select");
        }

        sigsetmask(oldmask);
}
```

Notifier

20.12 Advanced Notifier Usage

This section covers Notifier *prioritization* and *scheduling*. These topics should be considered advanced topics, since their use changes the way XView schedules and prioritizes event handling in the Notifier. If you incorrectly modify these areas of the Notifier, you may experience severe problems with your application.

NOTE

These facilities should rarely be used by clients; a client should normally rely on XView's default scheduling and prioritizing scheme.

20.12.1 Prioritization

The order in which a particular client's conditions are notified may be controlled by providing a `prioritizer` operation. Assuming asynchronous or immediate notifications have already been sent, the default prioritizer makes its notifications in the following order:

- Interval timer notifications (ITIMER_REAL and then ITIMER_VIRTUAL).
- Child process control notifications.
- Synchronous signal notifications by ascending signal numbers.
- Exception file descriptor activity notifications by ascending `fd` numbers.
- Handle client events by order in which received.
- Output file descriptor activity notifications by ascending `fd` numbers.
- Input file descriptor activity notifications by ascending `fd` numbers.

20.12.1.1 Providing a prioritizer

This section describes how a client can provide its own prioritizer.

```
Notify_func
notify_set_prioritizer_func(client, prioritizer_func)
    Notify_client  client;
    Notify_func    prioritizer_func;
```

The function `notify_set_prioritizer_func()` takes an opaque client handle and the function to call before any notifications are sent to `client`. The previous function that would have been called is returned. If this function was never defined, then the default prioritization function is returned. If the `prioritizer_func()` argument supplied is NOTIFY_FUNC_NULL, then no client prioritization is done for `client` and the default prioritizer is used.

The calling sequence of a prioritizer function is shown below:

```
Notify_value
prioritizer_func(client, nfd, ibits_ptr, obits_ptr
                 ebits_ptr, nsig, sigbits_ptr, auto_sigbits_ptr,
                 event_count_ptr, events, args)
Notify_client       client;
fd_set              *ibits_ptr, *obits_ptr, *ebits_ptr;
int                 nfd, nsig, *sigbits_ptr, *auto_sigbits_ptr;
                    *event_count_ptr;
Notify_event        *events;
```

```
Notify_arg          *args;
#define SIGBIT(sig)         (1 << ((sig) -1))
```

In this function, `client` from the associated `notify_set_prioritizer_func()` is passed to `prioritizer_func()`. In addition, all the notifications that the Notifier is planning on sending to `client` are described in the other parameters. This data reflects only data that `client` has expressed interest in by asking for notification of these conditions. The remaining arguments and the return values are described in the *XView Reference Manual*.

20.12.1.2 Dispatching events

From within a prioritization routine, the following functions are called to cause the specified notifications to be sent:

- `notify_event()`
- `notify_input()`
- `notify_output()`
- `notify_exception()`
- `notify_itimer()`
- `notify_signal()`
- `notify_wait3()`

The notifier won't send any notifications it wasn't planning on sending anyway, so one can't use these calls to drive clients programmatically. A return value of `NOTIFY_OK` indicates that `client` was sent the notification. The return value for an unknown client, `NOTIFY_UNKNOWN_CLIENT` indicates that `client` is not recognized by the Notifier and no notification was sent. A return value of `NOTIFY_NO_CONDITION` indicates that `client` does not have the requested notification pending and no notification was sent.

A client may choose to replace the default prioritizer. Alternatively, a client's prioritizer may call the default prioritizer after sending only a few notifications. Any notifications not explicitly sent by a client prioritizer will be sent by the default prioritizer (when called), in their normal turn. Once notified, a client will not receive a duplicate notification for the same event.

Signals indicated by bits in `sigbits_ptr` should call `notify_signal()`. Signals in `auto_sigbits_ptr` need special treatment:

- `SIGALRM` means that `notify_itimer()` should be called with a `which` of `ITIMER_REAL`.
- `SIGVTRM` means that `notify_itimer()` should be called with a `which` of `ITIMER_VIRTUAL`.
- `SIGCHLD` means that `notify_wait3()` should be called.
- `SIGTSTP` means `notify_destroy()` should be called with `status`

DESTROY_CHECKING.

- SIGTERM means notify_destroy() should be called with status DESTROY_CLEANUP.
- SIGKILL means notify_destroy() should be called with status DESTROY_PROCESS_DEATH.

Asynchronous signal notifications, destroy notifications, and client event notifications that were delivered right when they were posted do not pass through the prioritizer.

20.12.1.3 Getting the prioritizer

The routine notify_get_prioritizer_func() returns the current prioritizer of a client:

```
Notify_func
notify_get_prioritizer_func(client)
   Notify_client client;
```

This function takes an opaque client handle. The function that will be called before any notifications are sent to client is returned. If this function was never defined for client, then a default function is returned. A return value of NOTIFY_FUNC_NULL indicates an error. If client is unknown, then notify_errno is set to NOTIFY_UNKNOWN_CLIENT.

20.12.2 Scheduling the Notifier

Scheduling the notifier allows you to control the order in which clients are notified, which is done by a particular client's prioritizer function (refer to the previous section, "Prioritization"). The scheduler has the following calling sequence:

```
Notify_func
notify_set_scheduler_func(scheduler_func)
   Notify_func  scheduler_func;
```

The notify_set_scheduler_func() function allows you to arrange the order in which clients are called. The argument scheduler_func is the function to call to do the scheduling of clients. The previous function that would have been called is returned. This returned function will, almost always, be important to store and call later because it is most likely the default scheduler.

Replacement of the default scheduler is most often done by a client that needs to make sure that other clients don't take too much time servicing all of their notifications. For example, if doing "real-time" cursor tracking in a user process, the tracking client wants to schedule itself ahead of other clients, whenever there is input pending on the mouse.

The calling sequence of a scheduler function is:

```
Notify_value
scheduler_func(n, clients)
   int            n;
   Notify_client  *clients;
```

The argument n is passed into scheduler_func(). This is a list of clients, all of which are slated to receive some notification this time around. The scheduler scans clients and makes calls to notify_client() (refer to the next section, "Dispatching Clients"). Clients so notified should have their slots in clients set to NOTIFY_CLIENT_NULL. The return value from scheduler_func() is one of the following:

- NOTIFY_DONE – All of the clients had a chance to send notifications. The implies that no further clients should be scheduled this time around the notification loop. Unsent notifications are preserved for consideration the next time around the notification loop.
- NOTIFY_IGNORED – One or more clients were scheduled; that is, some clients may have been scheduled, but not all. This implies that another scheduler should try to schedule any clients in clients that are not NOTIFY_CLIENT_NULL.

20.12.2.1 Dispatching clients

The following routine is called from scheduler routines to cause all pending notifications for client to be sent:

```
Notify_error
notify_client(client)
   Notify_client  client;
```

The return value is NOTIFY_OK, NOTIFY_NO_CONDITION, or NOTIFY_UNKNOWN_CLIENT. The return value NOTIFY_OK indicates the client was notified. NOTIFY_NO_CONDITION indicates no conditions for client. This might mean notify_client() was already called with this client handle. NOTIFY_UNKNOWN_CLIENT indicates an unknown client.

20.12.2.2 Getting the scheduler

The following routine returns the function that will be called to do client scheduling:

```
Notify_func
notify_get_scheduler_func()
```

This function is always defined to be the default scheduler.

20.13 Error Codes

This section describes the basic error handling scheme used by the Notifier and lists the meaning of each of the possible error codes. Every call to the Notifier returns a value that indicates success or failure. On an error condition, notify_errno describes the failure. notify_errno is set by the Notifier as errno is set by UNIX system calls. (i.e., notify_errno is set only when an error is detected during a call to the Notifier. It is not reset to NOTIFY_OK on a successful call to the Notifier.)

```
enum notify_error {
    ... /* Listed below */
};
```

```
typedef enum notify_error Notify_error;

extern Notify_error notify_errno;
```

Table 20-1 contains a complete list of error codes.

Table 20-1. Notifier Error Codes

Error Code	Description
NOTIFY_OK	The call was completed successfully.
NOTIFY_UNKNOWN_CLIENT	The `client` argument is not known by the Notifier. A `notify_set_*_func` call needs to be made in order for the Notifier to recognize it.
NOTIFY_NO_CONDITION	A call was made to access the state of a condition, but the condition was not set with the Notifier for the client in question. This situation can occur when a `notify_get_*_func()` type call is made before the equivalent `notify_set_*_func()`. Also, the Notifier automatically clears some conditions after they have occurred, e.g., when an interval timer expires.
NOTIFY_BAD_ITIMER	The `which` argument to an interval timer routine was not valid.
NOTIFY_BAD_SIGNAL	The `signal` argument to a signal routine was out of range.
NOTIFY_NOT_STARTED	A call to `notify_stop()` was made, but the Notifier was never started.
NOTIFY_DESTROY_VETOED	A client refused to be destroyed during a call to `notify_die()` or `notify_post_destroy()` when `status` was `DESTROY_CHECKING`.
NOTIFY_INTERNAL_ERROR	Some internal inconsistency in the Notifier itself has been detected.
NOTIFY_SRCH	The `pid` argument to a child process control routine was not valid.
NOTIFY_BADF	The `fd` argument to an input or output routine was not valid.
NOTIFY_NOMEM	The Notifier dynamically allocates memory from the heap. This error code is generated if the allocator could not get any more memory.
NOTIFY_INVAL	Some argument to a call to the Notifier contained an invalid argument.
NOTIFY_FUNC_LIMIT	An attempt to set an interposer function has encountered the limit of the number of interposers allowed for a single condition.

The routine `notify_perror()` acts like library call `perror (3)`.

```
notify_perror(str)
    char *str;
```

`notify_perror()` prints the string `str`, followed by a colon, and followed by a string that describes `notify_errno` to `stderr`.

20.14 Issues

Here are some additional issues surrounding the Notifier:

- The layer over the UNIX signal mechanism is not complete. Signal blocking (`sigblock` (2)) can still be done safely in the flow of control of a client to protect critical portions of code as long as the previous signal mask is restored before returning to the Notifier. Signal pausing (`sigpause` (2)) is essentially done by the Notifier. Signal masking (`sigmask` (2)) can be accomplished via multiple `notify_set_ signal_func()` calls. Setting up a process signal stack (`sigstack` (2)) can still be done. Setting the signal catcher mask and on-signal-stack flag (`sigvec` (2)) could be done by reaching around the Notifier, but this is not supported.
- Not all process resources are multiplexed (e.g., `rlimit (2)`, `setjmp (2)`, `umask (2)`, `setquota (2)`, and `setpriority (2))`, only ones that have to do with flow of control multiplexing. Thus, some level of cooperation and understanding needs to exist among packages in the single process.
- One might intercept `close` (2) and `dup` (2) calls so that the Notifier is not waiting on invalid or incorrect file descriptors if a client forgets to remove its conditions from the Notifier before making these calls.
- One might intercept `signal` (3) and `sigvec` (2) calls so that the Notifier does not get confused by programs that fail to use the Notifier to manage its signals.
- One might intercept `setitimer` (2) calls so that the Notifier does not get confused by programs that fail to use the Notifier to manage interval timers.
- One might intercept `ioctl`(2) calls so that the Notifier does not get fouled up by programs that use `FIONBIO` and `FIOASYNC` instead of the equivalent `fcntl` (2) calls.
- One might intercept `readv` (2) and `write` (2) just like `read` (2) and `select` (2) so that a program does not tie up the process.
- The Notifier is not a lightweight process mechanism that maintains a stack per thread of control. However, if such a mechanism becomes available, then the Notifier will still be valuable for its support of notification-based clients.
- Client events are disjointed from UNIX events. This is done to give complete freedom to clients as to how events are defined. One could imagine certain clients wanting to unify client and UNIX events. This could be done with a layer of software on top of the Notifier. A client could define events as pointers to structures that contain event codes and event specific arguments. The event codes would include the equivalents of UNIX event notifications. The event specific arguments would contain, for example, the file descriptor of an input-pending notification. When an input-pending notification from the Notifier was sent to a client, the client would turn around and post the equivalent client event notification.

- One could imagine extending the Notifier to provide a record and replay mechanism that would drive an application. However, this is not supported by the current interface.