Advanced search

Linux Journal Issue #5/September 1994

Columns

Archive Index

Advanced search

Emacs: Friend or Foe?

Matt Welsh

Issue #5, September 1994

Frustrated with Emacs? Here's how to wrestle it into submission.

If you're like me, you find Emacs somewhat intimidating. Well, if you're like me, you've been an exclusive **vi** user for years, and have always wanted to start using Emacs, but have had trouble doing so. Each time you try to switch over, you find yourself pressing <esc> after every third sentence, which (in Emacs) produces some random, and usually undesirable, result. Once, I was thrown into the infamous “Doctor” mode, which is a simple AI program emulating a psychiatrist. There, I spent a good ten minutes telling Emacs what I thought of it. (The response was, **"Perhaps you could try to be less abusive."**)

Using one of Emacs' **vi**-emulation modes isn't of much help, either; none of them seem to handle even basic vi commands appropriately. And if you tend to use less-well-known vi features, you're simply out of luck. (Ever try changing the window size with *nnnzwww^*? I didn't think so.)

As impossible as it may seem, you can find a place for Emacs in your everyday life. If you have the diskspace and memory (a far-from-negligible detail), I strongly suggest looking into Emacs; it can make your life easier, even if you think that you're perfectly content with **vi**.

Whether you're accustomed to vi or not, however, the default Emacs configuration on many systems is, well, less than adequate. While many users learn to live with Emacs “out of the box”, I've never been content with that approach. Instead, you can get Emacs to do just about whatever you want it to do. So, let's take a look at some customizations that I've found particularly useful.

All of the customizations discussed here involve editing the Emacs configuration file, which is **~/.emacs** by default. This file is written in Emacs LISP (Elisp), which is the internal LISP engine that Emacs uses for nearly everything.

For example, keystrokes map to Emacs LISP commands. You can modify the command to be executed for each key sequence, and even write your own Elisp functions to bind to keys.

If you're not a LISP programmer, never fear. Most of the LISP forms described here are quite simple and don't require a degree in artificial intelligence to comprehend.

I do assume that you have at least tried to use Emacs, and know how to get into and out of it, are familiar with the Info documentation system, and so on. If not, fire up Emacs and type **Ctrl-h (C-h)** followed by t, which will drop you into a tutorial. (From there, you're on your own.) The following customizations work under Emacs 19.24. By the time you read this article, there will more than likely be a newer version available, for which certain things may have changed.

Basics: Rebinding keys

Before we can customize Emacs, we need a customization file. By default this is **.emacs** in your home directory. (Later, we're going to move the contents of **.emacs** to another file, so buckle your seatbelt.)

Our first task is to rebind several keys to perform more reasonable actions. Admittedly, several of these key bindings are vi-oriented, but should make sense nonetheless. First of all, I like to page back and forth through the document with **C-f** and **C-b**, as in **vi**. In order to rebind these keys, we'd include these lines in our **.emacs** file:

```
; Modify meaning of C-f and C-b
(global-set-key "\C-f" 'scroll-up)
(global-set-key "\C-b" 'scroll-down)
```

As you can see, comments in Elisp begin with a semicolon and extend to the end of the line. Here, we invoke the function **global-set-key** twice; once for **C-f** and again for **C-b**. In LISP, expressions are parenthesized. The first item in each list is the name of the function to call, and it is followed by any arguments. The first argument to **global-set-key** is a string constant, representing the key sequence to bind. The second argument is the name of the Emacs LISP function to bind it to. The function name is single-quoted in order to refer to the function name itself, not the function that it points to.

One question remains: How did we know that **scroll-up** and **scroll-down** referred to the Emacs functions to scroll pagewise through the document? Other than reading the documentation, you could find this out by using the Emacs function **describe-key** (which you can invoke with **C-h k**). Calling this function and following it with a key sequence will tell you the Emacs function name that the key sequence maps to. If you knew that the Emacs keys **C-v** and

M-v (**Meta-v**, where “meta” is usually the <esc> key) scrolled back and forth through the document, you could use **C-h k** to determine the function names for these keys. For example, the sequence **C-h k C-v** will display the function name bound to **C-v**.

Here are a few more key rebindings that I use often:

```
(global-set-key "\C-v" 'end-of-buffer)
(global-set-key "\M-v" 'beginning-of-buffer)
(global-set-key "\C-p" 'yank)
(global-set-key "\C-u" 'undo)
(global-set-key "\C-r" 'overwrite-mode)
```

The first two commands rebind **C-v** and **M-v** (which previously scrolled by pages) to move to the beginning and end of the buffer, respectively. The third causes **C-p** to yank the previously deleted or copied region of text, and the fourth causes **C-u** to execute the undo command. (I tend to use the Emacs' undo feature quite a bit, and find **C-x u** or **C-_ too** cumbersome.) The last command causes **C-r** to toggle overwrite mode.

There are some caveats associated with rebinding keys in this manner. For one thing, it destroys the previous definition of the key. In many cases, this is fine; as long as you didn't use the key as previously defined. However, the last of the above commands rebinds **C-r**, which is used to invoke a backwards incremental search; a very useful feature. Before rebinding a key you should be sure that its previous definition doesn't mean much to you.

Multiple key sequences

Now, what about bindings involving multiple keys? For example, I like to use the key sequence **C-d C-d** to delete a line (somewhat like its **vi** counterpart), and **C-d g** to delete text from point to the end of the buffer. The second argument to **global-set-key** can include multiple keys, but the problem is that **C-d** already has a meaning. We want to use **C-d** as a “prefix” key for the commands **C-d C-d** and **C-d g**, so first we have to unbind **C-d**. Here's what we do:

```
; Various keys for nuking text (global-unset-key "\C-d")
(global-set-key "\C-d g" 'my-nuke-to-end)
(global-set-key "\C-d\C-d" 'my-nuke-line)
```

We simply use **global-unset-key** to unbind **C-d**, and then bind new meanings to **C-d g** and **C-d C-d**. However, these functions are bound to the mysterious functions **my-nuke-to-end**, and **my-nuke-line**, which the astute among you will notice aren't standard Emacs functions. We have to define them.

Defining a new function

Defining an Emacs LISP function is rather simple. Of course, Emacs functions can be quite powerful and complex, but in this case we're going to use the function to call a short sequence of other functions, which isn't so frightening. In general, if you need a key binding to call several functions in sequence, you must define a new function to wrap the sequence in.

Here is the definition of **my-nuke-to-end**, which should be placed above the corresponding call to **global-set-key**, which uses it, in the **.emacs** file.

```
(defun my-nuke-to-end ()
  "Nuke text from here to end of buffer."
  (interactive "*")
  (kill-region (point) (point-max)))
```

The defun function takes as arguments the function name to define, a list of arguments (which, here, is empty), followed by the body of the function. Note that the first line of the body is a string constant, which is a short description of the function. This string is displayed when **describe-key** or **describe-function** is used to display information about **my-nuke-to-end**.

The second line of the body is a call to the **interactive** function. This is required for functions that are bound to keys. It simply tells Emacs how to execute the function interactively (that is, when called from a key sequence). The argument to interactive, “*”, indicates that the function should not be executed within a read-only buffer. Check out the documentation for **interactive** if you want the gritty details. (See the sidebar “Getting the Emacs LISP Manual” for information on obtaining this documentation.)

The last line of the body uses the Emacs internal function **kill-region** to delete a region of text. The two functions point and **point-max** return the current location of point, and the position of the end of the buffer, respectively. **kill-region** deletes the text between these two locations.

The definition for **my-nuke-line** is somewhat more involved, because there is no single Emacs function that this operation maps to. Here it is:

```
(defun my-nuke-line (arg)
   "Nuke a line."
   (interactive "*p")
   (beginning-of-line nil)
   (kill-line arg)
   (if (= (point) (point-max))
       (progn
        (forward-line -1)
        (end-of-line nil)
        (kill-line arg)
        (beginning-of-line nil))))
```

First of all, we see that this function now takes an argument, which we have called **arg**. Many Emacs key functions take a single numeric argument, which you can specify by prefixing the key with **C-u**, followed by a number. (That is, unless you've rebound **C-u**, as we have.) This numeric argument changes the behavior of certain functions. Here, **arg** is passed along to **kill-line**, used on lines 4 and 9 of the function body.

my-nuke-line is essentially a wrapper for **kill-line**, but takes care of a special case for the last line in the buffer. In this case, we want to delete the newline before the last line, which causes the last line to be clipped out altogether (otherwise, Emacs deletes the line, but leaves a blank one in its place). After **interactive** is called (with the "*p" argument, which causes arg to be converted to a number), **beginning-of-line** moves point to (surprise!) the beginning of the line. **kill-line** is then invoked. Note that **kill-line** only kills text from point to the end of the line; not the entire line.

Next, we compare the cursor position (point) with the end-of-buffer position (**point-max**). If they are equal, then we are trying to delete the last line of the buffer, and want to kill the previous terminating newline. We move back a line (using **forward-line** with an argument of **-1**), move to the end of that line, kill the rest of the line (which consists only of the terminating newline), and move back to the beginning of the line. All of this results in the last line of the buffer being deleted.

I'm sure that there are Emacs gurus out there that can find much better ways to accomplish the same thing; please bear with me. I've been brainwashed by **vi**. One of Emacs' better features is that there are many ways to modify its behavior.

Byte-compiling configuration code

Now that you've had a crash course in Emacs LISP programming, let's move on to something more practical. Once you start customizing Emacs, you'll notice that your **.emacs** file gets to be quite large, and may take a while to load. You may be aware that Emacs allows you to byte-compile LISP source files for faster loading, so let's utilize that feature on our **.emacs** configuration file.

The first step is to create a directory for your personal Emacs LISP files to reside. At first this directory will contain only one file; namely, your initial configuration file; but later in life you may wish to write separate Elisp files. I use the directory **emacs** in my home directory for this purpose.

Next, copy your **.emacs** file to this directory, and rename it to something like **startup.el**.

Now, we replace the contents of **.emacs** with a short bit of code that byte-compiles **emacs/startup.el** and loads it. However, we only want to byte-compile **startup.el** if it is newer than its compiled counterpart, **startup.elc**. Here's the trick:

```
(defun byte-compile-if-newer-and-load (file)
   "Byte compile file.el if newer than file.elc"
    (if (file-newer-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")
```

This is blatantly obvious, I'm sure, but by way of explanation: this bit of code defines a new function, **byte-compile-if-newer-and-load** (keeping in line with the Emacs' affinity for verbose function names), and executes it on **~/emacs/startup.el**. We have now moved all of the Emacs configuration code to **startup.el** which is byte-compiled when necessary.

Emacs and X

Emacs and the X Window System are two good things that are great together. In fact, my primary motivation for starting to use Emacs for the four thousandth time was to have an editor that incorporated many of the nice features of X, such as mouse-based region cut-and-paste, and so forth. Emacs 19 has support for many useful X-based features, some of which I'll introduce here.

The first thing that you might want to do when using Emacs under X is customize the colors. I'm no fan of black and white; and in fact, I prefer lighter fonts on a dark background. While you can customize Emacs' X-specific attributes using the X resource database (e.g., by editing **~/.Xdefaults**), this isn't quite flexible enough. Instead, we can use Emacs internal functions such as **set-foreground-color**.

For example, in your **startup.el** file, you might include:

```
(set-foreground-color "white")
(set-background-color "dimgray")
(set-cursor-color "red")
```

which will set these colors appropriately.

Emacs also provides support for faces, used most commonly within **font-lock-mode**. In this mode, text in the current buffer is “fontified” so that, for example, C source comments appear in one font face, and C function names in another. Several of Emacs' major modes have support for font lock, including C mode, Info, Emacs-Lisp mode, and so on. Each mode has different rules for determining how text is fontified.

For simplicity, I employ faces of the same font, but which use different colors. For example, I set the “bold” face to light blue, and “bold-italic” to a sick shade of green. Each major mode has a different use for each face; for instance, within Info, the bold face is used to highlight node names, and within C mode, the bold-italic face is used for function names.

The Emacs functions **set-face-foreground** and **set-face-background** are used to set the colors corresponding to each face. For a list of available faces and their current display parameters, use the command **M-x list-faces-display**.

For example, I use the following commands in **startup.el** to configure faces:

```
  (set-face-foreground 'bold "lightblue")
  (set-face-foreground 'bold-italic "olivedrab2")
  (set-face-foreground 'italic "lightsteelblue")
  (set-face-foreground 'modeline "white")
  (set-face-background 'modeline "black")
  (set-face-background 'highlight "blue")
  (set-face-underline-p 'bold nil)
  (set-face-underline-p 'underline nil)
```

The modeline face (which is referred to in the Emacs documentation as mode-line, for some reason) is used for the mode line and menu bar. Also, the function **set-face-underline-p** can be used to specify whether a particular face should be underlined. In this case I turn off underlining for the faces bold and underline. (A non-underlined underline face? Hey, this is Emacs. Anything is possible.)

In order to use all of these wonderful faces, you'll need to turn on **font-lock-mode**. You may also wish to enable **transient-mark-mode**, which causes the current region (text between point and mark) to be highlighted using the region face. The following commands will enable this.

```
  (transient-mark-mode 1)
  (font-lock-mode 1)
```

Using hooks

One problem with the above configuration is that **font-lock-mode** is not automatically enabled for each major mode. Including the **font-lock-mode** command in your **startup.el** file will enable this mode when Emacs first begins, but not for each new buffer that you create. In general, this is true for any Emacs minor mode; you must turn on the minor modes whenever you enter a new major mode.

What we want to do is have the **font-lock-mode** command be executed whenever we enter certain modes, such as C mode or Emacs LISP mode. Happily, Emacs provides “hooks” which allow you to execute functions when certain events occur.

Let's turn on several minor modes whenever we enter C mode, Emacs LISP mode, or Text mode:

```
(defun my-enable-minor-modes ()
   "Enables several minor modes."
   (interactive "")
   (transient-mark-mode 1)
   (font-lock-mode 1))
(add-hook 'c-mode-hook 'my-enable-minor-modes)
(add-hook 'emacs-lisp-mode-hook 'my-enable-minor-modes)
(add-hook 'text-mode-hook 'my-enable-minor-modes)
```

Now you should find that upon entering any of these major modes, the corresponding minor modes are enabled as well. In general, each major mode has an entry hook, named *modename*-**hook**.

Local key bindings

Instead of using **global-set-key** to define key bindings for all major modes, we can use **define-key** to set bindings for particular modes. In this way we can specify the behavior of certain keys based on the major mode that you happen to be in.

For example, I prefer that the return key indent the next line of text relative to the one above it (if the previous line is indented five spaces, the next line should be as well). To enable this feature in C mode, Emacs LISP mode, and Indented Text Mode, use the commands:

```
(define-key indented-text-mode-map "\C-m" 'newline-and-indent)
(define-key emacs-lisp-mode-map "\C-m" 'newline-and-indent)
(define-key c-mode-map "\C-m" 'newline-and-indent)
```

Each mode has a *modename*-**map** associated with it, which specifies the key bindings for that mode. As you may have guessed, **newline-and-indent** is the Emacs function which does a newline followed by a relative indent.

When editing files, Emacs usually determines the mode to use from the filename extension. If I were to edit a new file called **clunker.c**, C mode would be used as the default. However, when unable to make a decision, Emacs uses Fundamental mode. I prefer to use Indented Text mode instead, which is enabled with the command:

```
(setq default-major-mode 'indented-text-mode)
```

Local key bindings can be used for more interesting tasks than those demonstrated above. For example, the command **M-x compile** will issue the command make **-k** (by default) in the current directory, thereby compiling whatever code you may have been working on. Output and error messages from the make command are displayed in a separate window. You can select error messages from the compilation buffer, in which case Emacs will automatically open up the corresponding source file and jump to the line containing the error. In all, this makes editing, compiling, and debugging programs much more efficient; you can do literally everything within Emacs.

In order to automate the process of saving the current source file and issuing make, we can bind the key sequence **C-c C-c** to a new function; let's call it **my-save-and-compile**. The code looks like this:

```
(defun my-save-and-compile ()
   "Save current buffer and issue compile."
   (interactive "")
   (save-buffer 0)
   (compile "make -k"))
(define-key c-mode-map "\C-c\C-c" 'my-save-and-compile)
```

The **save-buffer** command is used to save the current source file, and compile is issued with the command make **-k**. Now, with two simple keystokes you can fire your source file off to the compiler, and wait for the error messages to roll in. Without the **my-save-and-compile** function, you have to save the source file (using **C-x C-s**) and issue **M-x compile** by hand.

Of course, to use this feature you'll have to create a Makefile in the directory where your source files live. (The **make** command is issued in the directory containing the source file in question.) Creating Makefiles is another issue altogether. A future issue of *Linux Journal* will discuss this subject, but in the meantime there are several sources of information about make. The book Managing Projects with make from O'Reilly and Associates is a good place to start, as is the GNU make Manual, which covers the version of make available on Linux systems.

Also note that compile will first prompt you to save any modified buffers. If you modify only one buffer at a time, **my-save-and-compile** saves it for you. We could have **my-save-and-compile** save all modified buffers, but you may not want that to happen automatically behind your back.

As we mentioned, the **compile** function will open a new window containing messages from the make command. From this buffer, you can select error messages for Emacs to automatically jump to, allowing you to fix the problem and move on. If you are running Emacs under X, clicking on an error message with the second mouse button will take you to the line containing the error. Otherwise, you can move point to the error message (in the compilation buffer), and use **C-c C-c** (don't be confused by the multiple meanings of this key sequence). Alternately, you can use the function **next-error** to visit the next error message. In my **startup.el** I have this function bound to **M-n** in C mode (which, by now, you should know how to do).

Miscellaneous Customizations

There are several small items that you might want to configure to depart from Emacs' default behavior. I'm going to list the code for these briefly below; none of them involve new concepts other than those discussed above. The comments should describe these customizations adequately.

```
;; Allow M-j, M-k, M-h, M-l to move cursor,
;; similar to vi.
(global-set-key "\M-j" 'next-line)
(global-set-key "\M-k" 'previous-line)
(global-set-key "\M-h" 'backward-char)
(global-set-key "\M-l" 'forward-char)
;; Commonly used buffer commands, requiring
;; less use of CTRL
;; (For the ergonomically-minded.)
(global-set-key "\C-xf" 'find-file)
(global-set-key "\C-xs" 'save-buffer)
;; Open a line below the current one; as in "o" in vi
(defun my-open-line ()
  (interactive "*")
  (end-of-line nil)
  (insert ?\n))
(global-set-key "\C-o" 'my-open-line)
;; Make the current buffer the only visible one,
;; and recenter it.
(defun my-recenter-frame ()
   (interactive "")
   (delete-other-windows)
   (recenter))
(global-set-key "\C-l 'my-recenter-frame)
;; Save all buffers and kill Emacs, without prompting
(defun my-save-buffers-kill-emacs (arg)
   (interactive "P")
   (save-buffers-kill-emacs t))
(global-set-key "\C-x\C-c" 'my-save-buffers-kill-emacs)
;; Preserve original save-buffers-kill-emacs,
;; in case we don't want
;; to save what we were doing
(global-set-key "\C-x\C-x" 'save-buffers-kill-emacs)
;; Real Programmers don't use backup files
(setq make-backup-files 'nil)
;; But Real Programmers do use RCS. Includes
;; rcsid[] definition in a C source file
(defun my-c-insert-rcsid ()
   (interactive "*")
   (insert "static char rcsid[] = \"@(#)$Header$\";"))
(define-key c-mode-map "\C-c\C-x" 'my-c-insert-rcsid)
;; Finally, prevent next-line command from adding
;; newlines at the
;; end of the document. Instead, ring the bell when
;; at the end of
;; the buffer.
(setq next-line-add-newlines 'nil)
```

I hope that this whirlwind tour through the world of Emacs customization has been useful, or, at least entertaining. I've found many of the above modifications to be invaluable. Remember the old saying: Have Elisp, will travel.

That being said, it's back to **vi** for a while.

Getting the Emacs LISP Manual

Matt Welsh (mdw@sunsite.unc.edu) is a programmer at the Cornell University Robotics and Vision Laboratory. He spends his free time homebrewing virtual beer and playing the blues.

Archive Index Issue Table of Contents

Advanced search

Interview: James MacLean

Michael K. Johnson

Issue #5, September 1994

In this interview, *Linux Journal* talks with James Maclean, of Nova Scotia, Canada, who is the current leader of the DOSEMU team.

Linux Journal: Tell us a little bit about DOSEMU...

James: The DOS Emulator for Linux started a few years ago when Matthias Lautner created an application for Linux that could load DOS from a diskette and then accomplish some rudimentary DOS commands. At that time I could remember getting one editor to run sometimes and other DOS programs to give signs of starting but crashing the application. All in all, very impressive to me.

Then Robert Sanders jumped in, and in what seemed to be a very short time, had coordinated and expended enough effort to make DOSEMU begin supporting much of the necessary low-level emulation that DOS programs expected.

Since then a group of about a dozen of us have taken on the challenge of furthering DOSEMU's DOS compatibilty, and increasing its performance. To date, we can run quite a gauntlet of DOS applications ranging from wordprocessors to database packages, from graphics packages (now including Windows 3.0 to a point) to even some games (albeit slowly).

Linux Journal: Tell us a little bit about yourself...

James: My first exposure to computers was around grade 9 when my folks bought a Texas Instruments 99/4a. I was quickly impressed by how my disorganized mind could write BASIC code that looked quite organized to the end user. So naturally I started creating games, and actually sold a few. My most memorable one being 'Oil Suckers'. Of course I knew then what I wanted in a career :-). In time I learned how to program that computer from the machine code up, and always dreamed of creating my own multi-tasking operating system after seeing how much of the computer's time was being wasted. (So naturally I envy Linus, and I hope he doesn't mind that there are those of us that live vicariously through him :-).)

Then came university, where I actually lost part of my ambition to go on with computers. Luckily in the year I graduated I started a job as an opertator for the provincial government's data center which was an IBM 3090 MVS/JES3 shop. From being amazed at the power of the mainframe, I started again quickly learning as much as I could about the system I had to make use of. It was due to this interest and growing knowledge base that I soon was given the nickname JES.

Then my job moved here to the Department of Education where I am a one-man team looking after in-house matters for our division. This was my first time to actually use a PC, and again I found it limiting to only be capable of running one program at a time (DOS). Later on our M.I.S. coordinator started a pilot project to see if the Internet would be of any benefit to the department. Luckily at the time I mentioned my interest in getting back to Unix, which was what we had in University, and he mentioned that Byte magazine had the ftp location of a free Unix-like OS called Linux. From there to here has been a fun roller coaster ride of running a powerful OS, being a coordinator of one project for it, and learning more than enough new stuff everyday.

Linux Journal: What got you interested in DOSEMU?

James: Once I had Linux up, I realized that our department (and myself) used many DOS-based programs. I had been running OS/2 2.0 on a 4MB system, and felt some of my DOS applications should be able to run under Linux. I heard about this DOSEMU app that was suposedly fairly primitive, but could do some things. I guess after I ran it, I realized how much I didn't know about Unix, and decided to hover around listening to what progress was made, if any, to DOSEMU.

Then Robert came along, and suddenly many of my DOS applications were either running, or almost running. This drove me to start tinkering with the code, passing some of my changes along to Robert, and finally coordinating the project.

Linux Journal: What is your technical background?

James: Well, I have a BSc. with a major in Computer Science, but I feel my real background comes from the experiences I have had; from programming the TI, to automating procedures for the IBM 3090, to taking long, hard looks at the internals of both DOS and PC's. I've programed in a multitude of languages (haven't we all :-)), and seem to be getting a good grasp of Unix from the inside out. I'm capable of discussing TCP/IP and Novell at the administrator level, and more and more on the technical level, but have yet to program at the network level, unless you count some small scripts in PERL.

Linux Journal: What interests you most about working with DOSEMU?

James: I think what interests me most about DOSEMU is the same thing that interests me most about Linux. The more I do with it, the more I realize what it can do. The source code is there, and the only limit seems to be time and imagination. I get a very rewarding feeling when something is added to DOSEMU that works and helps others out. Beyond any other OS to date, there really seems to be a sense of pride in running Linux, and for me, being a part of the DOSEMU team.

Linux Journal: Please tell us about some technical challenges that DOSEMU has presented, and how you solved them.

James: Technical challenges we've had many of, but the ones I can take credit for solving are few. One good example is our ability now to run graphics programs from within DOS on hardware that DOSEMU will support. We first set out to create our own routines to deal with saving and restoring all the necessary info for changing the video modes. Then Robert Sanders caught on to just calling the actual video bios already installed for each card, and using it. This little change made DOSEMU become much more functional, and no one had to code mode changes for each card.

We've also been attempting to create solid serial support for which I had a program that wouldn't even run correctly under OS/2's DOS. But with Mark Rejhon diligently scanning through kilobytes of my debug output, he was able to make this program run, as well as many others.

There was also the lack of support for FCB handling in the redirector. This was of course not documented anywhere, but I lucked into it when I decided to get a copy of Undocumented DOS 2. It had enough info to help me kludge in the additional code that allowed FCB using programs to run.

For me the most pleasurable one to fix was the stack-wrapping problem with many programs that start their stack at 0x0000 and of course wrap backwards (0xfffe). We always had many programs that didn't run, and I could never see anything in the debug listing to explain why. Then I got a small piece of code from a person who wondered why something would run only if compiled certain ways. With the code in hand, I started looking at it through debug and noticed that the program seemed to be running in reverse. Very strange, so I asked around and got some good ideas, but none panned out. Then when I was down to half as much hair as I used to have, I noticed that the stack didn't contain what was suposedly pushed on it. Suddenly it came to me, 0 - 2 <> 0xfffe when a linear address was made from the seg:off pair. Instead of wrapping at the offset, it was just subtracting two from the linear address. This little extra code enhancement started a multitude of programs running, none the least of which for me was Netware's lsl. Linus has made a much better set of functions for handling this scenario, and added them into the vm86 function of the kernel for us to use.

One of the most noticeable changes that has occurred lately is due to Lutz Molgedey. After some discussion had appeared about catching the signals DOSEMU uses at a lower (kernel) level, Lutz asked if it would be a worth-while task. I know quite honestly that I said something to the effect that I wasn't convinced about the speed improvement being that great, but if he wanted to try, go ahead and see what he comes up with. Shortly after, and very much to my surprise, he had made the necessary first layer of kernel changes, and DOSEMU started to run like a scalded cat. The increase in speed was enormous, and we all felt that it was worthwhile to give us a DOSEMU we would work with.

Linux Journal: What parts of DOSEMU do you like the most?

James: I find the redirector interesting because I can dynamically redirect anything Linux has for directories. The ability to connect to our Novell Server and also use Netware Lite has also made it much more functional for me. I would say though, from looking after the project, I am always impressed by the input I get from all over the earth, from ideas to patches, and the true enjoyment from sitting down every second Saturday on the #Ldosemu IRC channel to talk with folks that I share common interests with, no matter where in the world they harbour.

Linux Journal: What parts would you like to change?

James: I think we'd all agree the code still needs a bit of cleaning. The installation for the new user is still too raw, and should have possibly more documentation, or a front end to ask step-by-step installation questions. I'm hoping to see even better networking support like TCP/IP from within DOS using network cards known to Linux.

Linux Journal: Anything else you'd like to say?

James: I'd really like to say thank you to all those hackers that make up our DOSEMU team, as well as to everyone that has sent in patches, or just given suggestions along the way.

James B. MacLean can be reached via e-mail at jmaclean@fox.nstn.ns.ca.

ez—for the Programmer

Terry Gliedt

Issue #5, September 1994

In this article, Terry Gliedt continues his tour through the Andrew project and gives us a taste of another aspect of it. Here's the ez editor again, but this time as a source view editor for the programmer in us all.

AUIS had its roots in 1982 when Carnegie Mellon University and the IBM Corporation decided to jointly develop a campus computing facility based on personal computers to replace the time-sharing system then on campus. IBM provided not only generous funding, but also some talented individuals and access to IBM development programs.

The result was a graphical user interface we know as the Andrew User Interface System and a file system, the *Andrew File System*. The file system formed the basis of Transarc Corporation's *Distributed Computing Environment* (DCE) and is offered as part of the Open System Foundation software.

The *Andrew Consortium*, composed of a number of corporations and universities, funds the current development of AUIS. AUIS is available on a wide variety of platforms including Linux, AIX, Solaris, Ultrix, HP UX as well as others.

In early June, version 6.3 of the Andrew User Interface System (AUIS) was released by the Andrew Consortium. This led to the release of **auis63L0-wp.tgz** which contains just a small portion of AUIS that is suitable as a word processor.

Now another package has been released, **auis63L0-src.tgz**, to *sunsite.unc.edu* in */pub/Linux/X11/andrew*. This is a set of additions to the word processing package to support the programmer. This article will describe the support ez provides to ease the coding and analysis of our programs. Future articles in *Linux Journal* will describe other pieces of AUIS.

ez Views

As a programmer, we use the same simple tool, *ez*, as we did for word processing. Our source program is loaded (or created) and displayed in a window. But unlike *ez* the word processor, the data in the program source (document) does not directly control what happens. Rather the editor, *ez*, actually is “aware” of the syntax of the source language that is being edited and provides a specialized *view* of the source.

In the same way that a graph is really just a different representation of an array of numbers, *ez* has the ability to provide unique views of program source. This allows the editor to help you when entering the program source. *ez* also provides assistance in compiling your program. Finally, *ez* can help you analyze your program, to assist in understanding where data is referenced or where routines are defined.

ez provides support for a number of languages including assembler, Pascal, Modula, Lisp, C, and C++. It provides a unique view for each language source file. The view is triggered by the extension of the file you edit. For instance, ez will treat the file test.c as a C program and will treat the file test.C as a C++ program.

Entering Your Source Code

Figure 1. -fg black -bg lavenderblush

When you first edit a C program, you will see something like Figure 1. You see immediately there are many things to make the program easier to read:

- the include statements are in a fixed font
- comments are in a blue italized font
- special words in the language like char, int, while, etc. are in magenta. (These are words already recognized by the ez view.)
- function names, like main, are in red, using a different font
- other special words, like, exit, are in green. (These are words the user defines as special.)
- strings are shown in a fixed font

Figure 2. Comment in progress

Figure 3. C-clause in progress

As you enter your code, ez will react as soon as it figures out what you are coding. In Figure 2 you can see a comment being formed. Notice the color and font have already been applied, even though the comment is not closed. In Figure 3 you can see how *ez* handles pair delimiters like (), {}, and []. When you enter the trailing delimiters (e.g.,"}") then ez will reverse-video the entire region to the matching delimiter. As you continue typing, the reverse-video is removed.

ez will attempt to keep its view correct at all times, but you can confuse it. For instance you might code a string without the comment delimiters and then later add the /* */ operators. When this happens, you only need select the menu item Redo styles on the Source Text menu card. This will cause the entire screen to be redrawn applying the view.

<u>Figure 4. Reformatting your C source</u>

ez will also attempt to help as you enter text by automatically indenting your source lines as you enter them. You do not need to enter a tab to indent, since *ez* will do this for you automatically. Unlike other aspects of the view, this does modify the source since it inserts tabs to align the text. You can also ask *ez* to reformat your source at any time. Simply select the area you want to reformat (i.e., so it is shown in reverse-video) and select the menu item Format line/ region on the Source Text menu card. You can see the result of this action on our original source program in Figure 4. One convenient feature is that when you enter a trailing delimiter (like a "}"), *ez* will reverse-video the region to the matching delimiter ("{"). If you press tab at this point, ez will reformat the selected region, saving you the need to later re-select the area with the mouse.

There is a great deal of tailoring that can be done to control the behavior of your source view. If you do not like the way *ez* indents your source code, simply avoid the use of the *Format line/region* menu item and your source will not be modified. Regardless how your source looks, the lines of your original source are not modified unless you select this menu item.

<u>Figure 5. Source from Fig. 4 in wider window</u>

Just as with a text document, *ez* will automatically wrap lines based on the width of the window. For instance, Figure 5 shows the same source as in Figure 4. Notice the comments and source lines are re-aligned to make it more readable.

In the past I would spend a great deal of energy making my source look "pretty". It's important to me that my source look consistent since it makes the code much easier to read. Since I have been using *ez*, I no longer make any effort to format code, but rather concentrate on entering it. When I'm done, I reformat it with *ez* and I'm done. I especially like the reflowing of comments.

This has caused me to change my commenting style. I seldom put comments on the same line as code any more. Rather I rely on paragraphs at the beginning of a block of code to explain what will follow. I much prefer this approach in coding. Others may not like this style. I use the parts of *ez* that I find useful, and ignore the others. I encourage you to do the same.

Compiling Your Code

ez has support to assist you in finding compile errors. This is done by invoking make, capturing the output and then asking ez to show you the source lines in error. This is done by creating a second window of the source (select *New Window* on the *Window menu card*). In either of the source windows, select *Start Compile* on the *Make menu card*. You can see the results in Figure 6. When the compile completes and there are errors, select Next Error on the Make menu card. ez will then switch files to the offending file (if necessary) and show the line identified in error by the compiler. Correct the errors and start the process over again.

Figure 6a. Finding compile errors

Figure 6b.Compile errors

Code Analysis

Sometimes just having the editor format the source helps to understand the code. The first time I used ez on my C source, I immediately noticed that some code was shown as comments. “Ha! A bug in ez”, I thought. Well, there was a bug, but not in ez. I discovered I had missed an ending comment delimiter in a program that I thought was working. I actually had 10 lines of code commented out. That was enough to convince me of the value of views in ez.

Figure 7. Compressing lines in source program

ez is not a complete source browser in the sense of some commercial applications, but rather it provides a set of functions to help you to navigate through your source. Sometimes you don't want to see some parts of your program. Highlight that part of your program with the mouse and select Compress region on the Source Text menu card and you will see something like that shown in Figure 7. If you click in the “compressed lines” box, the code will be shown again. Selecting Decompress all on the Source Text menu card will expand all compressed source areas.

The program ctags is a program found on most Unix systems. It will generate a file called tags which contains cross-reference information on the functions within a set of source files. You can also get cross-reference information on variables by issuing the command *"ctags -v *.h *.c"*. ez can use this file to show you where functions or variables are defined. Load the tags file by selecting the menu item Load New Tag File or Rebuild Tag File on the Tags menu card. To find where a variable or function is defined, select the variable with your mouse and select Find Tag on the Tags menu card. ez will switch to the file where this is defined and show you the line where it is defined.

Templates

So far we have been talking about editing existing source files. ez provides additional support when you edit a new, non-existent source file, by providing a means for you to control a "template" or default file. Each language has its own template file which can be found in /usr/andrew/lib/tpls. You can tailor your own by creating your own template directory and telling AUIS applications where your templates are by defining the

TEMPLATEPATH environment variable. To set this up, issue the following commands:

```
mkdir $HOME/tpls
cp /usr/andrew/lib/tpls/c.tpl $HOME/tpls
export TEMPLATEPATH=${HOME}/tpls:/usr/andrew/lib/tpls   # For bash/ksh
setenv TEMPLATEPATH ${HOME}/tpls:/usr/andrew/lib/tpls   # For
csh/tcsh
```

Figure 8. Template for C language source file

Now you can edit your own $HOME/tpls/c.tpl template and tailor this to your needs. Figure 8 shows you one such template. What you will see is much like a conventional program source, and indeed, it is. This is where you put your own "boiler-plate"; code you'd want in most any program. More interesting, however, is the existence of **dogtags** (what's in a name?). These are fields, delimited with "<@" and "@>", which are automatically substituted when the template is loaded. For instance in Figure 8, you see <@name@>, which is replaced with the name of the file being edited (fig1.c). Most of these are pretty obvious. For a complete list of all dogtags, see the help text shown by the command auishelp dogtags.

Controlling Colors and Fonts

This template file also controls the colors and fonts you first saw in Figure 1. While editing the template file, select Edit Styles on the File menu card. This opens a second window used to define the attributes of data within this template. Start by selecting the <No Menu> field in the upper left-hand corner. To the right you will see keywords which correspond to aspects of the language (comment, function, etc.). Select one of these and other attributes like the font and size, color, spacing, etc., will be shown. Using these you can change any attribute you'd like. This change will not become effective until you save the template and then edit a new source file. It does not take effect for files which you are already editing.

Printing and Previewing

Just as with the text documents, ez will print the source program as you see it on the screen, including the fonts. Regardless of the colors you display your source with, printing uses a black foreground on a white background (as you'd expect). The preview process uses ghost-view to display the PostScript document that is generated.

For More Information

A mailing list is available at info-andrew@andrew.cmu.edu (mail to info-andrew-request@andrew.cmu.edu for subscriptions). The newsgroup comp.soft-sys.andrew is dedicated to the discussion of AUIS. A World Wide Web home page can be found at http://www.cs.cmu.edu:8001:/afs/cs.cmu.edu/project/ atk-ftp/web/andrew-home.html. A book, Multimedia Application Development with the Andrew Toolkit, has been published by Prentice-Hall (ISBN 0-13-036633-1). An excellent tutorial is available from the Consortium by sending mail to info-andrew-request@andrew.cmu.edu and asking about the manual, A User's Guide to AUIS.

Terry Gliedt (tpg@mr.net) left Big Blue last year after spending over twenty years with IBM. Although he has worked with Un*x and AUIS for over six years, he is a relative newcomer to Linux. Terry does contract programming, teaches classes in C/C++ and Unix and writes the occasional technical document.

Linux in the Trenches

Ph.D.. G.W. Wettstein,

Issue #5, September 1994

One of the first people to see a commercial use for Linux was Dr. Greg Wettstein at the Roger Maris Cancer Center. A few months after Linux was first announced, he saw the potential for an operating system capable of running a large patient care system, and has seen his vision vidicated so far. This is his story.

The Roger Maris Cancer Center; Depending on Linux

I still remember the lone dollar prompt springing to life on the monitor of a lowly 80386sx in my office as it booted Linux for the first time. The appearance of that single dollar sign has proven to be possibly one of the most seminal moments in the history of the Roger Maris Cancer Center, and of my professional career. The birth of this free operating system in an environment previously dominated by commercial software was to have profound ramifications on both the quality and efficiency of the care our staff provides to our patients.

I had watched with interest the early announcements Linus made on the networks about a Unix clone he was writing called Linux. Consumed by work and development pressures, I did not allow myself the indulgence of playing around with the new offering for fear of engaging in yet another eternal timesink. Eventually, overcome by curiosity, I decided to take the plunge and investigate Linux. Our experiences have far exceeded our expectations; Linux certainly is a wonderful timesink...

The actual incentive and decision to move toward an free/open systems approach to our information processing needs occurred at a much earlier time. I had been recruited in 1988 to serve on a team whose goal was to develop a comprehensive cancer care facility. The only perk I asked for was an 80386-based computer, since my instincts told me that the advent of this microprocessor into the personal computer market was going to revolutionize desktop computing.

Our medical director, Dr. Paul Etzell, was intrigued by my request and wanted to know why I thought this. I told him, “Because it can multi-task and handle large amounts of memory the way it is supposed to.'' He was excited and told me that he had always dreamed of a specialized patient information system for his cancer center. Armed with this mandate, a 16Mhz 80386sx ALR, and a Xenix 2.3.3 development system we set out to design new solutions to problems in the delivery of outpatient cancer care. We deliver care that allows patients to have their treatment and still conduct a somewhat 'normal' lifestyle.

That was in 1988. The Roger Maris Cancer Center has now grown to be a major regional treatment and referral center. Medical care, chemotherapy, and radiation therapy are provided to approximately 125 patients with a diagnosis of either cancer or hematologic (blood) diseases each day. A total of 13 physicians deliver cancer care supported by a staff of approximately 100 people and 25 Linux workstations. The Cancer Center is a distinct service line of the MeritCare Medical Group which consists of approximately 250 physicians and 2000+ employees supporting one central clinic, a 400 bed hospital and 25 regional clinics. For everyone with aspirations for Linux in the commercial environment, we are about as commercial as things get.

When the first Linux dollar prompt appeared, our Cancer Center was not naive about free software. I had already ported a number of the GNU tools to Xenix in an attempt to produce a suitable development environment. By this time our Cancer Center was supported by two 33Mhz 80386dx's (Gateway-2000's), the ALR having been mercifully put to rest.

However, compared to compilers, editors and utilities, an operating system is a completely different beast. As I sat staring at the screen contemplating the fact that a machine that had been previously sentenced to MS-DOS was now quietly multi-tasking, I couldn't even believe what was in front of me. Could something that I had ftp'ed for free from some distant spot in Finland really be capable of playing a role in our development plans?

At our morning coffee I looked across the table at Dr. Etzell and quietly commented that I might have run across something that could have a profound impact on the Cancer Center.

Linux has had that profound impact because of the doors which it has opened for us. Our medical director, a true visionary, knew that the face of medicine would be radically changing. True to his prediction, the medical community in the United States is facing extreme pressure to deliver high-quality, cost-effective care. If our Cancer Center was to survive, let alone thrive in this environment it would be critical to innovate and adapt quickly. The reason that Linux has had such a profound impact is that it has provided us with the tools necessary for this evolutionary process.

It probably goes without saying that a decentralized system of peer-to-peer networked workstations was not the same vision that corporate data processing possessed for us. Based exclusively on centralized mainframe processing, the corporate wheels were turning toward network solutions, but a network whose main purpose was to deliver datastreams from mainframes to diskless workstations; expensive terminals connected by expensive wiring. Our vision was of independent workstations capable of functioning without network connections if need be, but also capable of utilizing networking to enjoy extreme synergism in their daily tasks.

If this was to be our gospel then it was up to Linux to preach it for us. The dichotomy in vision ensured that there would be no funds available for our work. Linux and free software provided the tools to implement our strategy in the absence of official support. With Xenix and no money for upgrades there was no potential for X-windows, and most importantly no money for the TCP/IP networking which was central to our development strategy.

This is not to say that Linux was a magical solution that revolutionized our method of patient care overnight. The first dollar prompt was actually from a bootdisk/rootdisk combination (I am dating myself here) that was version 0.95a (one of those really weird version numbers) in early 1992. At that point in its development Linux was far from the workstation contender that it now is. I developed application software furiously on the Xenix boxes while I nursed a stable Linux environment into existence. The bulk of our application code was in PERL/TeX/C which I knew would be available in only a matter of time on Linux.

All the development on Linux was done on a 20Mhz 80386sx (Gateway-2000) with a 120MB disk drive. This machine was tied back-to-back through serial ports with one of the Xenix machines whose 300MB hard drive provided a safe haven for the development sources. It is a true testimony to Linus and the other developers that we never experienced a filesystem crash or lost data during the development process. In fact, the original 20Mhz machine is still in service, still using the original Minix root filesystem written to its hard drive with 0.96a.

It wasn't until about 0.96c that Linux actually began to shoulder operational responsibilities. This was mainly due to the fact that 0.96c was the first release whose serial drivers would reliably withstand a UUCICO session. Our concept of 'networking' at that point was file relay and remote execution via UUX, making reliable serial communications essential.

During the 0.95-0.96 stages our development efforts were focused on putting together what would be called a distribution in today's parlance. Reality dictated that a reliable means of replicating and updating Linux would be required if multiple machines were to be committed to service. During this time I wrote the first version of StopAlop [StopAlop stands for ”Stop Alopocea“ (a medical term for hair loss) or ”Stop A lot of problems“]. StopAlop allowed us to package Linux into a series of modules which could be installed, verified and updated independently, reliably, and with version control. Our installations centered on a 'base' module which was everything needed for a standalone UUCP-capable Unix workstation. Additional modules provided our appli-cation code, emacs, a development environment, text processing (TeX), and ultimately networking and X11.

When reliable kernels arrived we were prepared with a well-tested, reliable system environment. Our application software was already maturing and in operation under Xenix. As we began deploying Linux, our medical director began negotiations which ultimately provided the hardware resources necessary to formulate our peer-to-peer distributed computing environment. No small part of his efforts were focused on negotiating for corporate survival of our fledgling project.

His efforts were enhanced by the profound impact that Linux began to have on our operations. No longer encumbered by the restriction of two licensed copies of an operating system, we were able to immediately deploy four additional workstations. Our application software hosted on these machines provided patient information, patient tracking, drug history databases, pharmacy support and automated patient charting to patient care environments which were largely non-automated.

Our most powerful argument to detractors (who were vocal) was that the systems being put into place were functioning well, were tuned to our application needs, were substantially increasing productivity, and had cost nothing over and above the hardware costs. The most astounding point made was that the workstations accomplishing these feats had been rescued from the fate of being diskless DOS network clients...

From this point we have progressed, passing some notable milestones. Six months after the first Linux machine began tracking patients, a major user-interface move was made by changing from using multiple virtual consoles to X. Serial connections were established between the Prime computer, which hosted laboratory information, and the Linux workstations. Receptionists and staff members who sometimes had three different computers and/or terminals to contend with were now interacting with different information sources via multiple xterms on the same display.

The most significant milestone occurred the day the concentrators were plugged in connecting all the workstations via 10baseT Ethernet. With high-speed TCP/IP connections between the workstations, our goal and vision of a peer-to-peer distributed information support environment was complete. The gospel according to Linux was named Perceptions; a name chosen to personify the design goal of our adventure, to provide a unified resource which would enable staff members at the Roger Maris Cancer Center to have a clear information picture of the patients they were taking care of. In December of 1993 the 'big red switch' was pulled on the last Xenix machine and our Cancer Center and its operations became totally dependent on Linux.

The story of Linux and Perceptions is the truest example of why I feel that 'free' software can play an important role in the commercial marketplace. The speed at which we were able to accomplish our goal is a direct result of our ability to innovate and respond to environmental needs. The impediments to our accomplishments were reduced to only what we were individually capable of developing and supporting. We called the shots, we made the decisions, if something didn't work we fixed it. It was these experiences that caused us to form the motto: I would rather spend 10 hours reading someone else's source code than 10 minutes listening to Musak waiting for technical support which isn't.''

The real basis of our success is, in part, the large, diffuse network of Linux Activists who made what we did possible. It is always easy to make good design decisions when you have good data to make those decisisons with. I read literally thousands of Usenet articles to make sure that I knew everything there was to know about the stability of kernels, what software worked and what didn't, and most importantly, how to fix what didn't work. On the basis of the experiences of hundreds or thousands of other people I made good design decisions. Good design decisions lead to successful implementations, which was really the bottom line of our survival. As Dr. Etzell so aptly put it, ”It's always difficult to argue with something that works.''

The story of Linux and Perceptions is far from over. Only the basics of our total design plan have been implemented. One of the fundamental design tenets of our information system is parallel database concurrency across multiple hosts. Our systems are designed to be capable of withstanding (and have withstood) severing of network connections without end-users knowing the event happened. Additional work with locking and updates across a wide-area network are a current research interest. An extreme area of interest and activity is digital document storage and retrieval. We are secure in meeting future challenges in health care with the knowledge that we have a stable open operating environment on which to formulate our information strategies.

So the story of Linux at the Roger Maris Cancer Center is really the story and testimony of the success of Linus Torvalds and the Linux Activist movement. I would hope that everyone in the movement shares the same sense of accomplishment that I do. It is with extreme pleasure that I am able to tell visitors that we take better care of cancer patients because of an experiment in protected mode programming conducted by a (then) 23-year-old computer science student from Finland.

1. StopAlop stands for “Stop Alopocea” (a medical term for hair loss) or “Stop A lot of problems”.

Greg Wettstein (greg@plains.nodak.edu) is a pharmacist who chose the profession because it required only two quarters of math. He is now a PhD in quantum chemistry and theoretical drug designer who won't leave the Upper Midwest because he likes team roping and raising cattle more than wavefunctions and the Hansch equation, and who spends 'spare time' at his best friend's construction company driving payloaders and reel trucks (into holes).

He can be reached as: G.W. Wettstein, Ph.D., Oncology Research Division Computing Facility, Roger Maris Cancer Center, Fargo, ND 58122.

Linux on the Motorola 680x0

Hamish Macdonald

Issue #5, September 1994

Most people think that Linux runs only on Intel 80386 and above processors. This article is a status report on Linux/68k, the port of Linux to Motorola 680x0-based systems.

Linux has been ported to 680x0 based machines. Amigas, Ataris and Macintoshes with the appropriate hardware support are the intended platforms. Only the 68020 (with 68851 MMU), 68030 and 68040 processors are currently supported. (A Memory Management Unit is required.) Versions of Linux/68k now run on various models of the Amiga and the Atari. The Amiga and Atari versions are not yet merged, but this will be completed shortly. The Atari version is based on the Amiga version.

Almost all of the x86-dependent code has been ported to run on the m680x0, including context switching, memory management, signals, “ptrace” support, “core” files and the “/proc” filesystem. The kernel supports Unix domain sockets, but does not yet support TCP/IP.

The Linux/68k kernel supports the Minix filesystem, the Linux ext2 filesystem, the ISO filesystem, and the Amiga Fast File System.

Patch 3 of version 0.08 of the Linux/68k kernel was released in May, 1994. This version is compatible with Linux/PC 0.99pl14. It provides support for a ram disk, the Amiga floppy drives, three popular Amiga SCSI controllers, and the IDE controller in the Amiga 4000. This version has a fast “console” full-screen VT100 emulation driver supporting various video modes. This console driver doesn't yet support multiple virtual consoles. There are drivers for the Amiga mouse and the Amiga parallel port. There are no serial drivers for the Amiga yet. This version is quite stable; it can be considered beta quality.

The Atari port includes the machine-independent support listed above, plus support for the Atari floppy drives, mouse, joystick, SCSI controller and the Atari Falcon IDE controller.

There is a “680x0” channel in the linux-activists mailing list which is used for some of the communications between the developers and other interested parties. Discussion also takes place on the “comp.unix.amiga” newsgroup on Usenet.

Just as this issue of *Linux Journal* went to the printer in late July, version 0.9 of Linux/68k was released. This version is derived from and equivalent to the Linux/PC kernel version 1.0.9.

In the longer term, it is hoped that the changes for the 680x0 support can be somehow folded back into the main Linux sources. In making these changes, care was taken to abstract out processor and platform dependencies. Hopefully this process will accelerate the effort that Linus Torvalds will be making to porting Linux to a Digital Equipment Corporation Alpha PC.

Hamish Macdonald Can be contacted at: (Hamish.Macdonald@bnr.ca)

Archive Index Issue Table of Contents

Advanced search

Dialog: An Introductory Tutorial

Jeff Tranter

Issue #5, September 1994

Linux is based on the Unix operating system, but also features a number of unique and useful kernel features and application programs that often go beyond what is available under Unix. One little-known gem is “dialog”, a utility for creating professional-looking dialog boxes from within shell scripts. This article presents a tutorial introduction to the dialog utility, and shows examples of how and where it can be used.

Linux is based on the Unix operating system, but also features a number of unique and useful kernel features and application programs that often go beyond what is available under Unix. One little-known gem is “dialog”, a utility for creating professional-looking dialog boxes from within shell scripts. This article presents a tutorial introduction to the dialog utility, and shows examples of how and where it can be used.

by Jeff Tranter

If you have installed a recent version of the Slackware Linux distribution, you've seen the professional-looking install process; it was created using the dialog utility.

True to the Unix tradition of writing general-purpose tools that work together, dialog allows creating text-based color dialog boxes from any shell script language. It supports eight types of dialogs:

- yes/no boxes
- menu boxes
- input boxes
- message boxes
- text boxes

- info boxes
- checklist boxes
- radiolist boxes

Dialog is very easy to use. If you've got your keyboard handy, here's a one-line example of a message box you can try. (Note: The examples in this article assume you are running a Bourne-compatible shell program such as GNU bash.)

```
% dialog --title 'Message' --msgbox 'Hello, world!' 5 20
```

This example creates a message box with the title “Message”, containing the greeting “Hello, world!”. The box is 5 lines high and 20 characters wide, with the message nicely centered in the box. An “OK” button appears at the bottom; pressing <enter> dismisses the menu.

Dialog Box Types

Most calls to dialog are in a similar format: an optional title, the dialog type, the text to be displayed, and the height and width (in characters) of the dialog box. Additional parameters specific to each menu type follow. Let's have a brief look at each of the available types.

The “yesno” menu is very similar to our first example:

```
% dialog --title "Message"  --yesno "Are you having\ fun?" 6 25
```

If you try this example, you will see that there are now two buttons at the bottom, labeled “Yes” and “No”. You can select between the buttons using the cursor keys (or <tab>) and make your selection by pressing <enter>. The exit status returned to the shell will be 0 if “Yes” is chosen and 1 if a “No” selection is made.

You may wish to try experimenting with the height and width parameters. If the width is less than the string length, the string is wrapped around (at word boundaries). If you make the dialog box too small, then characters will be lost.

We previously saw the message box. The “infobox” is similar except that it does not wait for the user to select an “OK” button. This is useful for displaying a message while an operation is going on. Here is an example:

```
% dialog --infobox "Please wait" 10 30 ; sleep 4
```

The “inputbox” allows a user to enter a string. The usual editing keys can be used, and the text field scrolls if necessary. After the user enters the data, it is written to standard error (or more commonly redirected to a file as in this example):

```
% dialog --inputbox "Enter your name:" 8 40 2>answer
```

The “textbox” type is a simple file viewer; it takes a filename as a parameter:

```
% dialog --textbox /etc/profile 22 70
```

The usual movement keys work here: the cursor keys, Page Up, Page Down, Home, etc. You can exit by pressing <esc> or <enter>.

The “menu” type allows creating a menu of choices from which the user can choose. The format is

```
% dialog --menu <text> <height> <width>
<menu-height> [<tag><item>]
```

Each menu entry consists of a “tag” string and an associated “item” string, both of which are displayed. The user can make a choice using the cursor keys and pressing <enter>. The selected tag is written to standard error. Here is a simple example:

```
% dialog --menu "Choose one:" 10 30 3 1 red 2 green\ 3 blue
```

The next type is the “checklist”. The user is presented with a list of choices and can toggle each one on or off individually using the space bar:

```
% dialog --checklist "Choose toppings:" 10 40 3 \
        1 Cheese on \
        2 "Tomato Sauce" on \
        3 Anchovies off
```

The third field in each choice is the initial state; -either “on” or “off”. The last type is the “radiolist”, essentially the same as the checklist except that the user must make one choice from a list of mutually exclusive options. The radiolist type, and the alternate form of title show here, were introduced in version 0.4 of dialog.

```
% dialog --backtitle "CPU Selection" \
  --radiolist "Select CPU type:" 10 40 4 \
        1 386SX off \
        2 386DX on \
        3 486SX off \
        4 486DX off
```

A Real Application

The preceding examples were somewhat unrealistic; dialog is normally used within a shell script to do some real work. Let's look at a simple but useful application. I use the following script to back up my home directory to floppy disk on a regular basis:

```
#!/bin/sh
# Backup all files under home directory to a single # floppy
# Display message with option to cancel
dialog --title "Backup" --msgbox "Time for backup \ of home directory. \
Insert formatted 3-1/2\" floppy and press <Enter> \ to start backup or \
<Esc> to cancel." 10 50
# Return status of non-zero indicates cancel
if [ "$?" != "0" ]
then
  dialog --title "Backup" --msgbox "Backup was \ canceled at your
  request." 10 50
else
  dialog --title "Backup" --infobox "Backup in \ process..." 10 50
  cd ~
  # Backup using tar; redirect any errors to a
  # temporary file
  # For multi-disk support, you can use the
  # -M option to tar
  tar -czf /dev/fd1 . >|/tmp/ERRORS$$ 2>&1
  # zero status indicates backup was successful
  if [ "$?" = "0" ]
    then
    dialog --title "Backup" --msgbox "Backup \
completed successfully." 10 50
    # Mark script with current date and time
    touch ~/.backup
  else
    # Backup failed, display error log
    dialog --title "Backup" --msgbox "Backup failed \ -- Press
<Enter>
    to see error log." 10 50
   dialog --title "Error Log" --textbox /tmp/ERRORS$$ 22 72
  fi
fi
rm -f /tmp/ERRORS$$
clear
```

To run this automatically, I put these lines in my .profile file to call the backup script on login if more than 3 days has elapsed since the last backup was made:

```
# do a backup if enough time has elapsed
find ~/.backup -mtime +3 -exec ~/.backup \;
```

A Longer Example

The sound driver for the Linux kernel uses a program called “configure” to prompt the user for sound configuration options. It generates a C header file based on the chosen options. A replacement based on dialog could offer some advantages, such as a more professional appearance and the ability to select options randomly from menus rather than as a linear sequence of questions.

Due to time and space constraints, I only present a partial (but functional) implementation of a sound driver configuration script. This could quite easily be extended to fully replace the current configure program.

The complete script is shown in as Listing 1. I'd like to explain it using a top down approach, which means reading the listing starting from the bottom.

The last part of the script is a while loop which simply calls the shell function main_menu repeatedly. Above that is the code to implement the main menu. We present the user with three choices, and redirect the selection to a file. One of three shell functions is then called, based on the user's choice.

The most important menu in this script is the next one, the config_menu function. Again we present the user with a number of choices. Note that in this case there is an option which returns the user back to the main menu.

Continuing to read our listing backwards, we come to the select_cards function. The kernel supports multiple sound cards, so here we use a checklist to present the user with the available choices. The command “on_off” is a utility function that will be shown later; it returns the string “on” if its parameters are equal, otherwise it returns “off”. This is the form that the checklist menu requires. Note that the return status of the command is checked. If the user selects “cancel” from the menu then the return status is non-zero and we return immediately without making any changes. Otherwise, we set appropriate variables to indicate which sound cards have been enabled.

The next function, as we read our listing backwards, it the function view_summary. This uses the textbox type to display a file containing information on the currently selected options. We first build up the data in the file before displaying it.

Our next function is select_dma. Here the user must make one of four mutually exclusive options, so we use the a radio list. If you try this example yourself, be aware that the radiolist type was added in dialog version 0.4; if you have an older version then you will have to make do with a checklist.

Availability

Up above, the routine select_irq uses very similar code to allow the user to select the final option in our configuration utility.

The purpose of this script is to generate a C language header file defining the compile options for the kernel sound driver. The “save” function does this. Notice how a dialog box is displayed while the save is in progress.

Above that we see the on_off function alluded to previously. This avoids some repetitive code in the script.

Finally, we see the clean_up routine which allows the user to exit from the script. At the top of the script some default values are defined for the configuration options and the temporary filename to use.

The configuration utility still needs a few enhancements to replace the existing program, including more kernel options and error checking, but the example does function and gives a feel for what can be done with dialog. I encourage you to type it in and try it.

Advanced Features

There are several more things that dialog can do. You can create and use a dialogrc file to customize the color and appearance of the dialog boxes. Dialog also supports displays that do not provide color or graphics characters. The details are given in the man page.

Dialog is “8-bit clean”, meaning that that international character sets other than the standard US ASCII are supported.

More Applications

For some longer examples of using dialog you can look at the sample scripts included with the dialog source code. Under Slackware Linux, the system configuration scripts can be found in /usr/lib/setup.

There are undoubtedly many possible uses for dialog. You could, for example, create a fully menu-driven interface for Linux users not familiar with shell commands. This could even be expanded into a simple bulletin board system that allowed users to read mail and Usenet news, edit files, etc.

The example sound driver script could be expanded into a tool for configuring all of the kernel compile options.

Incidently, dialog is reasonably portable and should run with minimal changes on any Unix-compatible system that has a curses library. It can also be used from any shell script language.

Listing 1. Sound Driver Configuration Utility

Conclusions

Dialog is a simple yet powerful utility, true to the Unix tradition of making each tool do one thing well. It can add a polished look to your applications and make them easier to use.

Thank you to Savio Lam, the author of the dialog package, Stuart Herbert, who updated dialog to version 0.4, and Patrick Volkerding, who wrote the dialog-based setup scripts in the Slackware Linux distribution.

(Jeff_Tranter@mitel.com) is a software designer for a high-tech telecommunications company in Kanata, Canada. He bought a PC just over 18 months ago in order to run Linux and has not looked back since. He is the author of Linux Sound and CD-ROM HOWTO documents.

Writing an Intelligent Serial Card Driver

Randolph Bentson

Issue #5, September 1994

Every wonder what it's like to write a driver under Linux? Here's a summary of one hacker's experiences.

It started out innocently enough. I had been looking for an upgrade for my home system, a decrepit Unix workstation with only 8MB RAM and 40MB disk. I had been looking at 386BSD, but was a bit put off by the AT&T lawsuit and such. Suddenly last September I noticed this other system, Linux, and saw articles pointing to distribution sets. I copied a set to some floppies (on a Sun system, ugh) and borrowed a partition on a client's system that normally runs MS-DOS.

When I discovered how easily Linux loaded and how well it functioned, I knew I had the answer. I spent much of my monthly billing of that client to order a hefty 486 through their purchasing agent. I was a little concerned about assembling the pieces, but the guys in the manufacturing area were interested to see how it worked. I couldn't have kept them from putting it together if I had wanted to.

All this was very timely. I had to get back to my studies and finish my dissertation if I were ever going to graduate. Although I had access to plenty of systems over the Internet, it would have been messy analyzing the data from afar. With all the X tools now available on my new system, my work from home was greatly improved. For all this, I owed the Linux community a lot for the tools upon which I so thoroughly depended.

Then came my chance to pay back this debt (and benefit as well). Phil Hughes posted the following on a local Linux mail list:

```
... anyone out there want to write a driver for the Cyclades serial
board? This is an ISA-bus card with a Cirrus Logic RISC processor
on it....
Why do you want to do this?
1. Linux needs it :-)
```

```
   2. You will get a free 8-port board for your effort
   What you have to do:
   1. Write a driver that really works and make it
      available on the net.
   2. Write an article on this for Linux Journal
```

That sounded like a bargain! I immediately sent off a note saying I would be up to the task. I had done a lot of OS internals in the '70s, hacked in the BSD Unix kernel in the '80s, and had done some PC drivers in the last few years. That seemed to persuade Phil I could do the job. He wrote that his “big fear was that people who had no idea what a device driver is would decide to try to write a driver so they could get a free board”.

Things then went on hold while I finished up that last of the notes for my defense. (Actually I was hoping my advisor wouldn't ask what I was up to.) Fortunately it took a while for us to work out the details and for the card to arrive, so I didn't look bad by the delay. The developer's kit that finally arrived had a wealth of information: a 130-page data sheet on the Cirrus chip and lots of code fragments showing how the board is accessed. With that stuff in hand, I finally got to work inside the kernel.

The character tty-like interface is implemented across several files. Some look at the high-level part of the data flow; implementing the canonical character processing, newline to carriage return conversions, command line echoing, escape character processing, and such. The others are involved with controlling the devices; the console, the keyboard, and the serial lines. I started my driver by copying serial.c and changing all the names of the form rs_something to cy_something. I then had to hook this new low-level driver to the the system. There are two routines that are called to do this: cy_init and cy_open. Once these are slipped into tty_io.c, the Cyclades driver development is limited to the new file, cyclades.c.

The very first part was recognizing the boards and initializing the data structures. There was suitable code from Cyclades to do the former and the serial.c code took care of the latter. One major difference appeared. The serial.c code was based on the design where each port had its own IRQ which was to be deactivated when the port was closed. I had to move the IRQ setup code into the board initialization. The inital success was to boot the system and get the message reporting that the board and its ports were present.

The next phase was to fix up the cy_open routine. Once I stripped out the IRQ stuff, the rest was mostly to establish the link between the low-level driver and the high-level driver. This didn't need much of a change at all. I just added a subroutine call to initialize the port on the card, setting character size, baud rate, modem control signals, etc. That seemed to work, so I went on to the next part.

Then it got scary; actually sending characters. Again, using code from serial.c as an outline (hinting strongly where something has to to be done to the hardware), I changed cy_write and cy_interrupt.

cy_write is called whenever the high-level driver has queued characters to be sent. The serial.c version actually stuffed some of the initial characters into a hardware register, which starts sending characters out. I changed this so that I only enabled interrupts; the interrupt service routine would be the only code that put characters on the wire.

The interrupt service was even more scary. For the first time, code would be executed outside of the context of the calling program. If things went astray, it would be really hard to figure out where. Fortunately the code fragments from Cyclades and from serial.c merged without too much difficulty. Whenever an interrupt occurs, it indicates that something on the board needs tending to. It's a simple matter of programming (:-)) to poll each of the chips to see if any of their four ports need attention, and if so, whether it is for transmit, receive, or modem conditions. With some trepidation I compiled these changes and rebuilt and rebooted the kernel. A simple test: date > /dev/ttyC0 worked!

I was glad I had made this much progress because about this time I got a query from Cyclades. They wanted to know how things were going. With some relief I replied I was actively testing “increasingly feature-full versions of the driver.”

This was well received. They had issued a small advertisement regarding a Linux driver and were starting to get responses. To me this was also good news. There would be a pool of folks to test my code.

I continued my efforts, working up my courage to try the receive side, as well as addressing a mysterious (to me) kernel crash. As part of my tests, I was actually issuing the command “sync” just prior to the test transmission, so the kernel crashes hadn't hurt anything yet.

I traced the crash to a pointer that was being cleared prematurely. By comparing the serial.c code with my code, I discovered I had moved too many things around. Some minor checks in cy_close fixed that. (I also explored how I might get some kind of display out of the console showing what's going on. I looked around and found that although printk() isn't always appropriate for messages from within interrupt service routines (if ever), console_print() can be called anytime if interrupts are turned off. A little effort allowed me to sprinkle single character messages showing how far a routine had progressed. More calls to console_print() allowed me to zero in on what went wrong.) Finally, after making the best of the Memorial Day weekend, I reported the following:

Capabilities

- auto-detects card and uses assigned IRQ for given address
- presents DTR as function of open/close status
- can send/receive data on all eight ports
- works for login session from terminal problems
- reception of character before transmitting first character is seen as a “hangup” by something; once first character is sent, reception works fine shortcomings
- speed is fixed at 9600 baud
- mode is fixed at 8 bits, No parity, 1 stop bit
- modem status is ignored
- wait-on-open doesn't wait
- break is ignored
- written for release 0.99p12 testing
- haven't tested simultaneous send/receive
- haven't tested simultaneous multi-port operation

And a few days later I added:

> It appears the problem I reported for the Cyclodes driver is actually deeper within the kernel and appears with the other asynchronous drivers, i.e., it was there to begin with. Therefore I will ignore this problem for the moment, since the other ports work for all applications I know, and focus on getting the rest of the features right. First will be speed and line mode stuff, then the modem control.

and then:

> I've dropped in the speed setting code and tested it at speeds up to 19200. Once I rig some kind of loop-back cable, I'll check higher speeds.

It now recognizes parity errors and break, the wait-on-open feature works, and multiple simultaneous sends and receives have been tested. The upgrade to kernel 1.1.8 is done and I'm working with some other folks on testing it more rigorously.

Checking back in my log one can see how I worked in spurts. I spent a bit over a week overall on this, mostly in day-long chunks. This was after a lot of hour-long periods reading the documentation.

So what did I gain and would I do it again?

I got a chance to pay my debt to the community. I got to play inside the kernel. I'm more confident that I can write drivers for this system and get them to work. I also got a mux board.

I'm not sure I would do it again. Not that it was that demanding, but it did take time. I don't think the gains would be as great the second time around. Still, if an interesting-enough device was offered to me, I'd be tempted.

Randolph Bentson can be reached at: (bentson@grieg.seaslug.org)

Using iBCS2 Under Linux

Eric Youngdale

Issue #5, September 1994

iBCS2 emulation under Linux is a relatively new feature that offers you the ability to take an application that was designed to run under a system such as SCO Unix or SVR4 and run it directly on your machine running Linux. This feature is most useful for commercial applications for which the source code is not publicly available, and therefore would be impossible to simply port to Linux. In this article, I will give you an introduction to iBCS2.

iBCS2 emulation under Linux is a relatively new feature that offers you the ability to take an application that was designed to run under a system such as SCO Unix or SVR4 and run it directly on your machine running Linux. This feature is most useful for commercial applications for which the source code is not publicly available, and therefore would be impossible to simply port to Linux. In this article, I will give you an introduction to iBCS2 and tell you how to install the emulation code on your system in order to run iBCS2 programs. There will be future articles that will explain some elements in more depth.

by Eric Youngdale

The reason that iBCS2 is of interest is that there are a lot of commercial applications available for both SCO and SVR4 which people would like to run under Linux. The vendors who write these applications are often reluctant to port their application to a new platform such as Linux until they are sure that they will sell a lot of copies, and so there is no guarantee that a Linux port would ever be done. By providing iBCS2 compatibility, we suddenly make it possible to run hundreds of commercial quality applications under Linux.

At the time of this writing the iBCS2 code for Linux is still in ALPHA status. This means that you may experience problems; some applications will not run or will do the wrong thing. It also means that you are expected to have some familiarity with patching the kernel. That being said, there are only a few areas where work is still being done, and once these are complete the whole thing will become BETA. This may have already happened by the time you read this.

The term iBCS2 simply stands for the “Intel Binary Compatibility Specification 2”, and is the name of a standards document for binary compatibility between different systems running Unix on Intel 386, 486, and “greater” CPU's. Some parts of iBCS2 overlap with POSIX, and since Linux is POSIX compliant it means that there are portions of the emulation which are trivial. Unfortunately, there are also places where Linux and iBCS2 are quite different, so iBCS2 emulation is by no means trivial.

The iBCS2 emulator is mainly designed to be used as a loadable kernel module. This means that when you boot the kernel the emulator will not be present in the kernel's address space, so any attempt to run an iBCS2 application will fail. You must run a special program to “load” the emulator into the address space of the kernel, and once you have done this you will be ready to use the emulator. If you wish, you can unload the module when you are through with it to reduce the memory usage, but most people would not bother to do this.

To install and use iBCS2 under Linux requires one of several things. If the distribution of Linux that you are using already includes iBCS2, then you are in quite a good position. If this is so, then the chances are that all you will have to do is locate and load the module, and you will then be ready to run your iBCS2 applications. At the moment, the odds are fairly poor that your distribution has the iBCS2 patches already applied, so the remainder of this article will tell you how to get the emulator built and loaded.

As a prerequisite, you should be running either a 1.0 or 1.1 series kernel. You should know that the 1.1 kernels are officially development kernels, and may not be as stable as a 1.0 kernel. There are also frequent patches to the 1.1 kernels as development proceeds, so if you are a new user you are probably better off staying with a 1.0 kernel. If you are running anything older than a 1.0 kernel, you will have no end of difficulties, so your first step should be to upgrade. If you are running a 1.1 kernel, you should probably be at the most recent patch-level to minimize difficulties in compiling and installation.

Next you should get the source code for the emulator itself. This can be obtained via anonymous ftp from tsx-11.mit.edu or one of the many mirror sites of tsx-11, and can be found in the directory pub/linux/ALPHA/ibcs2. The current version of the emulator is called ibcs-940610.tar.gz, but by the time you read this a much newer version should be available. Once you have obtained this, you need to unpack it. Most people have their kernel source tree in the directory /usr/src/linux, and the examples I am about to give assume this.

To unpack, use the commands:

```
cd /usr/src/linux
gzip -d -c path/ibcs-940610.tar.gz | tar xvf -
```

where you substitute the actual path of the file obtained by anonymous ftp for the word “path”.

Now you need to see whether there are any patches which need to be applied to the rest of the kernel. Look in the directory /usr/src/linux/ibcs/Patches and see what is there. At the time this was written, there is a patch file for the 1.0 kernels, but there are no patches required for the 1.1 kernels. If you are running a 1.1 kernel, then skip down to the section on 1.1 kernels. If you have a 1.0 kernel, then you should do the following:

```
cd /usr/src
patch -p0 < /usr/src/linux/ibcs/Patches/kernel-1.0.pat
cd /usr/src/linux
make config
make dep
```

When you run “make config”, it will ask you whether you want iBCS/ELF/COFF in the kernel. The correct answer is “N” if you want to use iBCS2 as a loadable module. If you answer “Y”, the source tree will be configured so that the iBCS2 emulator will be linked directly into the kernel, but this has not been tested in a long time, and will probably not work very well.

Now you are ready to actually build the emulator.

Just type:

```
cd /usr/src/linux
make
```

and this will build both the kernel and the iBCS2 emulator. When it is done, you will have to install the kernel image in the proper place so that you can boot from it. If you want to boot from a floppy, then type:

```
make zdisk
```

and you will have a bootable floppy image. If you are booting directly from your hard disk, you will have to find the configuration files for the lilo program, and see where it expects to find the kernel image. You should copy the zImage file to this location and run lilo so that this information is properly recorded. If you are not sure of what you are doing at this step, please be careful, because you can screw up your system so that it is not bootable if you do something wrong. Once you have done this, you will need to reboot so that the patches you applied to the kernel are in effect on your system.

If you are running a 1.1 kernel, and there were no patches in the directory /usr/src/linux/ibcs/Patches that you needed to apply to your kernel, then you simply need to make sure that you are running a kernel based upon the source tree into which you have unpacked the ibcs2 sources. Then you type:

```
cd /usr/src/linux/ibcs
make
```

to build the emulator. If you were already running the kernel that was built from the current /usr/src/linux source tree prior to adding the iBCS2 stuff, there is no need to reboot. Otherwise you will need to either make a bootable floppy or copy the zImage file to the proper location and use lilo to register the new kernel.

At this point you are nearly done. There should be a file called /usr/src/linux/ibcs/iBCS which is the loadable module for the kernel and it is ready to be loaded. Unless you already have the "insmod" program on your system, you will need to obtain the sources to the modutils package, and you can get this from tsx-11.mit.edu in pub/linux/sources/sbin. At the time of this writing, the package is called modutils-0.99.15, and even though it is old it still works. If there is a newer version, some of the paths may have changed, so pay close attention. You can unpack this more or less anywhere, and the following commands illustrate how to do this:

```
gzip -d -c modutils-0.99.15.tar.gz | tar xvf -
cd modutils-0.99.15
make
make install
```

You will need to be root to do the "make install", because it needs to copy the executables to a system directory. Once you have done this, you run the insmod program to load the iBCS2 emulator, and the following command will do the trick:

```
/sbin/insmod /usr/src/linux/ibcs/iBCS
```

which should load the iBCS2 emulator into kernel memory.

Before you actually try and use iBCS for the first time, you need to create a few special device files. These are used for networking applications that come from SCO systems, and they are used for access to the local X server. The commands you need to run are:

```
mknod /dev/socksys c 30 0
ln -s /dev/null /dev/X0R
mknod /dev/spx c 30 1
```

Now you are truly ready to run iBCS2 applications on your Linux system; you simply run them in the normal way you would run any other program on your system.

In the limited space available to me, I have described how to get iBCS2 up and running on your system. You may find that there are problems of one kind or another, and there are text files in the iBCS2 emulator source tree which you can use to help troubleshoot the system.

There are also some elements of emulation which are missing right now. Support for some types of networking (specifically TLI) are currently missing, but people are working on this, so hopefully this will only be a temporary limitation. There is support for socket-based networking, however. Also, some applications may require shared libraries of some kind or another. This is one of the areas where work is still in progress, so your best bet is to simply see what the status is by asking on the iBCS2 channel or looking in the directory of the ftp site that you got the emulator sources from, in a file called libc_s-<date>.tar.gz, libc_s-940616.tar.gz as of this writing.

In the end you may wish to obtain a demonstration version of a package before you spend money to buy the package, and you can use this to see how well the application will actually work. Some demonstration versions are available via anonymous ftp; there is a demonstration version of WordPerfect available from ftp.wordperfect.com in ftp/unix/demos/sco/sco.z. Also, there is a list of applications that are known to work in the iBCS2 emulator source tree, and this is updated every so often.

If you wish to keep up with the latest developments in iBCS2 emulation, you can join the mailing list that the developers use for discussions, posting announcements and patches. To join, you can send a mail message to linux-activists-request@joker.cs.hut.fi, and you should include the line:

```
X-Mn-Admin: JOIN IBCS2
```

either in the header or as the first line of the message. You should get a message back to confirm that you are on the list, which you should save as this also tells you how to get off the list if you should ever need to do so.

Eric Youngdale Eric Youngdale has worked with Linux for over two years, and has been active in kernel development. He also developed the current Linux shared libraries.

Archive Index Issue Table of Contents

Advanced search

Linux Events: Two Views on Heidelberg

Bob Amstadt

Issue #5, September 1994

As this issue went to press, the Heidelberg “Linux and Internet Conference” met. Here are candid reactions from two conferees, garnered from e-mail messages.

As this issue went to press, the Heidelberg “Linux & Internet Conference” met. Here are candid reactions from two conferees, garnered from e-mail messages.

In a letter to the Wine1 development team,

Bob Amstadt wrote:

For those of you that are curious....I arrived in Heidelburg two nights before the Linux Congress. My first night was spent sleeping (I know - boring). The next day around lunch we ran across a handful of the other Linux developers. I made the mistake of not recognizing Linus from his picture. Everyone had a good laugh when I asked him what his name was.

That night all of the speakers met for dinner and beer. Lots of beer was had by all (some more than others).

The next day was the first day of the Congress. Around 300 people were in attendence. Apparently, about the same number were turned away because seating was limited. Linus gave the keynote speech to open the festivities.

He was extremely well received by the audience.

We then split into two sections. In the morning I listened to Remy Card, Stephen Tweedie and Theodore Ts'o talk about Linux filesystem. Also, Eric Youngdale spoke about iBCS2. I must admit that I am very excited about the iBCS2 work.

I led off the afternoon with a 45 minute talk about Wine. I was extremely pleased with the audience response. People were quick to cheer when I spoke of desire to be independent of Microsoft. I was questioned numerous times about Win32. Also, later that same evening I was offered a large sum of money to pay an intern to work on Wine.... I was also lucky enough to demonstrate Wine using a PCI bus computer. I assume it also had a Pentium because Wine was very speedy (for a change :-)).

The next day I presented a two-hour tutorial about using and hacking Wine. This was given to a much smaller group of people. Ted Ts'o followed with a two-hour discussion of Internet security.

All-in-all things went very well. My hope is that I inspired more support for our project.

There was some talk about holding another Congress next year although it is purely speculation at this point....

In a letter to the iBCS22 development team,

Eric Youngdale wrote:I am now home (and a bit jet-lagged) after having gotten back from Heidelberg (Note the time of this post - I have been up for 3 hours now :-). All in all, the meeting was an incredible success - there were 350 seats available and these quickly sold out and they reportedly turned away another 300 people. All of the talks were standing-room only, and there is already talk of another meeting next year in Berlin, though that is still speculation.

This is not really related to iBCS2, but there was a beta version of a Linux-native version of Maple on a machine that people could try out. I have a handout that they were giving out, but it is in German, so I cannot say that much about what they were saying.

It is not obvious at all in the English- speaking world, but a German computer user benefits from a much greater coverage of Linux in the media. This month's issue of iX (a German Unix magazine) has an article written by Drew about the PCI driver, and an article by Bob Amstadt about Wine. I understand that they have been running about one article a month for the past 6 months or so. A second magazine, C't, has also been running about 1 article per issue lately. In addition, there are a number of books available.

I walked down the main pedestrian street in Heidelberg and came across a store that sold computer stuff - in the front display window, clearly visible from the street, there were 3 Linux books plus the Yggdrasil CD-ROM. One of the books, written by Thomas Uhl and Stephan Strobel, has been translated into

English by Springer, and will soon appear in the U.[S,K]. market. This book is a basic introduction for the new user, so other more advanced books would also be required for a new English-speaking user. The other two German books for Linux were more advanced one contained a bunch of man pages translated into German, and the other one had to do with kernel internals.

Pictures were taken of the developers present, and are included here. JPEG files should become available soon :-).

One thing that became clear fairly quickly was that there is a lot of interest in iBCS2 within the user community. I gather that a lot of people are trying binaries of one sort or another, and a lot of these are never formally reported to us even if the binaries work perfectly. Also, there was a machine set up with the demo versions of both WP and Xess running this was the first time I had seen either program.

Bob Amstadt graduated from Rose-Hulman Institute of Technology in 1986 with a BS in both Electrical Engineering and Computer Science. For the past five years he has worked as an independent engineering consultant specializing in embedded control and communications systems. His first exposure to Linux was in December 1992 when he installed it on his e-mail server. He began work on Wine as a result of discussions on comp.os.linux in May and June of 1993.

Linux Programming Hints

Michael K. Johnson

Issue #5, September 1994

For several good reasons, the Linux standard library implements standard I/O (stdio) in a somewhat strange way. Unfortunately, many programs make unwarranted assumptions about how stdio is implemeted that cause the programs not to compile properly under Linux. I have alluded to this problem before in this column; this month I will explain how to fix such source code to compile under any operating system, including Linux.

Strange I/O

For several good reasons, the Linux standard libary implements standard I/O (stdio) in a somewhat strange way. Unfortunately, many programs make unwarranted assumptions about how stdio is implemented that cause the programs not to compile properly under Linux. I have alluded to this problem before in this column; this month I will explain how to fix such source code to compile under any operating system, including Linux.

by Michael K. Johnson

Linux stdio is not exactly non-standard; that would imply that there is a real standard for how standard I/O is supposed to be implemented. Theoretically, all I/O operations that use the stdio library should only use the “published” FILE mechanisms, which are abstract, and should not pay any attention to details. Unfortunately, many stdio implementations are rather slow, and do not provide functionality that programs need.

Instead of writing a working replacement for stdio, many programmers chose to abuse the stdio interface by directly accessing “private” members of the FILE structure that are not guaranteed to be the same from system to system. In practice, this worked very well from system to system, because almost all the systems came from the same source and a prototype using the same variable names was widely available.

Programmers learned, for instance, that the _cnt member of the FILE structure contained the number of bytes which had been read by the library but not yet read by the application, and that the _ptr member contained a pointer to the buffer in which the characters that had been pre-fetched by the library were stored. It was general knowlege that behind the scenes, the _filbuf() (sometimes called _ _filbuf()) macro was called to cause the stdio library to read more characters.

This worked as long as everyone used similar stdio implementations. Many well-respected applications used these methods to get around stdio; GNU emacs and the Rand MH mail handler are among them.

Linux is different.

The Linux stdio is based on the GNU libg++ iostream I/O. The FILE structure looks, in part, like this (from libio.h):

```
int _flags;          /* High-order word is
_IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags
char* _IO_read_ptr;   /* Current read pointer */
char* _IO_read_end;   /* End of get area. */
char* _IO_read_base;   /* Start of putback+get area. */
char* _IO_write_base; /* Start of put area. */
char* _IO_write_ptr;  /* Current put pointer. */
char* _IO_write_end;  /* End of put area. */
char* _IO_buf_base;   /* Start of reserve area. */
char* _IO_buf_end;    /* End of reserve area. */
```

This isn't at all the same. It is better optimized: instead of having one _ptr element, it has one pointer for reading, and one for writing, and a buffer for each as well. Instead of keeping track of the number of characters in the buffer, a pointer to the end of each buffer is kept, as well as the curernt pointer. It makes it easier to use all sorts of things as files, including shared memory, SYSV IPC, and anything else that fits the paradigm; it is dynamically extensible. It is also shared between C++ and C, and makes the C++ iostream implementation more robust because of the extra testing it gets as a standard io package.

If you have worked with the Linux or GNU C libraries in the past, you will notice that the names have changed. They used to be shorter names like _pbase and _pptr that looked like they were related to the old stdio names. In November 1993, the names were changed to what you see above. It is not anticipated that these will change again in the foreseeable future. See the sidebar “Old Names to New” for a listing of how the names changed.

Replacing direct access to the members of the FILE structure with abstract macros can make it possible to compile offending source on any system. Since the Linux stdio makes a distinction between reading and writing, the first thing to determine is whether each code fragment is reading or writing. Then you replace the direct use of the members of the FILE structure with macros; ones that are specific to reading an writing. Finally, you write the macros; one set for Linux, and one set for “standard” stdio. Here are some of mine:

Old Names to New

Under Linux or other similar stdio implementation:

```
#ifdef _STDIO_USES_IOSTREAM /* defined in libio.h */
#define FWptr(f) ((f)->_IO_write_ptr)
#define FRptr(f) ((f)->_IO_read_ptr)
#define Fptr(f)  (((f)->_IO_file_flags &&    \
                     _IO_CURRENTLY_PUTTING) ?  \
                 FWptr(f) :    \
                 FRptr(f))
#define FWcnt(f) (((f)->_IO_write_end - \
                 (f)->_IO-write_ptr) > 0 ? 0 : \
                 (f)->_IO_write_end - (f)->      \
                      _IO_write_ctr)
#define FRcnt(f) (((f)->_IO_read_end -       \
                 (f)->_IO_read_ptr) > 0 ? 0 :       \
                 (f)->_IO_read_end - (f)->       \
                      _IO_read_ctr)
#define Fcnt(f)  (((f)->_IO_file_flags &&    \
                     _IO_CURRENTLY_PUTTING) ?  \
                 FWcnt(f) :    \
                 FRcnt(f))
#define Ffill(f) __underflow(f)
#define Fflsh(f) __overflow(f)
```

Under “standard” stdio:

```
#else /* standard stdio */
#define Fptr(f)  ((f)->_ptr)
#define FWptr(f) Fptr(f)
#define FRptr(f) Fptr(f)
#define Fcnt(f)  ((f)->_cnt)
#define FWcnt(f) Fcnt(f)
#define FRcnt(f) Fcnt(f)
#define Ffill(f) _filbuf(f)
#define Fflsh(f) _flsbuf(f)
#endif
```

Note that some code may use **f->_cnt** as an lvalue (a variable to which something is assigned). In these cases, **f->_ptr** will always also be assigned to; both need to be updated at the same time in the standard stdio library. Since the “count” values in these abstraction macros are calculations for iostream-based stdio, they cannot be lvalues. However, since they depend on the “pointer” values and the “end” values, and the “pointer” values are updated, and the end" values don't change, they do not need to have the updated values assigned to them. Therefore,

```
f->_ptr++;
f->_cnt; ;
```

becomes (assuming that the code is reading using this pointer):

```
FRptr(f)++;
#ifndef _STDIO_USES_IOSTREAM
```

```
   FRcnt(f); ;
   #endif
```

or simply

```
   Fptr(f)++;
   #ifndef _STDIO_USES_IOSTREAM
   Fcnt(f); ;
   #endif
```

if you are not sure whether the code is reading or writing.

I will warn you: trying to apply these instructions and macros to code you are porting without understanding the code you are working on is likely to be disastrous. Your application may compile, but quietly lose data if you put the wrong macros in. It is most important not to use the **FR*()** macros when the library is writing, and not to use the **FW*()** macros when the library is reading. If you can't tell which is being done, you are far better off using the generic versions, **Fptr()** and **Fcnt()**, than you are guessing.

There are other mistakes waiting to be made, and I can't cover them all, because I don't know what they all are. The source code to the Linux C libary is available via ftp from tsx-11.mit.edu and sunsite.unc.edu, and is distributed with many Linux distributions. Reading the libc source code (usually found in /usr/src/libc-linux/libio/), and understanding what it is doing, is the safest route to knowing what to do when porting code that makes assump-tions about stdio. This article alone can only help you along your way; you will still have to understand the program you are porting and the Linux stdio to achieve success.

Michael K. Johnson may be reached by e-mail at: (johnsonm@merengue.oit.unc.edu)

Archive Index Issue Table of Contents

Advanced search

Internet Tour Guides

Putnam Barber

Issue #5, September 1994

The author gamely attempts to bridge the gaps between the characters he brings to life in his diary of exploration and the rest of us, the imagined readers of his book.

If you wanted to visit Cyberia, the psychic landscape described in the recent book of the same name by Douglas Rushkoff, you'd do well to prepare yourself with a trip through his pages. But you wouldn't have much to guide you when you were done; vague references to various locales in and around San Francisco and a strong sense of an extended group of people with a decidedly different take on life.

Here's a sample (from the chapter called “Gardners Ov Thee Abyss''):

The Temple Ov Psychick Youth is a nett-work for the dissemination of majick (their spellings) through the culture for the purpose of human emancipation. TOPY (rhymes with soapy) began as a fan club and ideological forum for Genesis P. Orridge, founder of industrial band Throbbing Gristle and its house spin-off Psychic TV, but soon developed into a massive cultish web of majick practitioners and datasphere enthusiasts. They are the most severe example of technopaganism....

Familiar figures (Timothy O'Leary, Mitch Kapor) make guest appearances in Cyberia. Familiar technologies (discos, BBSs) do as well. The author gamely attempts to bridge the gaps between the characters he brings to life in his diary of exploration and the rest of us, the imagined readers of his book. But in the end Cyberia comes up more a collection of in-your-face attitudes than a place; more a self-absorbed and self-reverential crowd slinging buzz words and recreational drugs than a representation of the forces that could (on the last page) ”slowly pull our society; even our world; past the event horizon of the great attractor at the end of time''.

If, on the other hand, you wanted to take a tour of cyberspace, as it exists today and is evolving on the anarchic Internet, you'd do well to prepare yourself with a trip through the pages of Ed Krol's The Whole Internet User's Guide and Catalog. There are literally dozens of books that cover similar ground on the shelves of any well-stocked bookstore. It's hard to imagine, though, one that would be of more general use to anyone familiar with a desktop computer who needed the keys to unlock the mysteries of exploring the 'net.

This is the second edition of a work that (by its publisher's count) sold a quarter of a million copies in the first edition. That popularity is, to my taste, well deserved. This is a book that can lead you step-by-step through first-time use of some very powerful, and very frustrating, tools. And it's a book you can return to time and again for useful tips as your knowledge grows and your sense of the possible expands. The detailed instructions on the major tools; ftp, mail, newsreaders, gopher, WAIS and WWW; go well beyond the basics without trying to substitute for a manual page or technical compendium. Yet they always start from a clear vision of what an explorer will need to master the tools personally, to enjoy creative use in place of keystroke-by-keystroke repetition.

This is not, though, the book I'd choose as a gift for someone newly venturing alone onto this network of interconnected computers. The goal of serving the needs and interests of people as they grow in their use of the 'net is well met. But it's met by saying so much that any true newcomer can hardly avoid being overwhelmed.

Throughout, though, Krol adopts a breezy, reassuring tone that seems directed at newcomers. It occasionally leads to a heartiness that strikes a false note with me.... "The commands you use may be slightly different, to make them more like a 'normal' command on your computer system, but when and why you use which command will remain the same.'' Only if it works! It doesn't take too long to learn that one machine's ”slightly“ is another's light-year. Keeping the reader informed (as he does) when upper vs. lower case makes a difference is a step in the right direction. But there are just too many places on the 'net where getting something ”slightly“ wrong means getting nothing at all. A brief discussion of strategies to employ in such cases wouldn't be out of place.

In general, this book is pitched toward the user on a campus or in a large organization where there are sources for day-to-day tech support available for the asking. He encourages his readers to ”ask the systems administrator“ about intractable problems. Little guidance is provided for people on community systems or low-cost Unix providers, such as searching for on-line help in the form of manual pages or FAQ files for the quirks that every 'net cruiser comes to know and love, or even bravely asking the question in newsgroups where there may be solace for the baffled.

The great strength of this book is opening up practical use of the 'net as a source of information for research or personal interests. Seventy-two pages at the back list resource after resource for exploring topics broad and arcane. The index is divided into two parts; one ”technical'', the other a “catalog” of gopher sites to visit and files to transfer.

It disappoints me, though, that there is so little recognition of the possibility that a new sort of community is growing up in cyberspace, where world-circling friendships and alliances can be formed for all manner of purposes unrelated to the location of data or capture of information. The catalog cites “December's Guide to Internet Resources” (anonymous ftp to ftp.rpi.edu, cd pub/communications) without shedding much light on the fascinations of computer-mediated communications. IRC gets four not very informative pages; MUDs two. The possibility of using the MUD metaphor to link people for real-world, not fantasy, tasks gets one cryptic sentence about “MOO-style” conferencing tools.

And while I'm on the subject of things being cryptic, let me carp for a moment about cross-references. There are important terms in the text that don't show up in the index (“terminal specification”) or in the glossary (“daemon” and “escape”). Some subjects are mentioned once and carefully explained. Others pop up time and again (“regular expressions”) but are never addressed. There's a seven-page “Unix Primer” (very useful) in Appendix D. But these pages weren't included in the Technical Index, so the only reference to “grep” (for example) sends the reader to a footnote which advises learning more about Unix, not to the brief and helpful explanation of this command which is provided in Appendix D.

In a first edition, these missed opportunities might be expected. After 250,000 copies have been printed, it should have been possible to find, and deal with, most of them.

Carping aside, Ed Krol's The Whole Internet has great strengths. Strengths which justify its position as a pre- eminent guide for touring the Internet. Not the least are the numerous full-text examples of terminal sessions, where every keystroke is shown and many are explained. These examples make this book particularly useful for anyone who has an occasional reason to pick up a new tool and visit an unfamiliar part of the 'net. If you anticipate having that need, this tested and updated classic will be a good companion to have at your side on the journey.

Cyberia: Life in the Trenches of Hyperspace, by Douglas Rushkoff. HarperSanFrancisco (a divison of HarperCollins Publishers). ISBN 0-06-251010-X. $22.00.

The Whole Internet User's Guide and Catalog (second edition), by Ed Krol. O'Reilly & Associates, Inc. (A Nutshell Handbook). ISBN 1-56592-063-5. $24.95.

Letters to the Editor

Various

Issue #5, September 1994

Readers sound off.

From the Wilds of Florida…

Ye brave souls at the Journal:Perhaps it was the pioneer spirit of Linux that first caught my attention. Maybe it was the sense of computing in the wilderness with no official support from the creators. It could have been the opportunity to mangle the source code beyond any recognition. Or was it the 'subtle' marketing campaign in .signature files that read, “I use and recommend OS/2 and Linux; 32-bits and Microsoft-free.” Nope, it was the price; free. Free is good. Besides, when did calling IBM or Microsoft do any good? Real computer users solve their own problems or get help from the net. But since free required a full Internet feed that is unknown in this part of the Union, I allowed Walnut Creek to sell me a CD. Well, it wasn't free any longer, but I don't think that makes any difference, does it?

Well, Yggdrasil slipped a tiny subscription card into the back pouch of the manual. In fact, it was your subscription card... So I scanned the Unix conference at Mac's Place BBS and found a positive comment and no negative comments. I see that you are also engaging in a subtle guerrilla marketing campaign.

Please start my two-year subscription immediately! No, don't take a coffee break! DO it NOW! Trust me, it won't take long, it barely hurts, and you might even enjoy it... Well, ok, finish the letter first...

Which brings me to a special request... I have this friend... He's a good person overall, but a bit of a slacker... And I think he would enjoy your magazine. But if I allowed him to borrow any of my copies, they would disappear into the chaotic vortex of pizza boxes, aluminum cans, and tractor-feed printer paper that defines his world. So, would you please send me an additional copy of my first issue? I guess we could call it a 'concurrent' sample issue for a subscriber, couldn't we? I knew you'd understand...Richard K. Evans, rk.evans@cfactory.com

Glossary Needed?

Yesterday I subscribed to *Linux Journal*. I haven't read the entire Journal as yet, but I do enjoy Phil Hughes' writing. Bernie Thompson has a clarity that is easily understood, and I congratulate you folks for the presentation of subjects that are not easily communicated.

Page 29 of the July issue reads “The Open Development of Debian”, by Ian Murdock. This one page is well beyond my understanding. I suspect Debian is some kind of perturbation of Debra and Ian, and the fully cocked and loaded Linux folks probably well understand these developmental pseudonyms, but if you're looking for a broad support for Linux, I and others will need to understand these terms. Where do you go?

Maybe a glossary in the *Linux Journal* that is appended to by the authors as these terms develop, or maybe a complete definition of the terms in the article.George L. Clute, geo@aircarg.celestial.com

LJ Replies:

A glossary won't fit, but we do intend to define terms that need defining. Letters to the Editor help tell us which terms need defining. In the future, the Debian column will include a sidebar (or something) which explains what Debian is.

Linux Domain Names Usurped

Dear Sir,I would like to register my deep disapproval of the actions of Aris Corporation in registering both the Internet domains 'linux.net' and 'linux.com' without any consultation with the Linux developers. This kind of action, which is equivalent to me setting up something like linuxjournal.com without your permission, does not appear to be of benefit to the Linux community.

The Internet naming is already the source of several battles over this kind of thing and one lawsuit (Adam Curry versus MTV). The Linux community getting involved in this only makes things worse. A sensible linux.com domain holding any company working with Linux could have been created. Now we are at the mercy of whatever Aris Corporation decides to do with that domain.Yours, Alan Cox

Reader Tips for Directory Tree Cloning

Dear Editor,I enjoyed the SysAdmin column in *LJ* #3. I wanted to pass on a tip that could save some time and trouble when cloning a directory tree onto a new filesystem. The article had us save the current file tree in a tar file and then remove it. This requires lots of extra space for the tar file, space that just may not be available.

Instead, assuming my new partition is /dev/hd5 and the directory I want to move is /users, it can be done like this (# is the root shell prompt):

```
# cd /
   # mv /users /users.old
   # mkdir /users
   # mount /dev/hd5 /users   # new partition ready
   # cd /users.old
   # tar -cf - . | (cd /users ; tar -xpf -)
```

The last step is the crucial one. The first tar sends a tar archive of the current directory tree to standard output. The parentheses enclose actions done in a subshell. The subshell changes directory to the new filesystem and runs a tar there to extract the tree from standard input. When done, the usual sequence of actions is

```
           # cd /
           # umount /users
           # fsck /dev/hd5         # for safety
           <do a fresh backup of the file system>
           # mount /dev/hd5 /users
           # rm -fr /users.old     # kerbam!
```

This trick with two tar commands in a pipeline was documented in the original tar(1) man page; it has not survived into the current tar(1) for all Unix vendors. I hope this letter will help save some time and trouble for anyone needing to copy directory trees.Arnold Robbins, arnold@skeeve.ATL.GA.US

Kudos and A Blind Eye

I like the new look.

I also liked the in-depth articles on the GNU C Library and the VT interface. Humberto Ortiz Zuazaga, zuazaga@ucunix.san.uc.edu

P.S. Don't think I didn't see the misprints, I'm just being nice to you.

LJ replies:

Thank you. We are working on reducing the number of typos, and each issue should improve in this respect (as well as in others). We are interested in hearing our readers' opinions on articles they especially like, and what they like about them, as we try to establish a balance between beginner, intermediate, and highly technical articles.

Linux Port to MIPS

I've read your article 'Stop the presses' on page 6 of *Linux Journal* #3. It mentioned the port of Linux to the the 68k, PowerPC and Alpha machines.

I just wanted to add that I've started to port Linux to the MIPS. I'll first support the MIPS R4600. Lucky circumstances will make it possible for me to spend three full months of time to do the work. I hope to have a quite stable system ready within that time.Ralf Baechle, linux@informatik.uni-koblenz.de (Fido: Ralf Baechle 2:245/5618.2)

Linux: New Products and Events

LJ Staff

Issue #5, September 1994

POET 2.1, Cyclom-8Ys and more.

POET 2.1

POET 2.1, the Cross-Platform Object Database for C++ is now available on Linux both in a single-user “Personal Edition” and in a Client/Server “Professional Edition”. It features cross-platform support, not only at the source level, but also provides binary compatibility between objects on all supported platforms, including many Unix platforms, Novell, and Macintosh.

POET is provided as a set of C++ classes which provide a fully object-oriented system, including persistent classes.

For more information, e-mail info@poet.com or call (408) 970-4640 in the US, or e-mail info@poet.de or call +49 (0)40 609 90 18 in Germany.

Cyclom-8Ys

Due to the high volume of inquiries received, Cyclades Corporation has announced the release of the Linux driver onto the Internet for its intelligent RISC-based high-speed (115 Kbps) 8-port card, the Cyclom-8Ys. The driver was developed in cooperation with Randolph Bentson, a Seattle-based computer science consultant.

List price is $459, but Cyclades is offering the board for $99 to resellers who are first-time buyers. Interested distributors and resellers should contact Cyclades Corp-oration's sales team for more details, and end users may ask for a list of resellers in their region.

Cyclades Corporation is located at 44140 Old Warm Springs Blvd., Fremont, CA 94538. You may call toll-free (800)347-6601, call (510)770-9727, fax (510)770-0355, or e-mail cyclades@netcom.com.

Unix Expo

Linux Journal will be at Unix Expo in New York City from October 4-6. Please stop by and see us at booth #02078. If you'd like a free pass to Unix Expo, call us at (206) 527-3385 before September 10.

Also, during Unix Expo, the New York Linux Users Group will have their regular meeting. You can find them in Room 1E20 of the Jacob Javitz Convention Center on Tuesday, October 4 at 5:30 pm.

Linux System Administration

Mark Komarinski

Issue #5, September 1994

An easy-to-set-up and easy-to-use tool for accessing DOS filesystems (and others) from Linux.

One of Linux's amazing abilities is to read the filesystems of other operating systems. The most commonly used filesystem for the PC architecture is the DOS FAT (File Allocation Table) filesystem. This is the filesystem used by PC-DOS, MS-DOS, DR-DOS, etc. Linux has two ways to read files that are in the FAT format. First, you can compile support into the kernel (at the DOS filesystem prompt in 'make config') and then mount a drive using the command: mount -t msdos /dev/hda3 /mnt. This will mount your third partition of the first drive onto the /mnt directory.

Now this is fine and dandy if you have a hard drive, but what about floppies? If you're anything like me, you don't label your disks, and important things like next month's article wind up next to your unused Windows 3.1 install diskettes (now formatted) on a counter. Mounting and unmounting (with umount) would be a pain in the tail, because you now have to issue three commands to get a directory, the mount, the directory, then the umount. Sure you can put this in a batch file, but mounting takes a while. There has to be a better way. There is.

Most of the distributions of Linux come with a package called mtools. This handy package was designed for Suns and other systems that did not have easy FAT support. It's system-independent, so you don't need to go tinkering with your kernel or re-compile it. Just install the software and go. Most of the commands have names that a DOS user would know such as del, dir, md, rd, and so on. The difference is that the mtools have the letter 'm' in front of the command to specify that it's an mtools command. So 'dir' would be 'mdir', 'del' would be 'mdel', and so on.

Because mtools does nothing to your kernel, you don't need to mount and unmount diskettes. Need to copy a file quickly from a floppy? Just use 'mcopy'.

The box at left lists the available mtools commands and their respective DOS counterparts. Two commands that do not have real DOS counterparts are mread and mwrite. These two commands are defined as low level reads or writes. Mread will read only from a FAT filesystem to a Linux file, and mwrite will write only from a Linux file to a FAT filesystem. You may want to use mread and mwrite if you are in a multi-user setup and you want some people to be able to only read FAT filesystems, and others to only write to them. Remember that the FAT filesystem has no user ownership and all files can be modified if users have access to mtools. Be sure to keep this in mind if you plan on having multiple users.

In order to use mtools properly, you must have a /etc/mtools file set up. To do that, you have to know how your floppy drives are seen by Linux. As I mentioned in a previous article, all of your physical devices such as your modem, monitor, printer, and sound card, are listed as special files in the /dev directory. The same is true of your drives. There are a list of files in the /dev directory that start with fd0 or fd1 of the form:

/dev/fdXYNNNNwhereX 0 - First floppy drive (A)1 - Second floppy drive (B)Y d - low density 5.25"D - low density 3.5"h - high density 5.25"H - high density 3.5"NNNN - Three or four digit number listing the number of 1K blocks on that drive. A high density 5.25" can have 360, 720, or 1200 while a high density 3.5" can have 360, 720, and 1440. Low density 5.25" is 360, and low density 3.5" has 360 and 720.

This means about 18 files in the /dev directory just for floppy drives. Luckily, you won't need to remember these combinations except for special circumstances.

Why? Because two additional files, /dev/fd0 and /dev/fd1 are set up as auto-sensing devices. When Linux boots up, it checks your system setup (the BIOS) for what floppy drives you have. The BIOS then tells the kernel what floppy drives you have, what size they are (5.25" or 3.5") and if they are high-density or low-density. In addition, when you insert a pre-formatted disk in the drive, Linux can determine if you have a high-density or low-density disk. So you can mount a low density floppy in a high density drive by just using 'mount -t msdos /dev/fd1 /mnt'.

Okay, now back to mtools and /etc/mtools.

An entry in /etc/mtools looks like this:

```
d <device> <fat> <tracks> <heads>
<sectors>
```

Where:

d Drive letter you want to use. Yes, you can assign the first floppy drive to 'Q:' like you always wanted. Just be sure that your entry does not have the ':' in it. Just the drive letter.

device In most cases, this will be the /dev entry that you want to use. This can be /dev/fd0, /dev/hda1, or /dev/sda1 (for you SCSI people).

fat Size of the FAT table. Floppy drives and hard drive partitions have a size of 12, while large hard drive partitions have a size of 16. If you're unsure about your hard drive setting, use the 'fdisk -l' command which lists all of the partitions on all of your hard drives whether they are IDE or SCSI. DOS drives should say either 'DOS 12-bit <32M' or 'DOS 16-bit >=32M' Whatever is before the '-bit' is what you want for <fat>.

If you have a hard drive, or are using /dev/fd0 or /dev/fd1 as a device for an entry, you may fill in the rest as 0. This tells mtools to ask the kernel to find out. For other entries, you'll need to specify:

tracks Number of tracks on the drive. High density have 80 tracks while low density has 40.

heads Number of heads. Always 2 unless you have a really old drive. Like one of those drives where you have to flip the diskette over to read the second side.

sectors Number of sectors for the disk. 3.5 inch disks have 18 and 5.25 inch have 15.

The only time you would want to specify a drive using the tracks, heads or sectors is when you would want to format a disk. Set up the drives you would most want to format. For me, this is my high density 3.5 inch drive, so my entry would look like this:

```
   e /dev/fd0H1440 12 80 2 18
```

I have my A through D drives set up as my two floppy drives and two DOS hard drives. The helps me avoid confusion while mucking with my drives.

Once you have your /etc/mtools file set up, save it and try accessing the drives you just set up. Try 'mdir c:' to see if it works. If you get a directory, it's working. If you get a 'Probable non-DOS disk' or a similar error message, you may be looking at the wrong partition. Edit your /etc/mtools file again and find the correct partition with the 'fdisk -l' command. Once you have all your drives set up, you can have fun with all the mtools commands.

Now on to formatting floppy disks. Now that you have /etc/mtools set up, this is a bit easier to do. If you have a new disk, you'll need to low-level format the diskette before you can put a DOS FAT filesystem on it. Both of these steps are done by the DOS format command, but we have to do it in two steps (unless you make a shell script). If all of your diskettes are in DOS FAT format, you can skip the next paragraph and find out how to do quick formats the mtools way. If your disks are in tar format or have been used as Linux boot disks, you shouldn't have to low-level format them, because mformat is more flexible than DOS's “format /q” command.

First is the low-level formatting. This is done with the fdformat command. The fdformat command uses as its option the device file you used earlier to define the

device in /etc/mtools. For example, the high density 3.5" disk you would put in drive 'A' is /dev/fd0H1440. So, to format that disk would be:

```
fdformat /dev/fd0H1440
```

You should see a little message saying that it has 80 tracks, 18 sectors per track, total 1440KB. You should then see it formatting, then verifying the diskette. Once this is done, you can lay down a DOS FAT filesystem.

To install the FAT filesystem, the mformat utility is used. It will only accept a drive letter, which is why you wanted different drive types set up in /etc/mtools. Since I had my high density 3.5" drive set up as drive E, I can just type:

```
mformat e
```

and my diskette will now be in DOS FAT format. Since this merely lays down a new FAT filesystem and root directory, it is comparable to quick-formatting programs under DOS like 'format /q'. If you get an error or the disk is unreadable with mdir, go back and low-level format the diskette, then re-format.

Once you have the FAT filesystem on the disk, it is compatable with mtools, native DOS, and can be mounted from the Linux kernel. It is as if you formatted it under DOS.

You can also place any supported Linux filesystem on the diskette like ext2fs, xiafs and extfs. Merely replace the mformat command with the command used to make that particular filesystem, such as mke2fs. You too can have your own library of Extended File-system diskettes, each 1.4MB big.

Using mtools will help you keep some of your DOS compatability, and it's easy to setup and use.

Getting mtools

If your distribution did not come with mtools, or if you would like to get mtools for some other machine, you can use anonymous ftp to prep.ai.mit.edu or some mirror site (postings in gnu.announce almost always include a long list of mirror sites), set binary mode, cd to /pub/gnu, and get mtools-2.0.7.tar.gz. It should compile under Linux with no problems. MTools and DOS Commands Table