Open Computing ‘‘Hands-On’’: ‘‘PC-Unix Connection’’ Column: March 94

Hidden Treasure

The little-known Windowing Korn Shell included with UnixWare has great potential for nonprogrammers

By Tom Yager

[*Editor’s note:* This column installment was rewritten to use the Motif toolkit instead of the Open Look toolkit that was used in the *Open Computing* print-magazine version of this column.]

While installing UnixWare on my lab’s Mobius 486 system, I saw a brief flash of a reference to something called the Windowing Korn Shell (Wksh). It was some weeks before my curiosity drove me to play with it, but once I did, I was enthralled. The Wksh application seems to be the only tool in its class that will allow non-career programmers to create graphical applications. This month I’ll see if I can get you hooked as well.

If you’ve done any shell programming, you’re already aware that the shell is a somewhat limited environment. The fact that shell statements are interpreted, not compiled, means performance cannot be an objective. The lack of support for floating-point arithmetic precludes the creation of number-crunching programs that put up analytical graphs. In other words, the addition of graphics capabilities to the Korn Shell doesn’t magically turn it into the only programming language you’ll ever need.

Even with these limitations, the practical potential of Wksh is enormous. System administrators can use it to make users’ lives easier. Examples of the kinds of tasks for which this tool is suited include custom program launch menus and button boxes, fill-in-the-blanks problem reports, and graphical ‘‘message of the day’’ news displays.

X Window System programmers can use Wksh to prototype their interfaces. An interpreted environment is much more responsive to ‘‘what would this look like?’’ experiments than the old edit-compile-run-edit cycle. And because Wksh uses the same library functions as your C code, the applications will port quite easily to C when you’re done fooling around.

Finally, programmers can use Wksh to hang graphical faces on existing text applications. An existing shell program can easily take on a fresh graphical paint job. Some text-based compiled applications could be adapted for graphical operation through the crafting of a Wksh front end.

The Big Secret

If you haven’t heard of the Windowing Korn Shell before, don’t kick yourself. Novell hasn’t made the fuss over Wksh that it deserves. With no hard-copy documentation in the standard bundle, it’s little wonder that a lot of UnixWare and other System V Release 4.2 users don’t even know it’s there.

So what is Wksh? Simply put, it is a version of the powerful Korn shell, enhanced with support for graphical interfaces. A set of extensions lets you turn C library functions into Korn shell functions. As packaged with UnixWare, this trick is used to hook the X libraries into the shell. You can use this same scheme to connect any set of C functions, even your own, to this special Korn shell. This column will focus on the graphical extensions.

How much these extensions buy you depends on how much time you want to spend learning them. While Wksh sets aside the requirement for proficiency in C, you do need to know enough about the X Window System (more specifically, X Toolkit and Motif) programming to build and present your interfaces. For really basic operations, like popping up a notice box when it's time for lunch, special Wksh built-ins are all you need. That's good for a cheap thrill, but a meaningful relationship with this shell requires a more thorough understanding.

Jumping In

Within the UnixWare/SVR4.2 documentation there are two manuals essential to exploiting Wksh: *Graphical User Interface Programming*, included with the C development system, and the *Windowing System API Reference*, which must be purchased separately from Novell. The first manual is a programmer's guide with coverage of Wksh jammed into 60 very dense pages. You need this manual because it's the only place you'll find examples of Wksh programs. And anyone who's written X Window System applications can tell you that the best way to teach yourself is to start with existing programs and modify them.

Even though one of the manuals is included only with the C programming set, the shell itself is standard issue. So, too, are the X libraries it needs--that's one of the fringe benefits of shared libraries. It means that the Wksh programs you create will run on even Personal Edition UnixWare nodes.

Just in case you'd like to play with Wksh now on your SVR4.2 or UnixWare system, here are a couple of basic annotated examples, which may give you a head start on building more complex programs.

First, try the basic example in Listing 1A. You'll need to run the UnixWare desktop (or a third-party X implementation). Just create a file containing the text in the example, set the file to executable (with `chmod`) and run it. The first line in the script directs the system to use Wksh running with the Motif libraries enabled.

This example pops up a confirmation dialog box with ''ok,'' ''cancel'' and ''help'' buttons containing the message in the command's first argument as shown:

If you click on the ''ok'' button, the second argument (`echo ok; exit`), a Wksh statement, is executed.

If you click on ‘‘cancel,’’ the statement in the third argument (`echo cancel; exit`) executes. This procedure is common to Wksh operations. It manages to turn the Korn shell into an event-driven language by tying user input (events) to actions. Actions are always represented as shell statements, so you gain the flexibility of choosing a simple response (like the ‘‘echo’’ statements in this example) or a complex one in which the input triggers a Wksh function or invokes an arbitrary external program.

The `confirm` function is a convenience because it doesn’t require you to dig into the X Window System programming interface at all. However, the somewhat more complex example in Part B of Listing 1 does.

If you’re a seasoned X Window System programmer, you should find this example familiar. It’s virtually identical to a typical first or second X Window System project. It puts up a small window with a button in it tagged ‘‘Press me.’’ When you click on the button, the program exits.

If the flow isn’t clear to you, don’t worry. The seemingly redundant function arguments can easily throw you off. Let’s look at each line in turn. The first call, `XtAppInitialize`, is a convenience function in the toolkit library. It combines all of the display initialization functions into a single call. The `TOPLEVEL` argument is a special Wksh variable that provides a hook for your first visible interface object (a button, in this case). The other two arguments refer to the application’s internal name and to the title for the window’s title bar.

The `XtCreateManagedWidget` call in the next line attaches the program’s only interface object--called a widget by the X Window System folks--to the invisible top-level window created by the `XtAppInitialize` call. The first argument is the Korn shell variable the function uses to hold a widget ID. You use this ID to refer to the widget in later calls. The next argument is an internal name used by the library to tag the widget.

Then comes the widget class, where you specify the kind of widget you want to display. The *Windowing System API Reference* provides the master list of the Motif widgets available to your Wksh programs. You can use virtually any of the objects Motif offers, from buttons and scrollbars to multiline text editors. In this case, the `pushButton` class refers to one of Motif’s basic button objects. The fourth argument on the second line tells the button what to attach to, in this case the top-level window, and the fifth argument is another nifty Wksh convenience: a direct resource assignment.

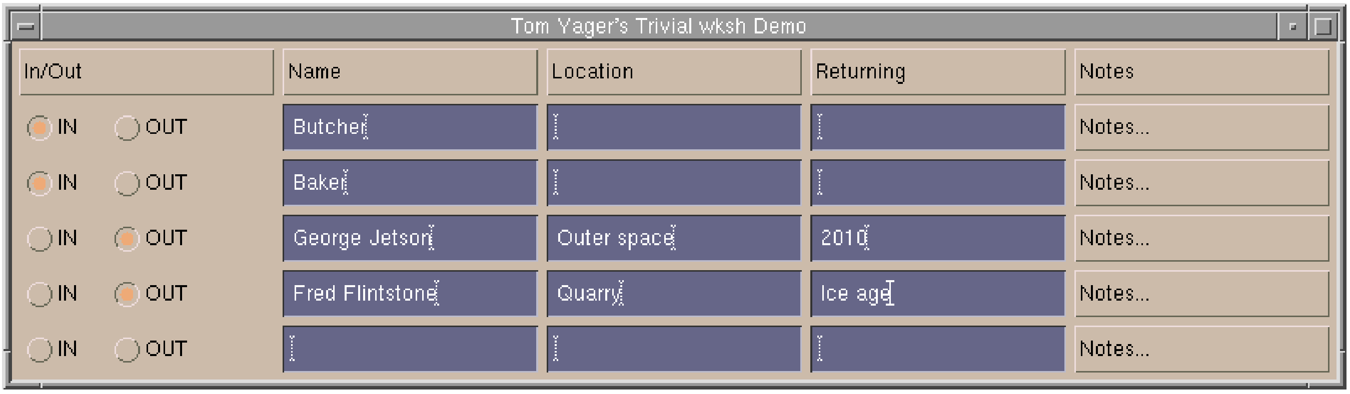
Resources are named properties that each widget uses to figure out how to look and behave. Unlike C, Wksh offers a convenient notation for altering resources. As the last argument to the function shows, the scheme is gloriously simple: the resource name, a colon, and the value you’d like it to assume. The `labelString` resource for the `pushButton` widget holds the text that appears on the button’s face.

The `XtAddCallback` (line 3) function tells a widget what to do in response to a specific user event. Here the activate event, referring to a user clicking the mouse over the graphical button, will simply exit. You can specify any valid shell statement as a callback; the `exit 0` could easily be replaced with `echo Ouch!` to display a word in the xterm window used to launch the shell each time the button is clicked.

The rest is child's play. The `XtRealizeWidget` (line 4) call makes the widget referred to as the function's argument, plus all the widgets below it, visible. The last function, `XtMainLoop`, locks into the processing loop that puts up the window with the button in it, listens for user events (in this case, a click of the button), and fires off the actions associated with those events. This loop never exits on its own. You should register a callback, as in the example, that exits the shell program in response to some user input.

Once `XtMainLoop` is running, all flow through your shell program will be initiated through callbacks. That's the nature of an event-driven program, and most shell programmers aren't used to doing things that way. As a result, your first few Wksh scripts might appear very strange. But resist the urge to do things the old-fashioned way by trying to maintain a linear flow through your shell code.

To help readers who don't have access to the programming documentation for Wksh, Listing 2 provides a more significant, heavily-commented example. The comments should help you understand how the program functions. The output is shown here:



UnixWare includes examples that are installed, by default, in `/usr/X/lib/wksh/xmexamples`. However, the worth of these examples is diminished by an overuse of abbreviations. Wksh recognizes a shorthand for most Motif toolkit calls, but the result is unreadable code that is much harder to translate to C if you're using Wksh for prototyping.

If you're working with UnixWare (or another version of SVR4.2), whether as an administrator, integrator, programmer, or user, you certainly have some problem that could be solved by a well-written Wksh program. If you've never dabbled in X Window System programming before, don't be intimidated. It's worth the effort to learn enough to create your own simple custom programs. Now that Wksh is here, the effort and cost have been reduced considerably.

References

UNIX SVR4.2 Windowing System API Reference. Englewood Cliffs, N.J.: Prentice-Hall (UNIX Press imprint). ISBN 0-13-017716-4

Flanagan, David (editor). *Volume 5: X Toolkit Intrinsics Reference Manual, 3rd Edition, April 1992*. Sebastopol, CA.: O'Reilly and Associates. ISBN: 1-56592-007-4. O'Reilly's descriptive Web page (http://www.ora.com/gnn/bus/ora/item/v5.html).

UnixWare OSF/Motif and Wksh Programmer's Guides: refer to UnixWare on-line documentation

---

Edited by Becca Thomas / Online Editor / UnixWorld Online / editor@unixworld.com

Go To Content   Search

Last Modified: Wednesday, 17-Jan-96 14:31:40 PST