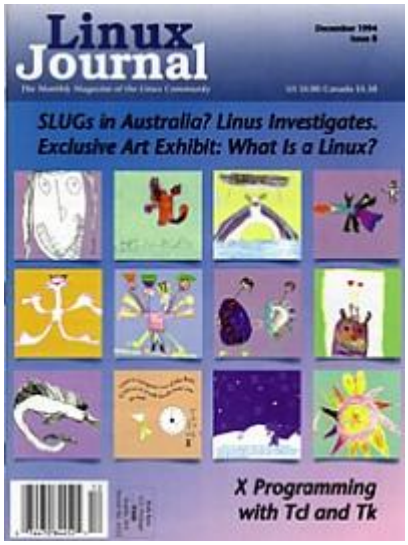
Linux Journal Issue #8/December 1994



Features

News & Articles

Reviews

Columns

Letters to the Editor
Stop the Presses
New Products
Kernel Korner *by Michael K. Johnson*

Archive Index

Advanced search

LINUX JOURNAL

X Window System Programming with Tcl and Tk

Matt Welsh

Issue #8, December 1994

Tcl and Tk will allow you to develop X Window System applications by writing simple, interpreted scripts. Learn how to unlock the power of X throught this unique programming paradigm.

Anyone who has ever programmed X applications through the Xlib interface, or the arcane X Toolkit Intrinsics, or even Motif, knows that it is an experience that cannot be fully appreciated without beating one's head against the desk several times an hour. X programming at the C level can often be quite complex, forcing the programmer to concentrate on messy technical issues instead of simply building an interface. (Of course, the tradeoff for this complexity is power and flexibility.) That's why we have Tcl and Tk.

Tcl (Tool Command Language) is an interpreted script language not unlike C Shell or Perl. It provides all of the basic facilities that you'd expect in such a language: variables, procedures, loops, file I/O, and so forth; nothing too flashy or bizarre. If you've ever written shell scripts you'll find Tcl quite easy to pick up.

What makes Tcl special is that it can be embedded in other applications. That is, the Tcl interpreter is a library of C routines which you can call from your own program. For example; let's say that you're writing a debugger, similar in nature to gdb.You need routines to allow the user to enter commands (such as “step 10” or “breakpoint foot c:23”) at a prompt. You'd also like to allow the user to customize the debugging environment by writing new command procedures or modifying state variables. A good way tohandle the user interface would be with Tcl. You can link the Tcl interpreter to your application, and all of the features of that language will be available. The user's commands, entered at the debugger prompt, would be executed by the Tcl interpreter, which could call C functions that you have written. If the user needs to customize aspects of the application, they can write Tcl scripts to implement new commands, and so forth.

Tcl itself may not be very exciting, but coupled with Tk it certainly is. Tk is a set of extensions to Tcl which implement commands for writing X Window System applications. These commands allow you to create buttons, scrollbars, text entry widgets, menus, and much more; permitting you to write X applications as simple Tcl scripts. With Tcl and Tk, there's little need to learn the C library interface for X programming; Tcl and Tk provide access to many X Window System facilities. In addition, you can embed Tcl and Tk in your own applications, written in C, to greatly enhance the power of this system.

This article is the first in a series on Tcl and Tk. In this article, I'm going to describe how to write simple X programs as Tcl/Tk scripts. Next month, I'm going to describe how to use the Tcl/Tk interpreter in conjunction with programs written in C or Perl.

The syntax of the Tcl language itself is relatively straightforward and will be obvious to anyone who has programmed, say, shell or Perl scripts. For this reason, I'm not going to spend much time describing the syntax of the Tcl language itself; I'm going to concentrate on the X- specific features of the Tk toolkit. See the sidebar, “Getting Tcl and Tk”, for other sources of information on Tcl.

Basic Tcl/Tk Scripts

Let's dive right in with a simple Tcl/Tk program. The Tcl/Tk interpreter is named wish (“window shell”). Assuming that you have Tcl/Tk installed on your system, and X running, you should be able to execute wish. You will be presented with a blank, rectangular window, and a wish prompt.

You can type Tcl/Tk commands at this prompt, and the results will be displayed in the wish window. For example, if you type: button .b -text “Hello, world!” -command { exit } pack .b

the wish window should reduce to a single button containing the string “Hello, world!”. (See Figure 1. next page) Pressing this button will cause the wish process to exit.

What did we just do? Every Tcl command begins with a command name, followed by any arguments. button is a Tk command which creates a button. In this case, we want the button to contain the text “Hello, world!”, and we wish it to execute the Tcl exit command when it is pressed. The first argument to button,“.b”, is the name that we wish to give to the button widget.

Figure 1

(A widget is simply a graphical object, such as a button, scrollbar, etc., which has certain visual and functional properties. Tk supports many types of widgets, as we'll see.) We can later refer to the button with this name.

The pack command is a geometry manager; it controls how widgets are placed within the wish window. pack is a simple geometry manager which “packs” (hence the name) widgets one at a time next to each other. The widget is not made visible until it is given a position with pack.

In this case, we wish to pack the button widget into the wish window. pack can take a number of arguments to specify the position of the widget relative to other widgets. However, here we have but one widget, so the default behavior is acceptable.

Instead of typing commands to the wish process, you can write scripts to be executed via wish. Here's a simple program which will prompt you for a filename, and then start an xterm running vi to edit the file.

```
#!/usr/local/bin/wish -f
label .1 -text "Filename:"
entry .e -relief sunken -width 30 -textvariable\
  fname
pack .1 -side left
pack .e -side left -padx
lm -pady lm
bind .e <Return> {
exec xterm -e vi $fname
```

If you save the above in a file named **edit.tcl**, and then run it, you should be presented with a window as in Figure 2. (This assumes, of course, that you have wish installed in **/usr/local/bin**. Edit the pathname on the first line of the script if not.) Just as with shell scripts, you will need to make the file executable before running it.

Let's walk through this script. The first line is a **label** command, which (as you might guess) creates a label widget. A label contains only a static text string. We name this widget **.1** (more on widget naming conventions later), and give it the text value “Filename:”.

The second line creates an entry widget, named **.e**. An entry widget is like a label, except that it allows the user to edit the text. The **-relief sunken** option indicates that the widget should appear as though it is “recessed” in the window, as you can see in Figure 2. The **-width** option sets the width of the entry widget in characters, and the **-textvariable** option indicates that the value of the entry text should be stored in the variable **fname**.

The next two lines pack the label widget, followed by the entry widget, into the wish window. In both cases we specify **-side left**, which indicates that the widgets should be packed into the left side of the window, one after the other. When packing the entry widget, we use the -padx and **-pady** options to leave a bit of “paddingH (here, one millimeter) around the sides of the widget. The Tcl **pack** man page describes these options in detail.

The last three lines of our script use the **bind** command to create an event binding for our entry widget. A binding allows you to execute a series of commands when a certain event occurs in a widget; for example, a mouse button click or a keypress. In this case, we wish to execute a command whenever the RETURN key is pressed in the entry widget. The command:

```
exec xterm -e vi $fname
```

will start up an **xterm** running **vi** on the filename entered by the user. Note the use of the fname variable,

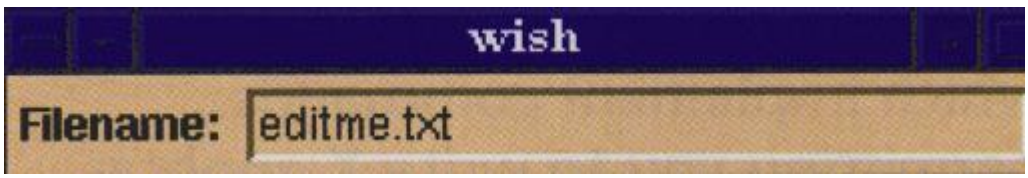


Figure 2

which we associated with the entry widget on the second line of the script. The variable name is prefixed with a dollar sign, to refer to the value of fname itself. This is similar to the variable syntax used in shell scripts.

Using Braces

One short note about Tcl syntax: Braces ({ ... }) are used to group a set of Tcl commands together as a ”sub-script“. The commands contained within the braces are passed to the Td interpreter without performing variable substitution. This is an important concept to understand. Without the braces, the Tcl interpreter would have attempted to substitute the value of the variable **fname** while interpreting the bind command in the script. That is, when the **bind** command is first executed, the variable fname has no value. Without braces, the Tcl interpreter would complain that fname is an unknown variable.

Using braces, however, we delay the interpretation of **$fname** until the event binding is actually executed; in this case, when Return is pressed in the entry widget.

Tk is a set of extensions to Tcl which implement commands... permitting you to write X applications as simple Tcl scripts.

Note that Tcl has several odd rules with respect to line breaks. Tcl expects each command to consist of a single line; the end of a line indicates the end of a command, unless the line ends with a backslash, the same as in shell scripts. However, if a line ends in an opening brace, Tcl understands that you are beginning a sub-script, to be contained within braces, and continues to read the script until a closing brace. For this reason, you can't say:

```
bind .e <Return>
{
exec xterm -e vi $fname
{
```

Tcl will think that the bind command ends after the first line, and complain that it needs a script to execute for the event binding. Therefore, when using braces to encapsulate a sub-script, be sure that the opening brace is at the end of the line beginning the script.

Naming Widgets

In Tk, widgets are named in a hierarchial fashion. The topmost ”shellH widget (that is, the main wish window) is named “ . ” (dot). All widgets which are direct childIen of . are given names beginning with ., such as **.b,.entry,.leftscroll**, and so forth. The widget name can be any alphanumeric string beginning with a dot; you choose the widget name when you create the widget, using commands such as button and label. Further subwidgets are given names such as **.foo.bar.bar**, where each level is separated with a dot.

Creating a Menu Bar Listing

For example, you might have a menu bar widget named **.mbar**. It is a child, of course, of ., the main window. Menu buttons contained within the menu bar might be named **.mbar.file,.mbar.options,** and so forth. That is, the menu bar is a child of the main application window, and the individual menu buttons are children of the menu bar. Arranging widgets into a hierarchy allows you to group them together for logical and visual purposes. I'll cover this in more detail later.

A Real Application

In order to demonstrate the power of Tcl/Tk, I'm going to present an actual application, written entirely as a Tcl/Tk script. As I go along I will describe the syntax used and the features available. You can use the Tcl/Tk man pages to fill in the gaps.

This program is a simple drawing application, utilizing the Tk canvas widget. The canvas is a simple graphics display widget which will display various kinds of objects: rectangles, lines, text, ovals, and so forth. What we're going to do is combine the canvas widget with the user interface capabilities of Tk to allow the user to draw objects using the mouse.

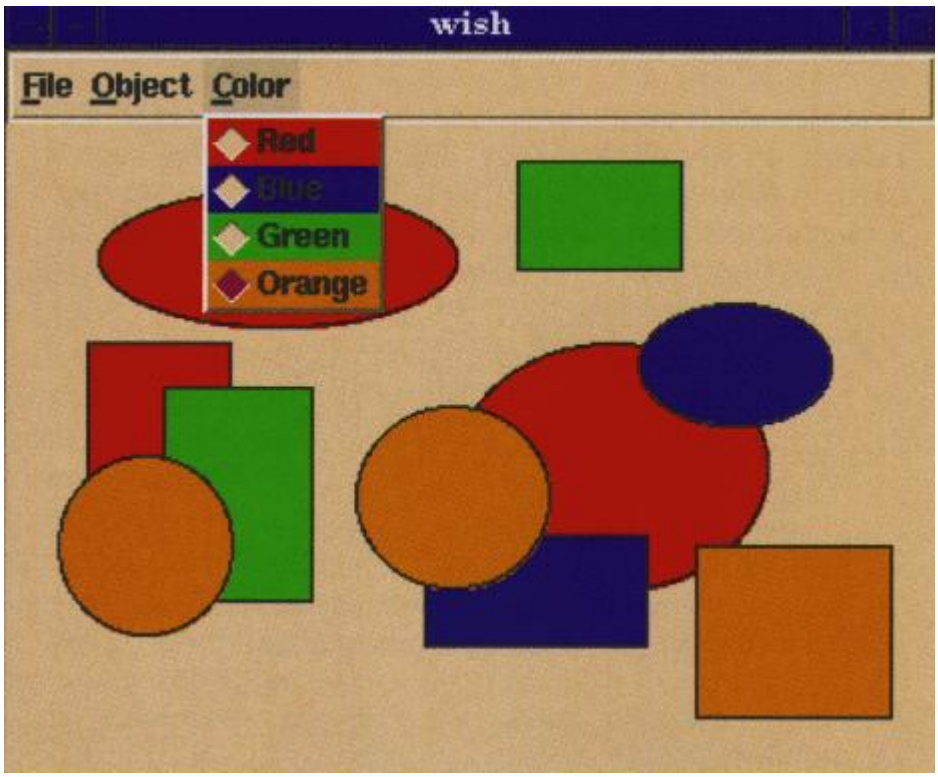


Figure 3

Figure 3 demonstrates what our application looks like when used. The Color menu has been pulled down so that you can see the various selections available.

The entire script, **draw.tcl**, is given here. Note that it's less than 200 lines long. This is amazingly short for such a complicated X application involving menus, colors, mouse input, and so forth. If you don't feel like entering this entire script, the code is available via ftp from sunsite.unc.edu, in the directory **/pub/Linux/docs/LJ**.

Creating the Menu Bar

At first glance, this script might appear to be complicated. There are a few rough spots, but the overall concepts presented here are quite simple. Let's take a closer look at this program, but let's start near the middle of the script, where we create the frame widget:

```
frame .mbar -relief groove -bd
pack .mbar -side top -expand yes -fill x
```

The frame command creates a frame widget, which is used to group widgets together. The frame itself is usually invisible, unless you specify that a border should be drawn around it.

Here we create a frame named **.mbar**, and specify that it should use the groove relief type. The "relief" of a widget indicates what kind of 3D border should appear around the widget. (Many options, such as **-relief**, **bd**, **-foreground**, and **-background** are supported by all widget types.) The valid types for **-relief** are:

- **raised**: Makes widget appear to be raised on display.
- **sunken**: Makes widget appear to sink into display.
- **ridge**: Draw raised ridge around widget border.
- **groove**: Draw sunken groove around widget border.
- **flat**: No relief; appear as if flat.

The **-ted** option specifies the border width to use for the widget (in this case, the width of the groove). Here, we set the border width to 3 pixels.

Next, we pack **.mbar** into the **wish** window. (By default, widgets are packed into their direct parent. In this case, the parent of **.mbar** is ., the topmost window). The **-side** argument to pack indicates which side of the parent we should pack **.mbar** into. The **-expand** yes option indicates that the widget should be given all of the extra space around it. Because we are packing the widget into the top edge of the window, the **-expand** option gives the widget any extra horizontal space to its left and right. The **-fill x** command causes the widget to grow until it fills this space. Using -expand yes without **-fill** would give the widget the extra horizontal space, but the widget wouldn't grow to fill that space. (If you're interested in how this works, experiment with the pack command in various forms. Also, see the pack man page or Ousterhout's book for more details.)

Creating Menus

Having created and packed our menu bar, we create three menu items, using the menubutton command:

```
menubutton .mbar.file -text "File" -underline 0 \
        -menu .mbar.file.menu
menubutton .mbar.obj -text "Object" -underline 0 \
        -menu .mbar.obj.menu
menubutton .mbar.color -text "Color" -underline 0 \
        -menu .mbar.color.menu
pack .mbar.file .mbar.obj .mbar.color -side left
```

A menubutton widget is simply a button which, when pressed, causes a menu to appear below it. First, note that the menu buttons are child widgets of the **.mbar** widget. The **-text** option specifies the text to be displayed within the menubutton, and the **-underline** option specifies the index of the letter in the menubutton text to underline for keyboard shortcuts. In each case, we underline the first letter of the string (see: Figure 3).

The **-menu** option specifies the name of the menu widget which should appear when the button is pressed. We haven't created these widgets (**.mbar.file.menu**, and so on) yet, but do so immediately thereafter.

Populating Menus

First, we create the File men

```
menu .mbar.file.menu
        .mbar.file.menu add command -label \
                "Save PostScript..." -command { get_ps
        .mbar.file.menu add command -label "Quit"
                -command { exit }
```

The menu command creates a menu widget with the given name. (Note that the menu **.mbar.file.menu** is a child widget of the menubutton **.mbar.file**). We then add menu entries to this widget using the add menu widget command.

Note that widget names themselves are commands. In general, if we want to perform a function on a specific widget, we invoke it as a command and use various widget subcommands. For example, to modify the appearance of a widget, we can use the widget subcommand **configure**:

```
<widget name> configure [ <options> ... ]
```

For example,

```
.mbar configure -background blue
```

would change the **.mbar** widget background color to blue. The widget man pages included with Tcl/Tk describe which subcommands are available with each widget type.

In the example above, we use the menu subcommand add to add entries to a menu. The syntax is

```
<widget name> add <entry type> [ <options> ... ]
```

where **<entry type>** is one of:

command-: Like a button widget, invokes a Tcl command when selected. radictutton- A group of radicLutton entries controls the value of a named variable. One radiobutton in the group (that is, only one radiobutton associated with a given variable) may be activated at any given time.

checkbutton: Similar to radicLutton. Will toggle the value of a variable to either 0 (off) or 1 (on). Unlike radiobuttons, how“ ever, checkbuttons are not mutually exclusive with one another.

cascade: Allows you to ”cascade“ sub-menus within the current menu.

separator: A nonfunctional menu separator, used to visually divide menu entries.

The **menu** man page describes these in more detail. Within the File menu, we create two command entries. When selected, each entry executes the command given by the **-command** argument. The **-label** argument, as you can guess, assigns a text label to the menu entry.

The Object menu is similar in nature to the File menu, except that it uses **radiobutton** entries. These entries are linked to the variable **object_type**. When we select the Ovals option from the menu, **object_type** is set to oval. Likewise, when selecting Rectangles, the variable is set to rect. Because these are radiobuttons, only one may be selected at a time. Later, we'll see how the **object_type** variable affects object drawing within the canvas widget.

The Color menu is like the Object menu: We have four radiobutton entries linked to the variable **thecolor**. We use the **-background** option with these entries to visually depict the color being selected.

After creating the menus, we use the **tk_menubar** command to tell Tk that this is the primary menu bar for our application. This enables keyboard shortcuts for the menu bar. If you press Alt along with one of the underlined letters in a menu title, that menu will be activated. For example, pressing **Alt-F** will activate the File menu. The **-underline** argument to the menubutton command controls which letter activates which menu. Pressing **F10** activates the leftmost menu, and you can use the arrow keys to move around.

The focus command is used to cause all keyboard events in the application window to be received by the menu bar. Otherwise, the mouse pointer would have to be within the menu bar for the keyboard shortcut events to be

received. This is yet another fine point about X programming which you shouldn't concern yourself with at this point.

Defining a Procedure

The first menu entry on the File menu, ”Save PostScript...“, executes the command get_ps, which is a procedure which we have defined earlier in the script. The proc command is used to define procedures; the syntax is:

```
proc <procedure name> <arguments> <body>
```

where **<procedure name>** is, of couse, the name of the procedure to define, **<arguments>** is a bracketed list of arguments to the procedure, and **<body>** is the script to execute when the procedure is called.

Take a look at the **get_ps** procedure. It uses the canvas widget postscript subcommand, which saves a PostScript image of the canvas widget to a file. This is a very handy feature, which we can use to ”save“ our drawing (possibly to print out or view with Ghostview).

Creating a Dialog Box

The **get_ps** command displays a dialog box asking for the name of the file to save, and two buttons, Okay and Cancel (see Figure 4, page 28).

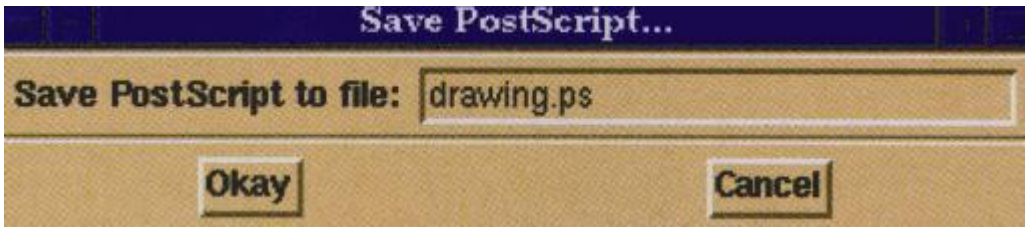


Figure 4

The dialog box, which is a separate window, is actually a **toplevel** widget. We use the **-class** option to toplevel to change the window class; this has to do with the X resource database settings for the new window (not an important detail here). We name our new **toplevel** widget **.ask**, and use the **wm** (window manager) command to set the title for the window.

Within the toplevel widget, we create two frames, **.ask.top** and **.ask.bottom**. These will serve to group widgets together. We wish to have a label and a text entry widget in the top frame, and the two buttons in the bottom frame. (This is for visual effect only: using frames is a very good way to group widgets together when using pack). Therefore, we create the frames using the frame command and pack them into the toplevel task. Nothing new here.

Within the top frame, we create a label and an entry. This is equivalent to our first example script, **edit.tcl**. Note that the label (**.ask.top.l**) and the entry (**.ask.top.e**) are children of the frame widget, which is in turn a child of **.ask**. Also, we bind the Return key in the entry widget to execute the postscript subcommand of the canvas widget, **.c** (which we will create later in the script), and destroy the **.ask** widget. This has the effect of ”popping down“ the dialog box.

In the lower frame, we create two button widgets and bind similar commands to them. This should be selfexplanatory. Finally, we use the grab command. This causes mouse and keyboard events to be confined to the dialog box window. Otherwise, you would be able to continue drawing within the main application window while the Save PostScnpt dialog box was active; we certainly wouldn't want that.

The Canvas Widget and Event Bindings

Having dealt with the menus, we are ready to tackle the canvas widget, which will be used for drawing. First, we create the widget and pack it into the application window. Next, we create two event bindings within the canvas: one for **<ButtonPress-1>** (executed when mouse button 1 is depressed) and one for **<B1-Motion>** (executed when the mouse is moved while button 1 is pressed).

The event names used with bind are the standard X11 event specifiers. These are described in any book on X11 programming (as well as good X user guides). There are too many types of X events to enumerate here; see the header file **/usr/include/X11/X.h** for a list of event names.

When button 1 is depressed in the canvas widget, we wish to start drawing an object of the type specified by the **object_type** variable, in the color **thecolor**. First, we set the global variables **orig_x** and **orig_y** to the original position of the mouse click; this defines the upper-left-hand comer of the object to draw. As the comments say, the pseudo-variables **sx** and **sy** refer to the **%x** and **%y** coordinates of the event.

Next, we use the canvas create subcommand to create an object. The syntax is:

```
<canvas name> create `(type> <xl> <yl> <x2>
<y2&gt: \
[ <options> ... ]
```

This will create an object of type **<type>** with upper-lefthand comer at **<xl>** ,**<yl>** and lower-right-hand corner at **<x2>**, **<y27gt;**. The valid object types are **arc**, **bitmap**, **line**, **oval**, **polygon**, **rectangle**, **text**, and **window**. The Tk canvas man page describes them all.

The canvas create subcommand returns a unique identifier (just an integer) for the object just created. A Tcl/Tk command contained within square brackets ([. . .]) is used to run a sub-script, the retum value of which will be substituted in its place. We assign the retum value of the create subcommand to the variable theitem. We will use this value, later, to resize the object when the mouse is dragged.

The binding for **<B1-Motion>** is very similar. First, we delete the item with the identifier given by the variable **theitem**, and then re-create the item with the new lower-right-hand comer defined by the current position of the mouse. The original upper-left-hand corner has

been saved in the variables **orig_x** and **orig_y**, and we re-use them here. We save the new object identifier back in the **theitem** variable. What we have essentially done is deleted the current object, and re-created it with a new size based on the mouse position during the drag. The visual effect of this is that the item is resized while we drag the mouse.

The last few lines of our script invoke the first entries in the Object and Color menus. This enables the oval object type, and the color of red, just as if we had selected these menu items with the mouse. If we did not do this, no object type or color would be selected when the application started. Of course, we could have set the variables object_type and thecolorby hand; however, the radiobutton entries in the menu would not be highlighted to correspond with those variable settings. Using the menu item invoke subcommand solves both problems at once.

There you have it! A complete X drawing application, complete with colors, menus, and PostScript capabilities, all in a few hundred lines of interpreted Tcl/ Tk script.

Along with the man pages and the information in this article, you should be ready to explore Tcl and Tk on your own.

As you can see, Tcl/Tk programming is easy; it's an ideal way to write simple X applications, or add X frontends to your favorite utilities. There's a complete text edit widget which will allow you to interface with other textbased applications, as well. And Tcl/Tk is extremely customizable; everything from the keyboard and mouse widget bindings to the fonts and highlight colors can be modified.

Writing entire applications as Tcl/Tk scripts may not suit your needs, however. In next month's article, I'm going to descibe how to use the Tcl/Tk interpreter, wish, as a ”server" for X interface requests from a C or Perl program. (You can

even draw directly to Tk windows using lower-level Xlib function calls from C.) This will allow you to write complicated X-based programs without having to dabble in the Xt Intrinsics or Motif.

Happy hacking.

Matt Welsh (mdw@sunsite.unc.edu) is a writer and code grunt working with the Linux Documentation Project and the Debian development team. The author welcomes questions and comments.

Archive Index Issue Table of Contents

Advanced search

Introducing Modula-3

Geoff Wyant

Issue #8, December 1994

One of the main tenets of the Unix philosophy is using the right tool for the right job. Here is a well-crafted tool well-suited for many large jobs that are diffficult to do well in C.

Suppose you want to develop a large, complex application for Linox. The application is going to be multiprocess, perhaps distributed, and definitely has to have a GUI. You want to build this application fairly quickly, and you want it to be relatively bug free.

One of the first questions you might ask yourself is “What programming language and environment should I use?” C might be a good choice, but probably not for this project. It doesn't scale as well as you'd like, and the tools for doing multiprocess/distributed programming for C just aren't there. You might consider C++, but the language is fairly complex. Also, you and others have discovered from past experience that a fair amount of time goes into debugging subtle memory management problems.

There is an alternative, the Modula-3 programming system from Digital Equipment Corporation's Systems Research Center (SRC). Modula-3 is a modern, modular, object-oriented language. The language features garbage collection, exception handling, run-time typing, generics, and support for multithreaded applications. The SRC implementation of this language features a native-code compiler; an incremental, generational, conservative, multithreaded garbage collector (whew!); a minimal recompilation system; a debugger; a rich set of libraries; support for building distributed applications; a distributed objectoriented scripting language; and, finally, a graphical user interface builder for distributed applications. In short, the ideal environment for the type of application outlined above. Moreover, the system is freely available in source form, and pre-built Linux binaries are available as well.

The remainder of this article will touch on the pertinent features of the language and provide an overview of the libraries and tools.

The Basics of the Modula-3 Language

One of the principal goals for the Modula-3 language was to be simple and comprehensible, yet suitable for building large, robust, long-lived applications and systems. The language design process was one of consolidation and not innovation; that is, the goal was to consolidate ideas from several different languages, ideas that had proven useful for building large sophisticated systems.

Features of Modula-3

You can think of Modula-3 as starting with Pascal and re-inventing it to make it suitable for real systems development. Beginning with a Pascal-like base, features were integrated that were deemed necessary for writing real applications. These features fall roughly into two areas: those which make the language more suitable for structuring large systems, and those which make it possible to do “machine-level” programming. Real applications need both of these.

Supporting Large Systems Development

There are several features in Modula-3 that support structuring of large systems. First is the separation of interface from implementation. This allows for system evolution as implementations evolve without affecting the clients of those interfaces; no one is dependent on how you implement something, only what you implement. As long as the what stays constant, the how can change as much as is needed.

Secondly, it provides a simple single-inheritance object system. There is a fair amount of controversy over what the proper model for multiple inheritance (MI) is. I have built systems that use multiple-inheritance extensively and have implemented programming environments for a language that supports MI. Experience has taught me that MI can complicate a language tremendously (both conceptually and in terms of implementation) and can also complicate applications.

Modula-3 has a particularly simple definition of an object. In Modula-3, an object is a record on the heap with an associated method suite. The data fields of the object define the state and the method suite defines the behavior. The Modula-3 language allows the state of an object to be hidden in an implementation module with only the behavior visible in the interface. This is different than C++ where a class definition lists both the member data and

member function. The C++ model reveals what is essentially private information (namely the state) to the entire world. With Modula-3 objects, what should be private can really be private.

One of the most important features in Modula-3 is garbage collection. Garbage collection really enables robust, long-lived systems. Without garbage collection, you need to define conventions about who owns a piece of storage. For instance, if I pass you a pointer to a structure, are you allowed to store that pointer somewhere? If so, who is responsible for de-allocating the structure in the future? You or me? Programmers wind up adopting such conventions as the explicit use of reference counts to determine when it is safe to deallocate storage. Unfortunately, programmers are not very good about following conventions. The net result is that programs develop storage leaks or the same piece of storage is mistakenly used for two different purposes. Also, in error situations, it may be difficult to free the storage. In C, a longjmp may cause storage to be lost if the procedure being unwound doesn't get a chance to clean up. Exception handling in C++ has the same problems. In general, it is very difficult to manually reclaim storage in the face of failure. Having garbage collection in the language removes all of these problems. Better yet, the garbage collector that is provided with the SRC implementation of Modula-3 has excellent performance. It is the result of several years of production use and tuning.

Most modern systems and applications have some flavor of asynchrony in them. Certainly all GUI-based applications are essentially asynchronous. Inputs to a GUI-based application are driven by the user. Multiprocess and multi-machine applications are essentially asynchronous as well. Given this, it is surprising that very few languages provide any support at all for managing concurrency. Instead, they “leave it up to the programmer”. More often than not, programmers do this through the use of timers and signal handlers. While this approach suffices for fairly simple applications, it quickly falls apart as applications grow in complexity or when an application uses two different libraries, both of which try to implement concurrency in their own way. If you have ever programmed with Xt or Motif, then you are aware of the problems with nested event loops. There needs to be some standard mechanism for concurrency.

Modula-3 provides such a standard interface for creating threads. In addition, the language itself includes support for managing locks. The standard libraries provided in the SRC implementation are all thread-safe. Trestle, which is a library providing an interface to X, is not only thread-safe, but itself uses threads to carry out long operations in the background. With a Trestle-based application, you can create a thread to carry out some potentially long-running operation in response to a mouse-button click. This thread runs in the

background without tying up the user interface. It is a lot simpler and less error prone than trying to accomplish the same thing with signal handlers and timers.

Generic interfaces and modules are a key to reuse. One of the principal uses is in defining container types such as stacks, lists, and queues. They allow container objects to be independent of the type of entity contained. Thus, one needs to define only a single “Table” interface that is then instantiated to provide the needed kind of “Table”, whether an integer table, or a floatingpoint table or some other type of table is needed. Modula-3 generics are cleaner than C++ parameterized types, but provide much of the same flexibility.

Supporting Machine-Level Programming

One of the important lessons from C was that there are times that real systems need to be programmed essentially at the machine level. This power has been nicely integrated into the Modula-3.

Any module that is marked as unsafe has full access to machine-dependent operations such as pointer arithmetic, unconstrained allocation and de-allocation of memory, and machine-dependent arithmetic. These capabilities are exploited in the implementation of the Modula-3 I/O system. The lowest levels of the I/O system are written to make heavy use of machine-dependent operations to eliminate bottlenecks.

In addition, existing (non-Module-3) libraries can be imported. Many existing C libraries make extensive use of machine-dependent operations. These can be imported as “unsafe” interfaces. Then, safer interfaces can be built on top of these while still allowing access to the unsafe features of the libraries for those applications that need them.

The Modula-3 Programming System

How often have you avoided changing a base header file in a C/C++ system because you didn't want to recompile the world? How many times have you restructured your header files, not because it was the right thing to do, but because you needed to cut down on the number of recompilations after each change?

The SRC implementation of Modula-3 has a rather elegant solution to this problem. If an item in an interface is changed, only those units that depend on that particular item will be recompiled. That is, dependencies are recorded on an item basis, not on an interface file basis. This means much less recompilation after each set of changes.

m3gdb is a version of GDB that has been modified to understand and debug Modula-3 programs. One of the nice features of m3gOb is that it understands M3 threads and allows you to switch from thread to thread when debugging a problem.

Also very exciting is Siphon. Siphon is a set of servers and tools to support multi-set development. The basic idea is that you can create a set of packages. A package is just a collection of source files and documentation. Siphon provides a simple model for checking out and checking in a package. Checking out a package locks it so that no one else can check it out and modify the contents. This probably doesn't sound that exciting. The exciting thing that Siphon does is to automatically propagate modified files to other sites when the package is checked back in. It does this in such way that packages are never seen in a “half-way” state; that is where part of the sources have been copied but not yet all of them. Further, it does this in the face of failure. One of the really interesting parts of multi-site development is making sure that everyone has the most recent copy of the sources. This is especially hard when communication links can go down. Siphon takes care of all of these problems for you. A system like Siphon can save you considerable amounts of work if you are involved in multi-site development. By the way, Siphon is not restricted to Modula-3 source files. It can manage any type of source or documentation file.

The Libraries

A good, simple object-oriented language makes a nice starting point, but that in itself probably doesn't provide sufficient motivation for considering a new language. Real productivity comes about when there are good reusable libraries. This is one of the real strengths of SRC Modula-3 system. It provides a large set of “industrial strength” libraries. Most of these libraries are the result of a number of years of use and refinement. They are as well- or better-documented than most commercially available libraries.

Libm3

Libm3 is the workhorse library for Modula-3. It is the Modula-3 equivalent of libc (the standard C library), but it is considerably richer.

Libm3 defines a set of abstract types for I/O; these are called “readers” and “writers”. Readers and writers present an abstract interface for writing to “streams”. Streams represent buffered input and output. Stdin, stdout, and stderr represent represent streams that are familiar to most programmers. The streams package was designed to make it easy to add new kinds of streams.

In addition to the standard I/O streams, one can open file streams and text streams (that is, streams over character strings). There is also a set of

abstractions for unbuffered I/O. In addition to the File type, there are Terminal and Pipe. The Fmt interface provides a type-safe version of C's printf. A big source of errors in C programs is passing one kind of data into a printf, but trying to format it as a different kind of data. The Fmt interface was designed to have the flexibility of printf, but without introducing its problems.

Libm3 also defines a simple set of “container” types as generic interfaces. The basic container types include tables, lists, and sequences. A table is an associatively indexed array. The list type is the familiar “lisp” style list. A sequence is an integer (CARDINAL, actually) addressed array which can grow in size.

Finally, Libm3 provides a simple persistence mechanism called Pickles. Writing code to convert complex data structures to and from some disk format is tedious and error prone. Many programmers don't do it unless they absolutely have to. With the Pickle package, you no longer need to write this kind of code. Since the runtime knows the layout of every object in memory, it can use this information to walk a set of structures and read them from or write them to a stream. The programmer does not have to write object-specific code for writing an object to a stream, although he or she can if a better representation is known. For example, the programmer of a hash table may choose to write out individual entries if the table is below a certain size.

Trestle and VBTKit

Most user interfaces (UI) are a spaghetti of event handlers, timers, and signals. This is because they need to deal with user input coming in at arbitrary times, they need to deal with refreshing the screen, and they have to make sure that long running operations don't cause the application's windows to freeze up. All of these constraints make developing user interfaces in traditional languages and libraries very difficult.

The SRC Modula-3 implementation provides a UI library known as Trestle. The notable thing about Trestle is that it is highly concurrent. It was written to make extensive use of threads and to be used in a multithreaded environment.

This simplifies the development of user interfaces considerably, since you don't have deal with the event loop any more. An event loop is essentially “poor man's multithreading”. Since the language and libraries support first-class threads, these can be used instead. If the action associated with a button may take a long time, the action can merely fork off a thread to handle the bulk of the action. This thread can make arbitrary calls into Trestle to update the screen with new results. Trestle protects itself through judicious use of locks.

Trestle provides two sorts of objects: graphics objects such as paths and regions, and a base set of user-interface objects. These user-interface objects are known as “VBT”s. These play the same role in Trestle as the X intrinsics play in the world of the X toolkit. They define how screen-space is allocated among different “widgets”. Trestle provides a simple set of buttons and menus in its set of base UI items.

VBTKit is a higher-level toolkit built on top of Trestle. VBTKit provides a much wider array of UI object kinds. It also provides a Motif-like 3-D “look and feel” (on color displays). The same interface can be used on monochrome displays without change, but without the appropriate visual appearance. VBTKit provides the usual complement of scrollbars, buttons, menu items, numeric I/O objects, and the like.

FormsVBT and FormsEdit

FormsVBT is a User Interface Management System (UIMS) structured as a library and is built on top of VBTKit. FormsVBT provides a simple language for describing the layout of a user interface and an event interpreter for that language. The layout language follows a “boxes and glue” model. Boxes hold some set of UI objects. A VBox arranges those objects in a vertical display, while an HBox arranges them in a horizontal display. Glue is used to force a certain amount of space between items in a visual display. As you might expect, boxes can be nested arbitrarily.

The FormsVBT library allows you to specify callbacks to handle input. The FormsVBT specification that you write specifies the “syntax” of your user interface; the event handlers that you write provide the “semantics”.

FormsEdit is a simple UI creation tool built on top of FormsVBT. It reads and displays graphically, and in source text form, a UI specification written in the FormsVBT language. It also allows for interactive modification of the source.

m3tk

There are times when you just need to develop some language-specific tools for a project. The problem is that very few language implementations give you any support in doing this. Many times, all you have is a public domain YACC grammar that you have to modify and then build from there. This is where m3 tk comes in. It provides a complete toolkit for parsing M3 source files; and generating and manipulating abstract syntax tree representations of M3 sources. Thus a M3 specific tool can be built with relative ease. In fact, the Network Objects (see below) stub generator was built using it.

Network Oblects

Distributed systems are quickly becoming commonplace in the '90s. Most languages provide little or no support for distributed programming. Most distributed applications are still built directly on top of sockets or use libraries that provide a simple stream or RPC interface. These libraries are poorly integrated into the language and introduce a severe impediment between the language and the distributed system.

Network Objects is a facility in the SRC implementation of Modula-3 that allows Modula-3 objects to be exported across address spaces and machines. With Network Objects, a program can't tell if the object it is operating on is one that it created in its own address space or was one that was created and exists in another address space. This provides a very powerful mechanism for developing distributed applications.

To turn an object type into a network object, that object must inherit (either directly or indirectly) from the type NetObj.T. The object cannot contain any data fields. The interface containing the declaration is then run through a tool called a stub compiler. This generates all the coding necessary to handle network interactions. That's all that is required to allow an object to be passed around the network. Pretty simple. Below is an example of a network object. It defines an interface called “File” that defines the operations on a file, and an implementation of that interface.

```
INTERFACE File;
TYPE T = NetObj.T OBJECT
METHODS
getChar(): CHARACTER;
putChar(c: CHARACTER);
END;
END File;
INTERFACE FileServer;
IMPORT File;
TYPE T = NetObj.T OBJECT
METHODS
create(name: Text): File.T
open(name: Text): File.T
END;
END FileServer;
```

The above code defines two types: **File.T**, which is an object with two methods to get and put a single character; and **FileServer.T**, an object which manages file objects. A server someplace defines a concrete implementation of these abstract types.

```
MODULE FileServerImpl;
IMPORT File, FileServer;
TYPE FileImpl = File.T
...state for a file...
OVERRIDES
getChar := GetChar;
putChar := PutChar;
END;
TYPE FileServerImpl = FileServer.T OBJECT
```

```
  ...state for a file server...
  OVERRIDES
  create := Create;
  open := 0pen;
  END;
  VAR
  fileServer := NEW(FileServerImpl);
  BEGIN
  NetObj.Export("FileServer",
  END;
  MODULE FileServerClient;
  IMPORT File, FileServer;
  VAR
  fileServer := NetObj.Import("FileServer file");
  file := fileServer.Create("someFile");
  BEGIN
  file.putChar('a');
  END FileServerClient;
```

In the above code, **FileServerImpl** creates an instance of a file server and puts it into a name server. (The **NetObj.Export** call does this.) The module **FileServerClient** (which would be running in a different address space or machine) imports the file server implementation. This gives a valid Modula-3 object back to the client. From that point on, the client invokes methods on it as if it were local. It then creates a File object which it begins adding characters to.

If you have done any development with SunRPC or DCE, you will immediately appreciate how much simpler this is than programming on top of either of these systems. Network Objects is similar in scope to these systems, but is tightly integrated into the programming model instead of being a poorly integrated adjunct.

Two interesting systems have been built on top of network objects. The first is Obliq which is an object-oriented, distributed scripting language. Obliq can call into existing Modula-3 packages. You can also create Obliq objects and hand them to other programs running on other machines. Obliq is similar in scope to Telescript or TCL-DP (TCL with Distributed Programming extensions). The other system is Visual Obliq, which can be thought of as 'distributed Visual Basic for Modula-3'. It includes an interactive, graphical application builder. Callbacks are handled by Obliq scripts. This makes it a very powerful tools for prototyping and building distributed applications. It can also be used as the basis of interesting collaborative applications.

Some Personal Experiences with Modula-3

Our group has been using Modula-3 for about six months now, although I have been involved with it since 1989 or so. Our group consists of experienced C/C++ programmers. Two of have been involved with C++ since version 1.2 and two of us worked on the implementation of a C/C++ programming environment.

Our experience with Modula-3 has been completely positive. The group members feel that the language, libraries, and supporting tools have made us

far more productive than we were when using C++. The libraries are of higher quality and have better documentation than many commercially available libraries. To accomplish a given task, we write considerably less code than we used to and we believe the code is of higher quality. We attribute this to two things. The first is that the language is clean and simple; far less mental effort is required to understand how to accomplish something. The second contributing factor is much heavier use of libraries. Instead of writing some piece of functionality, we first see if the standard libraries provide it or something close to it. Most of the time we find something close enough that we can take it as a starting point.

On a more personal level, I have rarely seen a language, tools, and set of libraries that so neatly combined simplicity, elegance, and power.

Conclusions

Modula-3 and the implementation from SRC provides an excellent basis for developing Linux applications. It is a system designed to meet the programming challenges of the '90s. The language is clean, simple, and powerful. The provided libraries are almost unequalled. The support for distributed programming is among the best available.

One way to think of Modula-3 and the SRC implementation is bringing a “NeXTStep-like” environment to Linux. They both start with a simple object-oriented language (though M3 is both safer and more powerful) and build useful and sophisticated libraries on top of it. Of course, Modula-3 has the advantage of being freely available and running on Linux!

Geoff Wyant (geoff.wyant@east.sun.com) is a researcher in distributed systems with SunMicrosystems Laboratories. In his past, he has built programming environments for C++, worked on distributed file systems and RPC systems, and hacked operating systems kernels.

Archive Index Issue Table of Contents

Advanced search

Linux Command Line Parameters

Jeff Tranter

Issue #8, December 1994

In this article Jeff looks at a useful and perhaps little known feature of Linux—the ability to pass command line parameters to the kernel during system startup.

Consider the following situations:

Scenario 1: You are installing Linux from CDROM, but the kernel isn't using the correct I/O address for your CD-ROM drive. You can correct this by recompiling the kernel, but to build the kernel you first need to install Linux from CD-ROM...

Scenario 2: You made a change to the system startup script rc.local and now your system hangs while booting. How can you fix the error without reinstalling Linux?

Scenario 3: You'd like to experiment with the various VGA console video modes available without having to recompile the kernel each time.

Scenario 4: You've just written a large application that runs well on your system. How well would it run on a friend's machine that has only 4MB of RAM and no floating point coprocessor?

One solution to each of these problems is provided by Linux and its ability to pass command line parameters to the kernel. Unfortunately, these options are not well documented, (some of the HOWTOs mention certain options in passing, e.g., the SCSI HOWTO mentions some SCSI related options) and a number of them have been added relatively recently. We'll explore them in this article.

Booting Linux

In order to understand how command line parameters fit into the scheme of things, let's briefly look at what happens when Linux boots.

For those who aren't afraid to look at kernel source code, I'll mention some of the relevant files. The filenames given are relative to wherever you have installed the kernel source, usually **/usr/src/linux**. Therefore, a reference to the file **boot/bootsect.S** should be found in **/usr/src/linux/boot/bootsect.S**. This information is valid for Linux kernel version 1.2.

Starting from power on, the PC ROM BIOS routines load boot code from floppy or hard disk. If booting from hard disk, this is usually the boot loader installed by LILO. If booting fJ5m floppy, it is the code in the file **boot/bootsect.S**. This in turn loads the code found in **boot/setup.S** and runs it. This module reads some information from the BIOS (the VGA mode, amount of memory, etc.) and makes note of it for later use. It will be needed later as the BIOS routines will not (normally) be used once the kernel starts up.

The setup code next switches to protected (32-bit) mode, then loads and runs the code found in **boot/head.S**. (Actually, for compressed kernels, which is always the case in recent kernels, the kernel proper is first uncompressed using the code found in **zBoot/head.S**). This sets up more of the 32-bit environment, gets the command-line parameters (usually from LILO), and passes them to the routine **start_kernel**.

Up to now everything was written in assembly language. At this point we now switch to the function **start_kernel**, written in C, found in the file **init/main.c**. This is the code that does most of the option parsing, saving information on a number of kernel-specific parameters in global variables so that they can be used by the kernel when needed.

Any other parameters given as “name=value” pairs are passed as arguments and environment variables to the next process.

This first kernel process now sets some things up for multitasking, and makes the first call to the fork system call, creating a new process; we are now multitasking. The original (parent) process becomes the “idle process” which is executed whenever there are no processes ready to run. The child process (which has process id 1) calls the program **init**. (It actually looks in a number of places, including **/etc/init**, **/bin/init**, **/sbin/init**, **/etc/rc**, and finally **/bin/sh**.) The init program then starts up all of the initial system processes such as getty and other daemons, and shortly we have a login prompt on the console.

Setting Parameters At Kernel Compile Time

There are a number of important options that can be set when compiling the Linux kernel. These include the root device, swap device, and VGA video mode. The toplevel Makefile allows setting most of these.

The problem with this method is that recompiling the kernel is somewhat time-consuming (at least on my machine; do you have a 100MHz Pentium?). You must also modify the standard Makefile, and remember to continue to do so when upgrading to newer kernels.

Setting Parameters Using rdev

The **rdev** command was written long ago to make it easier to set some of these important kernel options without a recompile. The program directly patches the appropriate variables in a kernel image. These are at fixed addresses (defined in **boot/bootsect.S**).

While using **rdev** is fast, it is still somewhat inconvenient in that you have to remember to run it after building each kernel. It is also limited in the options that can be changed. We can do better.

Setting Parameters Using LILO

If you are using the LILO (LInux LOader) program to boot Linux (usually from hard disk), then you can pass command-line options to the kernel at boot time. Typically these are set in the configuration file **/etc/lilo.conf**.

This is the most flexible method. It allows you, for example, to boot different kernels or boot the same kernel with different options.

Most options are passed by LILO on to the kernel; one useful option is parsed and handled by LILO itself. ,The console video mode for VGA displays can be set using a command-line option of the form:

```
vga=mode
```

where *mode* can be one of:

- “normal” for the default 80-column by 24-line display,
- “extended” or “ext” for 80 columns by 50 lines,
- “ask” to prompt the user at boot up time for the mode to use, or
- a decimal number to select various other modes, dependent on the type of VGA card (for example, on my Trident VGA card, mode 6 is 132x30).

Kernel-Specific Parameters

Let's now look at the specific options supported by the Linux kernel. These affect the behavior of the kernel itself and are not passed on to the **init** program.

Some of these options accept a numeric value, parsed by a simplified version of the strtoul library function. Values can be given in decimal (e.g., 1234), octal (e.g., 01234) or hexadecimal (e.g., 0x1234), and should be separated by spaces. Let's now examine each of the options.

```
root=device
```

e.g.,**root=/dev/hda**

This option sets the root device; the device used as the root ("/") filesystem; when booting. It accepts a value from a hard-coded list of common devices: **/dev/hda..b** (IDE hard disks), **/dev/sda..e** (SCSI disks), **/dev/fd** (floppy), and **/dev/xda..b** (XT hard disks). These are mapped into the corresponding major and minor device numbers.

This option indicates that the root filesystem should be mounted readonly. Typically this is done in order to run **fsck** on bootup.

```
rw
```

This option is the converse of the previous one, indicating that the root filesystem should be mounted for both read and write, the normal case once a Linux system has been booted.

```
debug
```

This option sets the kernel logging level to 10, rather than the default value of 7. It sets the global variable "console_loglevel". Currently this make no visible difference; apparently there is no kemel code that displays messages at levels higher than 7.

```
no-hlt
```

This sets the global variable "hlt_works_ok" to 0. When Linux is idle, it runs the previously mentioned idle process in a loop (found in **kernel/sys.c**). Having the idle process periodically execute a **hlt** (halt) instruction reduces power consumption on some machines, most notably laptops. However, a few users have reported problems with the kilt instruction on certain machines, so it can be disabled with this option.

Incidently, I routinely use this option on my desktop system; I find that it significantly reduces the level of bus noise picked up on my sound card.

```
no387
```

This option sets the global variable “hard_math” to 0. It forces the kernel to use co-processor emulation, even if one is installed. This can be useful if you suspect hardware problems with your co-processor or if you want to measure performance without a math chip.

```
mem=bytes
```

e.g., **mem=4000000**

This option specifies to the kernel the highest memory address to use (specified in bytes). Normally Linux uses all of the available memory. This feature can be useful for simulating machines with less memory or debugging cache problems on machines with lots of memory. As an experiment, try booting your machine with less memory, say 2MB, to highlight the difference memory makes. As another experiment, see what happens if you lie and tell Linux you have more memory than is installed...

```
reserve=port, size. . .
```

e.g., **reserve=Ox320,0x20**

This option reserves I/O ports; it marks them as used so they will not be probed by device drivers that do autoprobing. This may be needed on certain systems that have unusual hardware or device conflicts.

```
ramdisk=size
```

e.g., **ramdisk=2000000**

This option sets the size of the RAM disk, in bytes.

Device-Driver-Specific Parameters

The next group of options are specific to individual kernel device drivers. I won't describe each of them in detail, because some of them are rather specialized and are documented elsewhere.

```
ether=a,b,c,d,e
```

This option is for setting up Ethernet interfaces. It allows setting parameters such as the interrupt request number and base address. The meaning of the

parameters varies somewhat depending on the type of interface card. The Ethernet HOWTO document describes these in detail.

```
max_scsi_luns=number
```

This option sets the highest Logical Unit Number to be used for SCSI devices. Valid values are 1 through S. This may be needed if autoprobing of the SCSI bus causes problems on your system.

```
hd=cylinders, heads, sectors
```

This option sets the hard disk geometry for SCSI or IDE disks. Normally Linux obtains these from the BIOS; the command line option can be used to override those if they are not correct.

```
st=buffer size,write_threshold,tape_buffers
```

This option is for setting SCSI tape driver parameters. The file **drivers/scsi/README.st** describes these in detail.

```
bmouse=irq
```

This option sets the interrupt request line to be used for the bus mouse driver.

```
stOx=parameters
tmc8xx=parameters
stOx=parameters
tl28=parameters
pasl6=parameters
ncr5380=parameters
ncr5380=parameters
ahal52x=parameters
```

These are all options for setting up the various SCSI host adaptors supported by the Linux kernel. See the SCSI HOWTO for more information.

```
xd=type, irq,i/o base_address,dma_channel
```

This option sets the XT hard disk driver parameters. See the comments in the file **drivers/block/xd.c** for more information.

```
mcd=port, irq, workaround_bug
```

This sets up the Mitsumi CD-ROM interface. The first two parameters are the base I/O address for the controller and the interrupt request. The third option sets a delay value used to work around problems with some Mitsumi drives.

```
sound=parml, parm2,...
```

These options set up the sound driver. The parameter encoding is explained in the file **drivers/sound/Readme.linux**.

```
sbpcd=address, type
```

e.g., **stpod=Ox230,SoundBlaster**

This command sets the parameters for the SoundBlaster/Panasonic CD-ROM driver. See the file drivers/block/README.stpod for details.

Parameters to init

Any other options are passed on to init in its argv array and as environment variables. For example, LILO passes the argument “auto” if the system was booted without a manually entered command line. The command “single” will instruct init to boot up Linux in single-user mode.

Here is how to see what options were passed to **init**:

```
% ps -awww | grep init
        1 con S 0:03 init auto
```

The proc filesystem also lets you look at the environment passed to **init**, which always has process ID number 1:

```
% cat /proc/l/environ | tr t'\O'' "\n"
        HOME=/
        TERM=conl32x30
```

Final Comments

A colleague suggested a kernel option that is missing: “help”. While this is not implemented yet, hopefully this article has convinced you that there are many other useful options worth exploring.

(Jeff.Tranter@Software.Mitel.com) is a software designer for a telecommunications company in Ottawa, Canada. He has been using Linux for more than two years and is the author of the Linux Sound and CD-ROMs, and several Linux utilities.

Archive Index Issue Table of Contents

Advanced search

Linus Torvalds in Sydney

Jamie Honan

Issue #8, December 1994

Jamie Honan gives us a brief report on Linus' international visit, and the Australians' quest to impress Linus.

The Australian Unix Users' Group asked Linus to come and talk at their annual conference in Melbourne. While here in Australia, he also traveled to Canberra, Sydney, and Perth and talked at seminars organised by local Linux user groups.

I helped organise events in Sydney. We scheduled only two official events, which left Linus plenty of time to take in the local sights.

The first official function was a talk, which went quite well. Most of the 60 people who attended run Linux, and Linus was surprised at the technical depth of the questions. Although he had planned to talk for about an hour we went on for well over two. At the end of the talk, Peter Chubb presented Linus with one of his SLUG (Sydney Linux Users Group) tee-shirts. Quite leterally, this tee-shirt has a picture of a slug on it. As I said at the time, how could we give him anything that truly represented the gratitude we felt?

The day after, we went for a picnic on Shark Island in the harbour. It was also a success—the day was stunning and the harbour was awash with sailboats flying their spinakers. Lots of photos were taken; perhaps when they're developed some could be put up for ftp, so those who didn't come can live it vicariously.

On a personal level, Linus was an absolute delight to look after.

One highlight: walking past the local bowling club and explaining how the Australian variety of outdoor bowls is played.

We had been trying to impress Linus with Sydney's highlights. We all thought the opera house would do it. Linus would say, “Well, it's very nice, but I'm not really impressed.” on it went, the sunny harbour—“Yes, it's very nice;” how

about holding the Koala bear—“Yes, just nice.” After a while, it got to be a little challenge, and we ribbed hime about his Northern European “never being too impressed” attitude. But we finally gave up trying to impress him, and took him to the handiest dive for a quick cheap meal: a local RSL, or veterans association, club.

Just off the retaurant was the room which holds all the poker machines (slot mahinces, or “pokies” in Aussie slang), and we had a quick look inside: a ghaslty and garish collection of brighly lit and noist machines. I was slightly embarrassed, after all the good impressions we'd been trying to make, but Linux just smiled and said, “Now I really am impressed.”

I know Linus had a good time. I just hope circumstances permit him to come back before too long. There's an old fashioned saying, and it really applies to Linus: “You couldn't meet a nicer bloke”.

Archive Index Issue Table of Contents

Advanced search

The Term Protocol

Liem Bahneman

Issue #8, December 1994

Need to run multiple tasks but have only a standard dial-up connection to the host? Term may be the answer you have been looking for.

Term, originally developed by Michael O'Reilly (michael@iinet.com.au), is a program that allows multiple, concurrent connections over a serial line. Term allows almost all “standard” TCP/IP applications to be used on a Unix system that is connected by a serial connection to a networked Unix system. Unlike other common serial protocols, such as SLIP and PPP, term does not require non-user administrative maintenance, and requires no modifications to the host kernel. This means that virtually any user with a login shell on a dialup system can utilize network utilities that were once limited to SLIP/PPP users.

Unlike SLIP or PPP, your machine does not have its own IP address. All incoming traffic must be addressed to your remote host, and it will be directed to your local computer by term.

Term essentially works by redirecting packets on your remote host directly to your local Unix system. This allows any incoming network packets to reach your computer by proxy, via your remote dial-up computer. The same basic idea works for outgoing packets as well: local sockets on your computer are redirected to your remote host, and sent on their way to their actual network destination.

The entire term package is a basic suite of utilities and libraries that allow you to establish these network connections. These utilities are:

- **term**: This is the actual daemon that is run on both the remote and local computers. This establishes the bridge that is needed to link your computer to the remote host and the rest of the network.
- **tredir**: This is the most commonly used utility that comes with term. It allows the user to manually redirect an outgoing or incoming port for use

with non term applications, for example redirecting the SMTP (e-mail) port so that the user may send or receive e-mail.

- **tmon**: This utility monitors and displays the incoming and outgoing traffic over your serial line. Two bar graphs are displayed showing the levels of traffic, updated each second. This allows you to monitor just how much bandwidth you are using at any time while using term.
- **trsh**: This utility allows you to quickly access your remote login shell, much like rsh or rlogin would allow you to. This allows you to perform routine network tasks from your account if needed.
- **tupload**: Much like **sz**, this utility is used to transfer files to or from your remote account, depending on which “end” of the term-link it was executed from.
- **txconn**: When you need to display an X application remotely, or have one displayed on your local screen, txconn establishes the needed redirection to make this possible. (The same effect can be created with tredir, as will be explained later.)
- **Other applications**: Recently, a flurry of activity has resulted in a few more term clients such as **tudpredir**, a udp port redirector; **tdate**, which sets your computer's time by the Network Time Protocol; and “download, which reciprocates what **tupload** does.

Configuring Term

Before you can actually run term, you should run a utility called **linecheck** on the remote and local computers.

Linecheck is used to check the ”transparency“ of the link, by seeing which 8-bit characters are transmitted across the link. The results of **linecheck** are used to configure term to operate correctly and optimally.

To run linecheck:

- Using a communications program, log into your account on the remote system and run:

```
linecheck linecheck. log
```

- Suspend your comm program (^Z under kermit), otherwise it will steal characters from linecheck.
- On the local system, run:

```
linecheck linecheck.log > /dev/modem < /dev/modem
```

After linecheck has completed its operation, examine the two **linecheck.log** files. At the bottom of these files will be an indication of which characters you must escape in your .termrc configuration file. The messages in **linecheck.log** give the characters (if any) that need to be ignored on one end and escaped on

the opposite end of the link. For example, if my local results indicated that I should escape 3 4 and 121, my resulting **.termrc** files would have something like this in them:

```
Local:
escape 34
escape 121
```

and my remote **.termrc**:

```
ignore 34
ignore 121
```

because I have to *ignore* escaped characters on the other end.

Running Term

Term is very flexible with many configuration options on the command line as well as in the **.termrc** file. Running term is much like running **linecheck**:

- Using a communications program, dial up your remote account and log in
- Start term from that account. A sample command line might be:

```
term -l $HOME/tlog -s 38400 -c off -w
10      -t 150 -r
```

This command line indicates:

- Set the log file to tlog in your home directory
- Set the line speed to 38400 bps
- Turn off term's compression (presumably because your modem does better compression)
- Use a window setting of 10 (explained in the term documentation)
- Use a timeout of 150 (explained in the term documentation)
- Set this as the ”remote“ side
- Shell back to your local computer, either by suspending your terminal program, or using its built-in shell features. For Kermit, use ctrl-Z, for xcomm use ctrl-a-x. (Check your specific terminal program's own documentation.)
- Initiate term on your local computer:

```
term -c off -l $HOME/tlog -s 38400 -w 10 -t 150
< /dev/modem > /dev/modem &
```

The only difference in this case is the redirection to the modem device and lack of the -r option.

It should be noted that all of the command-line arguments can be placed in the . termrc file so you need only type term by itself to initiate it:

```
.termrc:
compress off
speed 38400
window 10
timeout 150
```

Note that you will still need to put the redirection on the command line.

Using the Standard Term Clients

The standard term clients **trsh**, **tredir**, **tmon**, "upload, and txconn are relatively easy to use. The most commonly used utility is **trsh**.

Trsh

trsh is used to access your remote account as if you were using rlogin to access it. **trsh** can also act like **rsh** and execute commands on your remote host:

```
% trsh
Remote: term 2.0.4
tty /dev/ttyp4. exec /usr/local/bin/tcsh
foober : /home/ j oeuser%
% trsh -s uptime 1:15am up 20 days, 17:30,       3 users,        load
average:
1.00, 1.00, 1.00
```

Tredir

The most utilitarian of the term clients, this command allows you to manually redirect TCP/IP ports for use with term. For example, to allow incoming telnet sessions to your home computer, you need to redirect a port on the remote host to your own telnet port, which is port **23**. The common command format of **tredir** is:

```
tredir [thiscomputer: ]port [thatcomputer: ]port
```

By default, the first port is the port on the machine you are running the command on, the second port is the port value on the other computer you are redirecting to.

In this example, I want to redirect port **4000** to my own port **23**:

```
remotehost% tredir 4000 23
Redirecting 4000 to 23
remotehost% telnet localhost 4000
Trying . . . Connected to localhost.
Escape character is '^] '
Linux 1.1.35 (linuxbox) (ttyp3)
linuxbox login:
```

Another example of use of tredir is to configure your system to allow reading news via your network's NNTP news server. This requires a tredir on the local side of term, instead of the remote:

```
linuxbox% tredir 119 news. server.com: 119
Redirecting 119 to news.server.com:ll9
linuxhox% export NNTPSERVER=localhost
linuxbox96 trn
```

[normal trn session follows]

Notice that in this example the NNTPSERVER variable is set to localhost. This is because the local 119 port has been redirected to the real network NNTP server. So any connections to the localhost NNTP port is redirected to the real one on the remote computer. A direct connection to the actual NNTP server (setting NNTPSERVER to news.server. com) would not be possible on a term link, unlike SLIP/PPP which would allow this. tredir makes possible the use of many applications that use standard TCP sockets, such as sendmail, IRC, MUD's, MUCK's, and many other similar multi-user games.

Tupload

This is the term equivalent of sz or other file upload/download protocols. It allows the transfer of files from the local machine to the remote, or vice-versa, depending on which end the command is initiated. Commonly, the command line would look like:

```
linuxtox% tupload foot tar. gz
```

Which would send a copy of the file foo.tar.gz to the remote host. Some useful flags are illustrated below:

```
linuxbox% tupload -vv -p -16 foo.tar.gz
Changing priority to -16
sending foo.tar.gz
30651 of 259727 (11%),
current CPS 3083. ETA:
76.8 TT: 84.2
```

The **-vv** flag means give verbose messages on the status of the upload, while **-p** means change the term priority of the upload. This prioritizing allows you to nice a term process so it doesn't hog bandwidth from the other term applications you may be running. This is useful for large background transfers.

Txconn

txconn is designed to ease the redirection of X applications from one host to another. If a user on computer2 wants to display an X application on your screen, you use txconn. Like using tredir, any incoming connections must use your remost host's name or IP address to connect to you.

Because X is outwardly different than normal TCP/IP clients, it needs special handling for redirection. By itself, txconn uses no command line arguments:

```
remotehost% txconn
Xconn bound to screen 9
:9
```

This means that your home X display can be accessed from the network as **remotehost:9**, meaning root window 9 on the remotehost. If a user on computer2 wants to send his xclock to your local display, he would type:

```
computer2% setenv DISPLAY
remotehost:9 computer2% xclock &
```

After a few moments, the xclock he executed will appear on your display.

If you want to display an X application running on your computer to computer2's display, you must use tredir. This is a bit confusing.

```
linuxbox% tredir 6004 computer2:6000
Redirecting 6004 to computer2:6000
linuxhox% export DISPLAY=localhost:4
linuxbox% xclock &
```

This may look a bit odd, but what you are doing is redirecting your display :4 (unused by you) to computer2's default display of :0. Ports 6000-6100 represent displays :0 to :100. By redirecting your own display :4 to his :0, any X application on your local machine which uses display :4 will appear on computer2's screen. It's a bit convoluted, but it works effectively.

Other Term Clients

Because of the nature of term, applications that work with standard TCP/IP will not work without the use of tredir, and even then, they may not always work. For example, it would be impossible to use tredir for an application like NCSA Mosaic because it makes so many different connections to different hosts and services. Other applications, though they use a single network connection, don't work because they use a secondary data port, such as **ftp** or IRC's DCC protocol. Applications such as this require modification of the actual source code to utilize the term socket. Most popular applications such as Mosaic, lynx, ftp, ncftp and irc have already been modified for use with the term protocol.

If you compile these applications yourself, they must be linked with the term library, **libtermnet.a** (This library has replaced the old **client.a** library.) This library contains the needed instructions and symbols for using the term socket.

New developments in term have made it extremely easy for users to modify existing TCP/IP applications for use with term, without the massive source code modifications that were once required. By using drop-in replacements for common socket/networking functions such as **connect()**, **gethostbyname()**, and **send()**,you need only modify the Makenle of an application to make it term-

compliant. This drop-in replacement is the **libtermnet.a** library, and a header file which translates standard networking calls into term-compliant calls. One interesting note is that a termnet-linked binary also works with normal TCP/IP, so if you ever change to SLIP or PPP in the future, your term-compliant binaries will still work!

Only two elements are needed in the Makefile to make a term-compliant binary. In the location where the **INCLUDES** are defined, you add:

```
-include /usr/src/term200/termnet.h
```

adapting the path for your term source path, of course.

And in the **LIBS** or **LDFLAGS** section, you add:

```
-L/usr/local/lib -ltermnet
```

If you have **libtermnet.a** or the **libtermnet.so.2.0.*** shared library installed in a common library path, **-L/directory/path** isn't needed.

Hopefully, after adding these definitions, you end up with,` fully term-compliant binary. There are still a few shortfalls, of course. Many applications use non-standard socket calls, and termnet cannot fully control things like that. Also, the newly integrated (as I write) udp support in term is still very rough.

Applications such as Chimera, lynx, xarchie, rsh/rlogin and fsp are a few examples of successful tennnett~ng, and more are sure to follow.

With the implementation of termnet, the days of manual source-level hacking on most applications is over, and more and more applications that were once too hard to hand-patch term support into will be available to term users.

The Future of Term

After **term116**, the term development was passed on to Bill Riemers (bcr@physics.purdue.edu). Major additions to term have been introduced, such as udp support, which is getting better and better, and shared libtermnet, which allows easy upgrading of term versions without recompiling term-compliant binaries every single time. The **udp** support has enabled such applications as ytalk, xarchie and fsp to Work through term.

There are still a lot of things to complete and improve with term, but it's a very successful and very useful tool for people who don't have the resources to run full-blown SLIP or PPP. It is very good, and can only get better.

Liem Bahneman is a student Unix consultant at the University of Washington and is the administrator of the Linux Organization WWW home page. Liem has heen using Linux and term for almost two years and in his free time develops X11 applications in C and tcl/tk.

Archive Index Issue Table of Contents

Advanced search

Linux System Administration

Mark F. Komarinski

Issue #8, December 1994

This month's article deals with your system clock and how you can set up your clock in Linux and keep it updated.

It may not appear difficult to keep your clock set correctly. DOS had the ability to set its time directly from the BIOS (Basic In/Out System) and you could set the BIOS time directly from DOS. Now you have to worry about time zones, machine time vs. software time, and finding out how to set the time when your system could be off by 30 seconds every day.

zoneinfo

It's easy to set up your system for the correct time and if you're networked, it's easy to keep your time updated. First, get a copy of the zoneinfo package. For releases like Slackware, it is already installed (and may be already set up in Slackware 1.2.0 and later). Look for the **/usr/lib/zoneinfo** directory. If you don't have that directory, get a copy of the package (see sidebar “Finding zoneinfo”) and install it.

For the best setup, you should set your BIOS (or hardware) clock for GMT (or Zulu, Universal time, or whatever you want to call it). The advantage of having your BIOS clock set to GMT is that Linux will automatically handle daylight savings time conversions for your particular area. You may not want to do this if you still use DOS, but I find that DOOM does not care what time I play. I also found that people in the office think you're really dedicated when you bring in files with time stamps of 2:30AM.

The first thing you should do is boot into Linux and go into the **/usr/lib/zoneinfo** directory. You should see a list of various time zones. Some are listed by country, some have directories with more listings under them (such as the US directory). Find the file that works with your country or time zone. For my system, this would be the US/Eastern file, as I am in the Eastern Time Zone for

the US. If your country is not listed, there is a list of files that relate to times + or - to GMT, such as GMT-6. Find out how far off you are from GMT, and use that file instead.

From here, I create a symbolic link from **/usr/lib/zoneinfo/localtime** to **/usr/lib/zoneinfo/US/Eastern**:

```
ln -sf /usr/lib/zoneinfo/US/Eastern /usr/lib/
zoneinfo/localtime
```

Another file to link is the posixrules file. This can be linked to localtime. If you have the TZ variable set, the **posixrules** link will be used to set the correct time zone.

```
ln -s /usr/lib/zoneinfo/
localtime /usr/lib/
zoneinfo/posixrules
```

clock

If your BIOS clock is set to GMT, you have to tell Linux this using the clock command. The best way to do it is edit your **/etc/rc.d/rc.local** file (or **/etc/rc.local** file if you don't have an **rc.d** directory) and add the following command anywhere in the file:

```
#Tell Linux the BIOS
is universal time!
clock -us
```

Or if you have your clock set to local time:

```
#Tell Linux that the BIOS is set for local time
already!
clock -s
```

The **-s** option indicates to set the clock, and the -u option indicates that the BIOS clock is set to 'universal time', or GMT.

From now on, when you use the **date** command to view the time, you will see the correct local time, along with the time zone you are in:

```
#date Thu Aug 10 22:15:35 EDT 1994
```

It's easy to set up your system for the correct time and if you're networked, it's easy to keep your time updated.

netdate

If you're networked to other machines with a better sense of time than yours, you can use the netdate command to periodically keep your machine time correct. As root on your machine, just enter the command:

```
netdate [ -v ] [ -l limit ] <hostl> ... <hostn>
```

where **hostl...hostn** is a list of hosts. It is usually best to list hosts that are physically close to your system, especially if you're using a dial-up PPP or SLIP.

The way neLdate works is to collect the hosts into groups based on how close the times are. Of the hosts with the times closest to the local time, the first alphabetical host of that group is used to set the time on the local host. The **-v** option will list the groups that get created, and the **-1 limit** option varied the amount of time that netdate will wait for time information from other hosts to come in. A limit of 0 will accept the time of the first host that responds and ignore all the others.

However, netGate does not update your BIOS clock, only the system clock. To update the BIOS clock, use the **clock -uw** command to 'write' universal time, or **clock -w** if you keep your BIOS set to local time. One idea you may want to try is creating a shell script which executes the netdate and clock commands automatically and keep it in your **/etc** directory. This way, if you notice the time is a bit off, you can execute the shell script to update the time, and update your BIOS clock simultaneously.

If you are interested in doing more with the time zones, check out the man pages for **date**, **clock**, and **netdate**. Also in the **/usr/lib/zoneinfo** directory is a **time.doc** text file which is good reading.

Archive Index Issue Table of Contents

Advanced search

Linux Organizations

Michael K. Johnson

Issue #8, December 1994

Have you ever wanted to rant about commercial applications for Linux? Wanted to find fellow Linux addicts? Wanted an understanding shoulder to cry on because you just typed “rm -rf /”? Linux organizations, at local and global levels, can enhance the sense of community between Linux users.

As the Usenet **comp.os.linux.*** groups get more and more crowded, it has taken more and more time to participate in the global Linux movement. For those without Usenet access, it is almost impossible. Linux users groups (LUGs) and organizations are starting to spring up all over the world, with many different goals. We asked three of them for statements explaining what they are about. One is a global group, one an amalgam of local Linux users groups, and one is a local Linux users group.

Linux International

Patrick D'Cruze pdcruze@li.org.au

Linux International (LI) is a not-for-profit, volunteer-run organization that has been formed to promote and encourage the growth of Linux. Essentially our goal is to encourage as many people, organizations, and communities as possible to start using Linux.

We are all Linux enthusiasts within Linux International. Each of us is grateful for the work that has been done by the countless Linux developers over the past three years. We ourselves would like to contribute something in return back into the Linux community that would benefit others. This is the common strand that ties all of us together within Linux International.

Linux International has begun working in a number of areas. The three most important areas are supporting Linux developers, supplementing Linux development, and lobbying commercial companies.

The Linux developers are one of Linux's biggest strengths. Developers are a committed, highly enthusiastic, professional group of people who have devoted an enormous amount of time and resources to Linux. It is a great credit to them that they have developed an extremely sophisticated and reliable operating system such as Linux without the huge resources and support of a large commercial company. Linux International would like to do what ever is possible to support their work and aid them in their continuing development efforts.

Linux International will soon be announcing a worldwide trust fund. (It will probably be in operation by the time you read this.) Many people have expressed interest in making some small monetary contribution to the Linux developers to thank them for their work and to be used to aid them in their future work. Linux International is in the process of establishing central donation points in nearly every country to make it easy for grateful groups and individuals to donate. All money collected will be distributed to the Linux development community.

Linux International has also been involved in some supplementary development projects. The work we undertake is designed to complement the existing work undertaken by the Linux developers, and will be aimed at filling in a few holes in an otherwise very impressive operating system. The work we undertake will be in a similar spirit to the existing Linux development efforts and it is hoped will be of use to the Linux community. Our work so far has concentrated on large projects such as national language support for Linux and the development of configuration and autoconfiguration tools.

The third major area we are involved in is lobbying third-party software and hardware manufacturers to support Linux. Many software developers and hardware manufacturers may be unaware of the opportunities available by targeting the Linux market. We are involved in an on-going campaign to alert them to the possibilities available and to encourage them to support the Linux user base.

These are just some of the areas in which Linux International has been involved. There are many other “public service” areas in which we would like to contribute; however, at the moment we are constrained by the number of people working in the organization.

Linux International is a volunteer-run organization. All of the work we are involved in is being undertaken by a number of volunteers. However the work never ends and we gladly welcome new volunteers who would like to help out.

Why would people be interested in contributing to LI? There are a number of reasons:

- Linux International has been formed to work for the greater good of the Linux community. Our aims and objectives are to aid the Linux developers and also to provide a number of other benefits to the community. Our emphasis is on helping the Linux community and doing all that we can towards satisfying their needs. We can and do provide those working with Linux International with job references testifying to the work that they have performed and the experience they have accrued. This has already benefited a number of students and other professionals.
- Many people are eager to contribute something back to the Linux community. However many people do not have the time or the skills to contribute by developing some software for Linux. There are many ways non-software developers can contribute within Linux International. We are always in need of more people to assist us in our efforts at lobbying software developers and hardware manufacturers. There are plenty of other ways by which people can contribute something back into the Linux community. If anyone is interested in doing so, we would certainly like to hear from them.

There are no fees associated with participation in Linux International; the only requirements are dedication, motivation, and enthusiasm. Each of us within Linux International is committed to using our talents to help and expand the Linux community. To others who share this philosophy and would also like to contribute something to the community: we are more than pleased to hear from you. Contact Linux International by sending e-mail to info@li . org. au.

If you have questions or suggestions for Linux International, please don't hesitate to contact us. We are here to serve the Linux community and we welcome any feedback or assistance that would allow us to better achieve this.

So in what way do we add to the Linux experience? There are many ways in which a company or organization can effect changes or accomplish things which individuals might find difficult to do. Two examples are a world-wide trust fund, and lobbying software developers. Both can be achieved by individuals, but they are arguably easier to accomplish by working within an organization. Linux International is committed, like many people, to helping the Linux community. We can and do use our organizational status to great effect when this allows us to achieve things that individuals may find hard to do. But perhaps more than this, we represent a group of committed Linux enthusiasts. We all know that Linux is the best operating system in the world. Our job now is to go out and “sell” it to the rest of the world and spread the “Linux gospel” to others. We firmly believe that there is nothing better than sharing a good thing

with someone else. And this is really what the Linux community is all about. Sharing.

International Linux AssociationCharles Liu 1645 S. Bascom Ave., #7Campbell, CA 95008Phone/Fax: 40tl-369-9818alte@rahul.net

The International Linux Association, or ILA, is an amalgam of LUGs. The ILA is an attempt to assist bringing Linux into the mainstream of computing and provide benefits and value to members. At this time we will start to:

- Hold monthly meetings at each chapter to allow members or members-to-be to have a place and time to get together face-to-face to exchange information. Also, the invited speaker will share his/her point view of Linux to the attendees.
- Provide training seminars/classes for experienced and inexperienced Linux users.
- Provide a package to allow student members to afford the Linux CDs and books. (We believe that no one should be without Linux for merely economic reasons.)
- We would like to organize a Linux speaker bureau to provide speakers available worldwide to talk about Linux.

Linux opens a lot of career opportunities for individuals, as well as business opportunities for companies. For example, the individual who is going to teach the course “Introduction to Linux” is currently a software consultant; ILA provides him the opportunity to teach and become visible and credible to the public, which creates some consultancy job opportunities.

Through initial and followup training courses, ILA will be able to start introducing and promoting Linux to members who participate in such processes would improve their professional career.

For corporate members, we hope to help them take advantage of Linux as a paragon of Open Systems.

I feel very strongly that if we do it right we will achieve what we want to do.

Michael K. Johnson is the editor of *Linux Journal*, but that doesn't keep him from hacking.

Archive Index Issue Table of Contents

Advanced search

Extracts from Linux Meta-FAQ Version 3.11

LJ Staff

Issue #8, December 1994

This is the Meta-FAQ for Linux.

September 15, 1994

This is the Meta-FAQ for Linux. It is mainly a list of valuable sources of information. Check out these sources if you want to learn more about Linux, or have problems and need help. Lars Wirzenius (wirzeniu@cc.helsinki.fi) wrote the first version of this document, and it is now maintained by Michael K. Johnson (johnsonm@sunsite .unc.edu). Mail him if you have any questions about this document.

NOTE: Filenames in this article are for the tsx-11 .mit.edu ftp site unless otherwise noted. (see below for names of more ftp sites). Files are usually located in similar places on other sites. The names are relative to the directory / pub/linux/ on tsx-ll.

SITE ADDRESS NOTE: If the beginning of a line starts with a dot, it is the continuation of the address from the previous line.

Support for PowerPC, Alpha/AXP, and MIPS is in the works, but don't hold your breath. Read comp.os.linux .announce instead.

See the FAQ for more exact hardware requirements. The Linux kernel is written by Linus Torvalds (torvalds@ kruuna.helsinki.fi) from Finland, and by other volunteers. Most of the programs running under Linux are generic Unix freeware, many of them from the GNU project.

The Linux FAQ

A collection of common problems and their solutions. Answers many questions faster than the net. Stored on many Linux ftp sites (docs/) and rtfm.mit.edu, the general archive site for all FAQs.

The Linux HOWTOs

These are somewhat like FAQ's, but instead of answering common questions, they explain how to do common tasks, like ordering a release of Linux, setting up print services under Linux, setting up a basic UUCP feed, etc. See sunsite.unc.edu, directory /pub/Linux/docs/HOWTO/ for all the HOWTO's.

Linux Newsgroups

There are several Usenet newsgroups for Linux. It is a good idea to follow at least comp.os.linux.announce if you use Linux. Comp.os.linux.announce is moderated by Matt Welsh and Lars Wirzenius. To make submissions to the newsgroup, send mail to linux-announce@tc.cornell .edu. You may direct questions about comp.os.linux.announce to Matt Welsh, mdw@sunsite.unc.edu

The newsgroup comp.os.linux.admin is an unmoderated newsgroup for discussion of administration of Linux systems.

The newsgroup comp.os.linux.development is an unmoderated newsgroup specifically for discussion of Linux kernel development. The only application development questions that should be discussed here are those that are intimately associated with the kernel.

The newsgroup comp.os.linux.help is an unmoderated newsgroup for any Linux questions that don't belong anywhere else.

The newsgroup comp.os.linux.misc is the replacement for comp.os.linux, and is meant for any discussion that doesn't belong elsewhere.

In general, do not crosspost between the Linux newsgroups. The only crossposting that is appropriate is an occasional posting between one unmoderated group and comp.os.linux.announce. The whole point of splitting comp.os.linux into many groups is to reduce traffic in each. Those that do not follow this rule will be flamed without mercy.

Other Newsgroups

Do not assume that all your questions are appropriate for a Linux newsgroup just because you are running Linux. Is your question really about shell programming under any Unix or Unix clone? Then ask in comp.unix .shell. Is it about GNU Emacs? Then try asking in gnu.emacs. Also, if you don't know another group to ask in, but think there might be, politely ask in your post if there is another group that would be more appropriate for your question. At least the groups comp.unix.{questions,shell,programming,bsd,admin}, and comp.windows.x.i386unix should be useful for a Linux user.

The World-Wide Web

Matt Welsh, mdw@sunsite.unc.edu, maintains the home WWW page for the Linux project. The URL is sunsite.unc.edu/mdw/linux.html

Getting Linux—Linux FTP Sites

A more complete list of Linux FTP sites is in the Linux INFO-SHEET (docs/INFO-SHEET). The most important sites are listed here; please see the INFO-SHEET for a site nearer to you (there are many mirrors).

Archive Index Issue Table of Contents

Advanced search

DOOM

Michael K. Johnson

Issue #8, December 1994

This popular DOS game is now available under Linux.

I first heard of DOOM on Usenet, when people would say “I can't wait to get rid of DOS, but I still need DOS to play DOOM.” Wait no longer. I first played DOOM a few days ago while running X on my Linux box.

I was rather skeptical. I play very few computer games, and not very often. When I do, they are usually games or clones of games like Minesweeper, Tetris, Mahjongg, Golddig, and those rare card games whose rules I am able to comprehend. I have never particularly enjoyed adventure games of any sort, until I played DOOM. Now my wife is worried I'm becoming addicted.

David Taylor (of Id, the company that wrote DOOM) recently completed a port of DOOM to X under Linux, and asked me to review it. I unpacked it (approximately 5MB worth), read the **README.linux** file (this is important if you have never played the game, because it explains how to move, shoot, and open doors, among other things), and played. And played. And played.

The first thing I noticed was incredibly smooth scrolling. And it's fast enough that I'm able to navigate well without feeling disoriented. I've seen other adventure games played under DOS and the scrolling has always been so rough that I could barely tell if the character was turning right or left.

The second thing I noticed was that although it is a shoot-'em-up game, it's not nearly as bloody as I had been lead to believe. Anyone who has seen video arcade games or the evening TV news should not be terribly bothered by the violence; you'll be too busy learning the floor plan and how to navigate to notice the blood, if I'm any judge.

DOOM is shareware. There are three adventures in the DOOM family; the first one is free, no strings attached, no guilt clauses telling you to register after 15

days or face legal action or moral rot. However, if you like the first adventure, there is a (reasonable) fee for purchasing the second and third adventures. I personally prefer this to guiltware (what Linus calls “limited-trialperiod shareware”).

My best recommendation for this product is that it is the first adventure game that has held my interest for more than a few minutes. My best recommendation against it is that you shouldn't start to play it if you don't have lots of spare time to devote to this game. You can blow away your friends by playing over the network. (I haven't tested this, but it's probably well done if it resembles the rest of the game.) Sound is supported if you have a sound card. I don't have one so I can't comment on the sound effects but I found the game perfectly playable with no sound.

A few tips (some of which are in the README.linux file, but you might miss them):

- If the screen is too dark to see easily, use the F11 key to change the “gamma correction”. There are four levels of gamma correction; press F11 repeatedly to cycle through them until you find the one you like best.
- Use a low-resolution video mode while playing DOOM. 640x480 looks good to me; DOOM uses a 320x200 window.
- If you are using *fvwm* as your window manager, you may have kept some default key settings that move you around on the virtual desktop. Some of these keys may be used as movement keys and, because of the combinations you can have, almost any SHIFT-, CONTROL-, or ALT-ARROW key combination may be used in DOOM. You might consider an alternate **.fvrmrc** file which does not set up these keybindings.

DOOM Resources

Michael K. Johnson is the editor of *Linux Journal*, and is also the author of the Linux Kernel Hackers' Guide. He welcomes your comments.

Archive Index Issue Table of Contents

Advanced search

Making TeX Work

Vince Skahan

Issue #8, December 1994

I was looking at this book as an opportunity to learn about all the little gotchas that just aren't made visible except by oral tradition from guru to wanna-be.

- **Author**: Norman Walsh
- **Publisher**: O'Reilly & Associates, Inc.
- **ISBN**: 1-5659Z-051-1
- **Price**: $29.95
- **Reviewer**: Vince Skahan

I'm an O'Reilly & Associates fan; a big fan. I went through my bookcases at home and at work and counted 19 different O'Reilly books that I've purchased with my own hard-earned cash. I refer to them almost daily.

I have one of the major commercial technical publishing packages available to me at work. I have a copy of Microsoft Word for Windows at home. I like their WYSIWYG look and feel and I like their intrinsic (to me at least) ease of use.

I'm admittedly not a big TeX fan at all; and I even participate quite a bit in the Linux DOC Project, which uses LaTeX as its document mark-up language of choice.

OK, call me a Linux heretic.

I was looking at this book as an opportunity to learn about all the little gotchas that just aren't made visible except by oral tradition from guru to wanna-be. Maybe after reading the book and trying some things, I'll be less TeX challenged. Naaaaahhhh...

Making TeX Work is a book that somehow disappoints me. The book follows the normal exceptionally high quality O'Reilly & Associates standards for

organization, format, binding, and appearance. There's 469 pages simply chock-full of information regarding TeX and associated utilities. I just never figured out why a generic Linux user needs to buy it.

Stated Goal of the Book

The Summer 1994 'ora.com' catalog/magazine from O'Reilly & Associates has an article on this book that describes it as “a complete reference to this complex typesetting system”. It also states:

> “We didn't want to do the same kind of book so many other people had written; describing the TeX language itself and how you can use it to write your documents in one of the many flavors of TeX macro languages... instead of writing a book on TeX, the text processor, we'd write a book on the entire TeX system.”

Cool—a book on how to put all the pieces together and make a document happen...

Later in the article, they mention that:

> “In addition to the basic TeX program itself, you need many other tools to write even a moderately complete TeX document. You need to understand how TeX uses macro packages and format files, fonts, pictures, figures, and a host of utilities... ”

Hey wait a minute. I'm getting this feeling it won't have hands-on examples...

“*Making TeX Work* guides you through this maze of tools and tells you how you can obtain them...”

Danger Will Robinson! Sounds like a how to find TeX sources FAQ...

1. An Introduction to TeX

The Big Picture describes what TeX is and what its goals are. It describes a little about how TeX formats pages to look nice, about the differences between text formatting, word processing and desktop publishing, and describes the underlying guts of how TeX does its magic. It's ugly in there, kiddies...

Editing briefly describes some of the usual editors that have varying abilities to provide a good environment for writing TeX documents.

This chapter disappointed me somewhat as I was looking for more information regarding exactly how to do something in typical editors. I was hoping that

they'd pick an editor (even [ugh...] Emacs) and give some real details regarding how to hook in and use the editor to efficiently write TeX documents.

Running TeX describes how to execute the program and deal with some of the most common errors in the documents.

Macro Packages describes some of the different formats such as TeX, LaTeX, and TeXinfo as well as some of the more obscure special purpose formats such as MusicTeX. While I found them interesting, there is probably nothing here that you can't find out about elsewhere on Usenet.

There were some comments about how to do TeX in color and how to print 'em on a black and white printer that I found rather interesting.

2. Elements of a Complex Document

Fonts describes in laborious detail more than any sane person would want to know about font selection and generation. While it's a nice amount of background information, it sure scared the heck out of me, and I'm probably a typical member of the book's target audience.

Pictures and Figures talks a little about the fact that if you try hard enough, you can draw pictures in TeX. It also gives a few examples about how to include external images in your document.

One nice addition is a description of some of the many image generation, viewer, editor, and manipulation programs commonly available such as xv and ImageMagick. People who plan to provide WorldWideWeb pages with embedded graphics might be interested in this chapter even if TeX and friends scare you off.

International Considerations describes the issues (and tools to deal with them) involved in writing nonEnglish documents in TeX.

Printing talks about utilities that generate the appropriate fonts to be able to print what you can preview.

Previewing describes some of the various viewers for MS-Windows, X, DOS (non-Windows), and Unix terminals.

Online Documentation talks about how to make your TeX documentation available online in some of the more common formats such as TeXinfo, HTML, or ASCII.

Introducing METAFONT describes the mechanics of building non-standard fonts. Again, I wonder whether this issue isn't far beyond the probable ability (or interest) of the book's target audience.

Bibliographies, Indexes, and Glossaries talks about how to use BibTeX to automate citations of references into a bibliography and also about many of the utilities available to automate the creation and manipulation of a bibliography database. It also describes how to construct an index and glossary by annotating your TeX documents.

3. A Tools Overview

Non-commercial Environments discusses free and shareware TeX systems.

Commercial Environments talks about commercial versions.

TeX on the Macintosh describes utilities that are Mac-specific. I don't know why it seemed necessary to have a separate chapter in order to do this.

TeX Utilities talks about many of the commonly available TeX utilities available on the CTAN archives.

Appendices

Filename Extension Summary describes about 55 different filename extensions you may run across that are related to TeX documents.

Font Samples gives 45 pages of font encoding tables and sample print from many of the METAFONT fonts.

Resources has 30 pages of extremely brief listings regarding how to a variety of TeX-related shells, editors, formats, and utilities.

Long Examples has 45 pages of scripts and other programs that are mentioned elsewhere in the book. The book examples are also available electronically on ftp.uu.net.

Conclusion

Making TeX Work has an incredible amount of information in it that you may someday either need to find or need to know. The problem is whether or not you need to spend $30.00 to find this information.

If you're a System Administrator who wants to cut to the chase regarding which public domain, commercial, or shareware software to acquire for your MS-

Windows PC or non-Linux (huh?) Unix system then I'd give it a “maybe”. There's an enormous number of skeletal descriptions of the hundreds of utilities that exist out there on Internet or are available commercially, that the book makes available for you in one nice neat place.

If you're more of a potential TeX user, I'm not so sure that you'd be well served in buying this book. There's enough description of the basic pdnciples that it serves as a good starting point before you go and buy one of the more common TeX or LaTeX books to give you the details regarding the language. On the other hand, if you buy one of the other books you'd have that basic information already.

I suppose if forced to make a recommendation, I'd recommend that a Linux user who was looking to get into the world of TeX save their pennies and:

- grab a recent copy of Slackware and install the whole T series of kits. Voila! You're most of the way toward having a functional TeX environment.
- for how to do a single document, grab Matt Welsh's “linuxdoc-sgml” package. Write your document in SGML. Use Matt's package to convert it to LaTeX.
- for how to write a big multi-part document, grab the sources for one of the larger Linux DOC Project documents (like Olaf Kirch's Network Administration Guide) and use it as an example.
- to get the list of Linux add-one for TeX document production, use the WorldWideWeb index to the TeX CTAN archives by getting into Mosaic on your local Internet site. See the comp. text . tex Usenet group for the Web URL for the archives.

Archive Index Issue Table of Contents

Advanced search

Linux—vom PC zur Workstation Grundlagen

Martin Sckopke

Issue #8, December 1994

The English version of Linux—Unleashing the Workstation in Your PC will be available by the time this issue is published.

- **Authors**: Stefan Strobel, Thomas Uhl
- **Publisher**: Springer-Verlag Bertin Heidelberg 1994
- **Price**: 38.- DM
- **ISBN**: 3-540-57383-6
- **Reviewer**: Martin Sckopke

Der grösste Teil der Dokumentation zu Linux ist auf Englisch, da Linux und der Informationsaustausch darüber grösstenteils auf dem Internet stattfinden und die Sprache der Wahl dort Englisch ist. Den meisten 'Hackern' macht das wohl nichts aus und einige Ubersetzungen ins Deutsche hätten sich die Verlage besser schenken sollen. Als Beispiel dafür mag die erste Ausgabe von Kernighan/Richies *The C Programming Language* dienen, in der Ubersetzung ist dort vom 'Zusammenbinden der Büchereien' die Rede.

Trotzdem ist es für viele Linux-Liebhaberinnen und solche, die es werden wollen, oft einfacher, ein Buch in ihrer Muttersprache zu lesen. Deswegen finde ich es sehr erfreulich, dass sich in den letzten Monaten gleich mehrere Autoren der Aufgabe angenommen haben, eine Einführung und Hilfestellung zu Linux auf Deutsch zu geben.

Das Buch *Linux—vom PC zur Workstation* beschreibt in lockerer Form die Installation, Administration und Benutzung eines Linux-Systems. Es beginnt mit einer Einführung in Multiuser/Multitasking-Systeme, beschreibt danach die gebräuchlichsten NetzwerkProgramme und Protokolle (TCP/IP, r-Utilities, NFS, RPC, NIS usw.), sowie die Besonderheiten von Linux im Vergleich mit anderen Unix-Systemen (Virtuelle Konsolen, Filesysteme, DOSEMU). Einen weiteren Abschnitt nehmen die verschiedenen LinuxDistributionen, die Konfiguration

des Kernels und die Systemadministration ein. Auch die Installation und Benutzung von XWindows sowie die gebrüuchlichsten Programmiersprachen und -werkzeuge werden kurz behandelt. Auch einige der gebräuchlichsten Anwendungsprogramme und Spiele werden beschrieben, Netzwerkanwendungen wie News, Mail, Gopher und WWW fehlen ebenfalls nicht. Den Abschluss bildet ein Verzeichnis der wichtigsten /etc-Dateien und ihrer Funktionen, eine Literaturliste und verschiedene FTPSener, sowie Hinweise auf weiter führende Informationen (HOWTOs, FAQs usw.).

Leider ist allerdings gerade dieser letzte Teil besonders wichtig, da das Buch alle obengenannten Themen nur kurz streift, die Autoren bleiben immer an der Oberfläche, ohne die auftretenden Probleme und Lösungen dazu zu beschreiben. Das ist auch auf dem zur Verfügung stehenden Platz recht schwer, denn die Seiten sind nur zu ca. 2/3 mit Text gefüllt, der Rand bleibt für Schlagworte grösstenteils frei. *Linux—vom PC zur Workstation* gibt einen guten Uberblick über die Möglichkeiten, die ein PC mit Linux bieten kann und richtet sich an Neulinge, die sich über Linux allgemein informieren wollen. Zum Nachschlagen für Benutzer und

Systemverwalter ist es meiner Meinung nach nicht geeignet.

Note to readers of English: The English version of Linux—Unleashing the Workstation in Your PC will be available by the time this issue is published. *Linux Journal* will provide an English review of the book in an upcoming issue.

Archive Index Issue Table of Contents

Advanced search

UNIX: An Open Systems Dictionary

Laurie Tucker

Issue #8, December 1994

The book includes over 6,000 entries and does a fairly good job of keeping definitions as free of jargon as possible.

- **Authors**: William H. Holt, Rockie,. Morgan
- **Publisher**: Resolution Business Press
- **ISBN**: 0-94526414-3
- **Price**: 524.95
- **Reviewer**: Laurie L. Tucker

When my boss first picked up this book, he went in search of the word “Linux”. To his great surprise he found it, and then declared that this was a book worth having. After using this dictionary for the past two months, I have to agree; and not just because it contains a definition for Linux in it.

The book includes over 6,000 entries and does a fairly good job of keeping definitions as free of jargon as possible. As a result, it can be used by people with a broad range of Unix experience, from the “newbie” (not defined, but we know what that is!) to the “wizard”.

As the Assistant Editor for *Linux Journal*, and a fairly new sysadmin, the book has come in handy quite a few times. I've used it to figure out what POSIX really stands for (Portable Operating System Interface for computer environments (X)), and I've used it to better understand what I read in articles that are submitted to *Linux Journal* for publication.

The book contains such basic terms as: pop-up window (with a figure showing one), command line, directory, port, space bar, kernel (with the standard bull's-eye graphic), software, CPU, and edit. These are all described so that true computer novices can better understand computers. Lots of acronyms are included, like ASCII, FTP, TCP/IP, VMS, SCSI, RISC, MTA, and LAN.

There's information on Unix-style word processing: serif, sans serif, roff, nroff, troff, and mm macros.

Important people: Dennis Ritchie and Brian Kernighan.

Unix operating systems: SVR, BSD, UNIX, XENIX, and SunOS.

The Information superhighway: Internet, WWW, nslookup, archie, gopher, WAIS, hypertext, T-1, PPP.

Sysadmin terms: wtmp, sendmail.cf, mountd, mnttab, named. local, /dev/null, DNS, telinit, a whole bunch of /etc/* entries.

This book even contains historical gossip about the Michelangelo virus!

What don't I like about the book? It includes pronunciations of words, like WA-BEE, SCUZZY, NAME-D, NROFF, GOO-IE, etc., listed as entnes. I think it's kind of hokey. But these pronunciations are also included with the “real” definitions, and the dictionary does a very good job of cross-referencing.

It also includes a scattering of figures and tables which enhance the text definitions.

At the end of the book there are a number of useful appendices, including references for basic vi commands, basic Emacs commands, FTP commands, lpc commands, RFS parameters, signal values (preSVR4, SVR4, BSD), Telnet stuff, and some commonly used talk-mode jargon.

Five years ago, Bill Holt decided that this was a book he needed. Since there wasn't one available, he and Rockie Morgan wrote it. That's one of the best reasons for creating something, and this dictionary is something I'm glad I have.

Laurie Tucker (info@linuxjournal.com) is the assistant editor of *Linux Journal*, cover designer of the September issue, and sysadmin of linuxjournal.com; a Linux system

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #8, December 1994

Readers sound off.

Load What?

Nice job by Matt Welsh on Emacs in Linux oumal issue #6. It really got me going! Encountered minor text bug using our 18.57 version. When trying to byte compile code, the “load-file” was missing a target file.

```
(defun byte-compile-if-newer-and
load (file)
"Byte compile file.el if newer
than file.elc"
(if (filenewer-than-file-p
(concat file ,'.el")
        (concat file ".elc"))
  (byte-compile-file (concat
file ".el")))
(load-file ) )
  (byte-compile-if-newer-and-load
"~/emacs/startup")
```

FIX:

```
(load-file (concat file " .elc") ) )
```

—Erik “Rev” Feddersenrev@datacube.com

Good Guy Gets Better

In the past few months, I've made a number of statements about Linux in your fonum stating that Linux documentation is worthless, Linux code is unreadable, and some Linux soffware is annoyingly unreliable.

I would like to retract, and apologize for, any statements I made about documentation or code. Documentation problems resuited from my not having read the Linux installation brochure carefully enough and from having missed the whence command. My code problems resulted from having picked a

particularly obscure and locally undocumented bit of code (profil. c) as the first arid only test case. Having now used whence a few times and browsed around some other code in the LGX (Yggdrasil) Linux distribution, I find that the quality of almost all documentation and code readability ranges between pretty good and very good.

I still regard Linux as noticeably less reliable as a whole than the other Unix system I've used extensively, namely SunOS. Although I have found the Linux kernel Qile system, process scheduler, etc.) and shell to be extremely reliable, there are several specific problems in other parts of the system that are a source of ongoing annoyance:

- If the X server ever nuns out of swap space, the entire machine locks up and I have to reboot. (I usually can't just shut down the server gracefully.)
- gdb gets badly confused by optimized code, showing bogus variable values with no indication that they are bogus.
- Attempting to debug profiled (-pg) code with gab offen results in strange, non-resumable traps when singlestepping.
- Profiling sometimes produces very large variations in observed times when running under X-Windows. I find I have to shut down X in order to get repeatable results.
- Having done a partially CD-ROM-based installation, I find the CD-ROM being invoked at unpredictable times and for odd functions; for example, just to do a ls 1 in the current (non-CD-ROM) directory.

Despite these problems, I have found Linux to be an excellent value for the price I would have paid for it (S99) if Yggdrasil hadn't sent me a free copy S99 won't even buy a decent C compiler in the PC world.L. Peter Deutsch,Aladdin Enterprisesghost@aladdin.com

YGGDRASIL RESPONDS:

...we have rearranged the order in which elements of SPATH are searched in the Fall 1994 release to reduce some of the CDROM accesses....

A minor correchon on pricing:Beta Release $60($99 for Beta + Fall 1993)Fall 1993 $49.99Summer 1994 $39.99Fall 1994 $34.99Adam I Richter,Yggdrasil Company Inc.adam@adam.yggdrasil.com

Linux Morality

I am a fairly experienced Unix user who has MSDOS on his home system. (I have used Unix for years at the low and high level at work and school.) I've been piddling with the idea of putting Linux on my box at home, and I even

have a new 1GB hard drive (in addition to the 500MB one) just screaming to have Linux run its hands all over the cylinders. Affer reading the HOWTOs and the installguide, I was sure I could install Linux on my PC. I've installed other Unix OS stuff on other machines (okay, once).

However, your article “Cooking with Linux” (issue #5), finally pushed me over the edge. It leff me rolling on the floor in some places and it leff me with a feeling that I have a moral responsibility to wake up and use that other 95% of my brain.

Oh, and it convinced me to install Linux at home, too.

Thanks.Lewis W. Beardlewis@damops.wes.army.mil

Making Faces

Thank you for your article entitled “Emacs: Friend or Foe?” in issue #5 of Linux lournaL I was wondering if you might have an idea as how to set the color schemes in C mode for such things as reserved words, comments, strings, etc.Thanks in advance.Mike Clarkaseng@bsinet.com

MATT RESPONDS

The article describes how to set the faces “bold”, “underline”, “italic” and so forth. These are achually used within C mode to set colors for reserved words, stnngs, and so forth.

However, if you do **M-x apropos** face you will get a list of functions and variables containing the word “face”. These include:

```
font-lock-comment-face
        Variable: Face to use for comments.
font-lock-doc-string-face
        Variable:       Face to use for documentation strings.
font-lock-function-name-face
        Variable:       Face to use for function names.
font-lock-keyword- face
        Variable: Face to use for keywords.
font-lock-string-face
        Variable:       Face to use for string constants.
font-lock-type-face
Variable:       Face to use for data types.
```

Apparently, these are the faces used within Font Lock mode (described in the article) for the features you mentioned. I have notpersonally experimented with these faces, and they appear to be undocumented. Your mileage may vary. 10

Archive Index Issue Table of Contents

Advanced search

Linux Journal at Unix Expo

LJ Staff

Issue #8, December 1994

A report on our booth and the perception of Linux amoung Open Systems users.

Unix Expo promotes itself as the largest trade show for corporate buyers and resellers interested in Unix and Open Systems Computing, with an estimated 30,000 attendees this year. The show ran from October 4th to 6th at the Javits Center in New York City.

Linux Journal's booth at Unix Expo proved to be a fun but harried spot for our Advertising Manager, Joanne Wagner, our Associate Publisher, Belinda Frazier and our Editor, Michael Johnson. Linux Joumal gave away thousands of copies of the October, 1994, issue of *Linux Journal*, our annual Buyers issue. People especially admired the “Linux The Virtual Brewery” T-shirt and many returned to our booth for our occasional drawings for Tshirts, mugs, and Linux CDs donated by Linux vendors.

Due to show regulations we were not able to sell Linux CDs, but we could have easily sold hundreds of CDs, even to people who had just recently heard of Linux. More than once, a manager told us, “All my engineers are using Linux! I have to find out more about it.”

Unix Expo offered user groups the chance to meet in the early evening hours. Over a hundred people gathered for the New York Unix/Linux Group's meeting which included a panel discussion with Michael Johnston of Morse Telecommunication, Marc Ewing of Red Hat Software, and Michael Johnson, Linux [oumal Editor and general techie. Beta copies of Red Hat Software's new Linux distribution were given out at the meeting.

At the end of the discussion, HJ. Lu, the Linux GCC maintainer, was invited to join the group to talk about his work.

Probably the most spectacular Linux product at Unix Expo was the accelerated-d X server from X Inside. Thomas Roell, the author of the original X386 server from which XFree86 was also developed, is the developer of the accelerated-d server, and it shows.

Several hardware vendors dropped by our booth to sa~ that they have had many customers asking about Linux drivers for their products. The vendors had two main concerns. The first was **copyright**. It is difficult for many vendors to release proprietary information in the form of publically available source code for drivers for their hard ware, no matter how convincing the arguments they hear. They also aren't clear on some points of copyright.

A representative of Adaptec stopped by to say that her company has been getting many requests for Linux drivers for their highend cards, but Adaptec was unclear about some copyright issues. Instead of putting firmware in a ROM chip on their high-end boards, Adaptec downloaded equivalent firmware into RAM. This makes their cards easier to upgrade and more flexible.

Adaptec, however, considers that fimmware to be just as proprietary as if it were in a ROM chip and will not release the source code to it just because it is downloaded by GNU Public License-protected code. When we explained that the GPL does not require Adaptec to release that source code with the source code for a Linux kemel driver, she was pleased and interested in helping Linux developers with a driver.

The second main question involved **support**. Several vendors were quite interested in making drivers available for their hardware, but do not (for now, at least) feel that they have resources to support the driver. We offered to help find volunteers who could support a driver, given all needed technical infommation and free hardware.

Here is an incomplete list of companies that expressed some sort of positive interest in Linux: Comeau Computing, ComputerWorld, DigiBoard, Eicon, Enhanced Software Technologies, Equinox, Frame Technology, SCO World, Stallion Technologies, UnixOPEN, UnixWare Technology Group, and Z-Code Software.

It was enjoyable to chat with the very many Linux users who stopped by, despite the fact that our voices were nearly destroyed by the end of the show. Of course, there were a few people who had difficult questions, and some questions didn't get answered. What I found most interesting, though, was the very large number of users who had installed Linux in the week preceding the show; it seemed like about a third of all the users we talked to were that new to Linux, which paints a picture of very rapid growth.

New Products

LJ Staff

Issue #8, December 1994

Red Hat Software offers Linux on CD-ROM, SoftStar announces Network Management System and more.

Red Hat Software offers Linux on CD-ROM

Red Hat has released a new distribution of Linux, RHS Linux. RHS Linux comes on CD-ROM only at this time, and features automated installation and X configuration, simplified system management tools, package installation and un-installation tools, and documentation. Source for all the binary packages is included in the same package format used to install the binary packages. RHS Linux complies with the Linux FSSTND, the standard that defines the location of most files on Linux systems, allowing you to install any FSSTNDcompliant binary package on your system without conflicts.

Red Hat Software can be reached at Red Hat Software, P.O. Box 4325, Chapel Hill, NC 27515; phone (919) 3099560; or info@redhat.com.

SoftStar announces Network Management System

Soft*Star s.r.1. has released a Linux version of NetEye, a network management system based on OSF/Motif, SNMP, and a relational database. NetEye has very powerful support for monitoring networks and provides an intuitive graphical user interface that allows relatively inexperienced network administrators to effectively manage networks. It includes management of trouble reports, active network monitoring, reactive SNMP trap handling, and extensive reporting.

Soft*Star can be reached at Soft*Star s.r.1., Via Camburzano 9, 10143 Torino, Italy; +39 11 74 6092; fax +39 11 74 6487.

VersaSoft releases dBMAN

VersaSoft has released dBMAN, a dBASE III+ compatible DBMS, for Linux. It includes ad hoc query and edit capability, a report generator that does not require programming, scrollable windows (using termcap with color support), multi-dimensional arrays, data encryption and protection at the field level, source compatibility with dBASE III+ applications, extensions to dBASE III+ that are completely compatible over the more than 30 platforms supported by dBMAN, and a source-level debugger, yet requires less than 2MB of disk space to install.

VersaSoft can be reached at VersaSoft Corporation, 4340 Almaden Expressway, Suite 110, San Jose, CA 95118; phone (408) 723-9044; or fax (408) 723-9046.

ObjectProDSP from Mountain Math Software

Mountain Math Software has just released a new DSP environment, licensed under the GNU General Public License. ObjectProDSP allows you to create, test, and run DSP networks, and it can support some DSP hardware. It includes approximately 400 pages of documentation and validation test suites. ObjectProDSP is designed to create self-documenting projects. This is demonstrated by the fact that much of the documentation that comes with ObjectProDSP is generated in this way.

ObjectProDSP can be downloaded via ftp from tsxll.mit.edu in /pub/linux/packages/dsp, and from sunsite.unc.edu in /pub/Linux/devel/opd.

Mountain Math Software can be reached at Mountain Math Software, P.O. Box 2124, Saratoga, CA 95070; phone (408) 353-3989; or support@mtnmath.com.

Alpha Base Systems releases Linux port of MAE

ABS has ported their Metropolis Applications Environment to Linux. MAE integrates a DBMS with a character-mode windowing system called “flip” and a personal organization manager called “Assistant Plus” which integrates e-mail, phone message taking, scratch pad, appointment scheduler, to-do lists, and a profile list manager.

Alpha Base can be reached at (213) 850-6577; fax (213) 876-0986.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Kernel Korner

Michael K. Johnson

Issue #8, December 1994

This month, we inaugurate a new column which will cover the Linux kernel. Material presented in this column will be used to further the Linux Kernal Hacker's Guide.

Most of the abstract functionality that is needed in a kernel is already in the Linux kernel. Linux has one of the best-designed buffer caches of any Unix-like operating system. A few things are left to be completed in the memory management and in the networking layer, but as more and more development is done, it seems (and reasonably so) that there are fewer and fewer projects that can be done by people with little operating systems knowledge and experience. Those who know enough to know what needs to be done with the memory management don't need to read documentation on how the kernel works; many of them find it faster to simply read the Linux source code.

That is not to say that there = are few projects in the Linux community that beginners can do; there are very many, but most of these projects are not within the kernel itself.

One project that relative beginners can accomplish, and which will never go out of fashion, is writing device drivers for new hardware. There is still hardware that Linux does not support, new hardware is being released by manufacturers all the time, and Linux users buy hardware and then want to use it with Linux.

Fortunately, the interface used to write device drivers is relatively simple and clean. By clean, I mean that there are not many exceptions to the rules or little tricks you have to play to get things to work. Over the next few columns (few is relative, I could spend a few years at this...) I will cover the information you need to know to write various kinds of device drivers.

Some of this information is already in the *Linux Kernel Hackers' Guide*, but I will expand on the information here. When I write something new here, it will

eventually find its way into the *Kernel Hackers' Guide*; this is one way that *Linux Journal* supports the Linux Documentation Project.

Devices in User Space

In an almost contradictory way, I'm to initiate this **Kernel Korner** column with a description of how (and when!) to implement a device driver as a user program.

The first rule of adding code to the Linux kernel is don't. The code that is in the kernel cannot be swapped, and therefore takes up precious memory whether it is being used at the time or not. Many hardware devices can be driven by user-space programs which are kept nicely out of the way (either swapped out or not running at all) when the device is not being used. One prime example of devices that are implemented this way are video cards.

While the Linux startup code has options to initialize the video modes for many different kinds of cards, the actual device support for video cards in the Linux kernel proper is extremely limited, and is comprised of support for putting text on the screen on either monochrome (hercules-style) or color (CGA, EGA, VGA, and above) cards. No support for graphics is included.

XFree86 provides user-level drivers for many graphics cards within the X servers. These are only loaded when the X servers are running, and parts that aren't being used at the moment can be swapped out when necessary. In addition, by not making the device use a system-call interface to write, these drivers are faster because they are implemented in user space.

There are, of course, drivers that cannot be written as user-space drivers: most commonly, hardware that requires a driver that can service interrupts. We'll deal with these in a future installment.

Talking to Hardware

Perhaps the most common way that a device driver communicates with hardware (at least on the PC architecture) is through the I/O bus. This is a bus which is completely separate from the memory bus, and which is accessed with special machine instructions. For a concrete example, let's use the parallel port. The parallel port driver is in the kernel for three reasons: it can be interrupt-driven, it manages contention, and it has historically been part of the kernel. It's also reasonably small, and very common, so it doesn't bloat the kernel very much. However, the parallel port can be driven from user space. Let's look at how this could be done.

The parallel port has three addresses on the I/O bus, and they are specified by a base address and two offsets. This is common for devices; many devices have

several base addresses to choose from, and any other ports that are used are specified as offsets from the base. The three base addresses for the parallel port are given in **linux/lp.h**, and are (in hex) Ox3bc, Ox378, and Ox278. The status port is the next port above that, and the control port is above the status port. So if the base I/O port, to which characters are written, is Ox378, then the status port is Ox379 and the control port is Ox380.

Perhaps the most common way that a device driver communicates with hardware... is through the 1/0 bus.

You need enough documentation for a device to know how to talk to it. The 8255 chip is the chip that the parallel port is based on, and the documentation for that chip and for the parallel port interface describes the three ports.

The status port can report several conditions when it is read:

Bit	Condition
Ox80	If 0, printer is busy
Ox40	If 0, printer has ACKnowledged the character sent
Ox20	If 1, printer is out of paper
OxlO	If 1, printer is on-line
Ox08	If 0, printer has in an error condition

The control port controls several things when it is written:

Bit	Condition
OxlO	Set to 1 to enable sending interrupts when the printer is ready
Ox08	Set to 1 to tell printer ready to talk
Ox04	Set to O to tell printer to initialize itself
OxO1	Set to 1 to prepare to send another byte to printer

Unfortunately, not all printers agree about all the signals that can be sent, so the least common denominator has to be used. This means that I won't use all of the bits you see in the tables. Also, I obviously won't use the interrupt-enable bit, since interrupts can't be used from user-level programs.

I'm also not going to do any serious error detection; I want to show how simple it can be to write a simple driver that works (more or less). If you want to see how error detection could be handled, simply read **include/linux/lp.h** and **drivers/char/lp.c** in the Linux kernel source.

The program **userlp.c** (see sidebar) needs to be compiled with optimization fumed on and run as root (or setuid root) to work. It takes a file from the standard input and prints it to the printer specified on the command line: O, 1, or 2, corresponding to lpO, lpi, and lp2, respectively.

This has only been lightly tested on one printer, unlike the standard kernel driver, so I can't say that it will work on your printer. This doesn't matter, because this is only an example. Note that I could have written the same driver as a /bin/sh script that used **/dev/port** and **dd**, and probably done it in less time, but you are more likely to be writing a device driver in C than in **/bin/sh**.

Caveats

In order to use **inb_p()** and **ou tb_b()**, not only did I have to compile with optimization on and use **ioperm()** to allow access to those ports, I also had to use **ioperm()** to allow access to port Ox80. This is because the ***b_p ()** functions use port Ox80 to slow down port access.

I was also lucky in that all my ports were less than Ox3ff. To access ports higher than Ox3ff, you either need to use **/dev/port** (as will be described below) or, for fastest access, use the **iopl()** function to set your I/O protection level to “ring 3”, the same as the kernel. This is unfortunate (although there are good reasons for it; read **kernel/ioport.h** if you care), because it means that you can access any port at all, and if you access the wrong one through some programming error, you may much more easily mess up the entire machine. Imagine what will happen if your program accidentally writes “random” values to one of the I/O ports that controls the hard drive. At “ring 3”, code is nearly as powerful as the kernel, and so one of the advantages of a user-level driver is gone.

If you are going to do something as dangerous as use iopl() to put your code in ring 3, you should probably know how to read kernel source code, so I will simply refer you to **kernel/ioport.h** for details. System calls are called sys_*name* within the kernel, so look for sys **iopl()**.

if (netuid_root == BLEcH || iopl(3) == SCARY)

Note that I used the **ioperm()** function to read and write directly from and to the ports with the **inb_p()** and **outb_b()** functions, and that this function requires that the code run as root. Another option is to read and write from **/dev/port**. This is a little slower, but has the advantage that the code does not require root permissions to run; just read and write permission to **/dev/port**. Simply use **lseek()** to seek to the address of the port you want to read from or write to, and **read()** or **write()** a single byte to the file. If you want to read or write again, you need to use **lseek()** again. If you make a group called port and make **/dev/port** readable and writable by group port, then any user in group port can use user-space device drivers written in this way without the programs being setuid root.

Another way to access **/dev/port** is to use **mmap()** to map it into some memory space. Then you can write to ports directly at the memory address you map them to. See the section on memory mapping, below, to learn how to map files; the details (other than the filename) are the same. Since perl can use the **mmap()** call, it is possible to write device drivers that access **/dev/port** and **/dev/mem** as perl scripts.

Memory

Other devices may need to be accessed at some place in physical memory. The first 3GB of physical memory (if you have more memory than that and don't know how to access the 4th gigabyte, you don't have my sympathy...) can be accessed through **/dev/mem**. The sidebar (at left on page 20) gives a rough version of the **mmap()** code from svgalib, which, like XFree86, is a user-space device driver for video cards:

The code first opens /dev/mem, then allocates enough memory to map the section of /dev/mem it wants into, and then maps /dev/mem over the already allocated memory. Once this has been successfully done, whenever that process writes to or reads from that memory, it is writing to or reading from physical memory at the address that **/dev/mem** was mapped to.

Cute Note

Since perl can use the **mmap()** call, it is possible to write device drivers that access **/dev/port** and **/dev/mem** as perl scripts. If you don't already use perl, it's probably not worth it, but if you do use perl, you may find the idea intriguing. If you try it, I'd like to know how it works for you, and if you have any hints, I may pass them on to the readers of this column. Similarly, it is technically possible (although in practice “too clever by half” and rather slow) to write a device driver as a shell script, by using **dd** to read and write ports. Just to be contrary, I worked on such a driver, and found that the chief problems are the lack of

binary bit-wise operations and lack of real binary data. I am not distributing this shell script; anyone who seriously cares about playing in this way can cook up their own based on the **userlp.c** file presented in this column. If you get it to work reliably, please notify me, and I may print your version in a future Kernel Korner.

Listing 1

Listing 2

Michael K. Johnson is the editor of *Linux Journal*, and is also the author of the Linux Kernel Hackers' Guide. He welcomes your comments.

Archive Index Issue Table of Contents

Advanced search