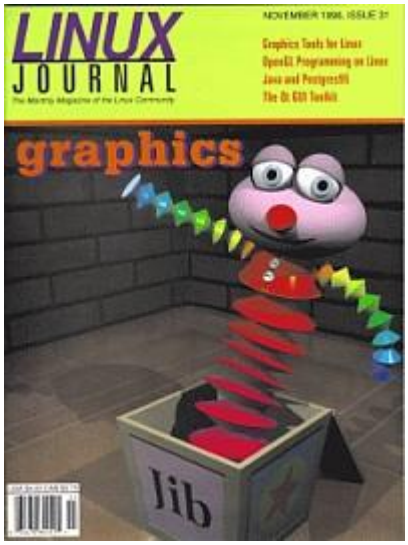
Linux Journal Issue #31/November 1996



Features

Linux Goes 3D: An Introduction to Mesa/Open GL *by Jörg-Rüdiger Hill*
Discover Mesa, a 3D graphics library that is source code-compatible with OpenGL.

Qt GUI Toolkit *by Eirik Eng*
Porting graphics to multiple platforms using a GUI toolkit.

Graphics Tools for Linux *by Michael J Hammel*
Can you really do professional graphic art on a Linux system? If you're aware of all the available tools, you can.

OpenGL Programming on Linux *by Vincent S Cojot*
How one student used Linux and OpenGL to build a 3D, network-capable tank game.

News and Articles

The Java Developer's Kit
by Arman Danesh

LJ Interviews Larry Gritz
by Amy Wood

The Linux-GGI Project
by Steffen Seeger and Andreas Beck

Java and Postgres95
by Bill Binko

Columns

Letters to the Editor

Directories & References

LINUX JOURNAL

Linux Goes 3D: An Introduction to Mesa/OpenGL

Jörg-Rüdiger Hill

Issue #31, November 1996

3D graphics aren't hard when you have an expert to tell you how it's done.

Recently, I installed Linux for the first time on my home computer. After the excitement of having a Unix workstation at home had faded away, I started looking for a way to port my molecular graphics program Viewmol to Linux. I used to work with IBM and Silicon Graphics workstations, and Viewmol had been written using the Silicon Graphics' Graphics Library (Iris GL). There are a lot of 3D graphics libraries available for Linux (over 180 are listed on the Technische Universitat Berlin web site, www.cs.tu-berlin.de/~ki/engines.html) including some rudimentary implementations of Iris GL (YGL at WWW.thp.Uni-Duisburg.DE/Ygl/ReadMe.html, 2D only, and VOGL at http://www.cs.kuleuven.ac.be/~philippe/vogl/), but none of the libraries had the full functionality that I needed—then I discovered Mesa. Mesa is a 3D graphics library which is source code compatible with OpenGL, Silicon Graphics' successor to Iris GL. Mesa's goal is to make programs which have been written for OpenGL runnable on every X windows system including Linux. So I took a better look at Mesa and decided to rewrite my program for OpenGL.

Mesa has been written mainly by Brian Paul over the last 3 years and is currently (as of this writing) at version 1.2.8. Nearly all of the OpenGL functionality is available; the only missing features are anti-aliasing, mip-mapping, polygon stippling and some of the texture querying functions. Mesa's home page, www.ssec.wisc.edu/~brianp/Mesa.html, lists a number of applications (basically scientific visualization tools, but also a VRML browser) which use it. Currently, Mesa can be called from C and Fortran routines.

While OpenGL has been designed as a software interface to high-performance (and high-price) graphics hardware, Mesa is a software-only solution which uses X windows to interface with the hardware. (Recently a SVGA driver and some support for 3D PC-hardware have been added to Mesa.) Therefore Mesa based programs usually execute slower than OpenGL based programs. Both

libraries are hardware independent and window system independent; thus, the handling of the window system is left to the application programmer. Having the programmer handle windowing is different from Iris GL, but was considered necessary for Mesa in order to achieve hardware independence. OpenGL is the standard for 3D computer graphics and is managed by the Architecture Review Board. Implementations are available for a number of operating systems: different flavours of Unix, Windows and MacOS. Mesa also supports all these platforms. OpenGL requires an extension, GLX, in the X server to run. Mesa does not need this extension as it emulates the calls to GLX. There are commercial implementations of OpenGL available for Linux which also include X servers with GLX.

OpenGL/Mesa (I will use only the term Mesa in the following text, but it should be noted that everything applies to OpenGL as well) do not provide high-level commands for describing models in 3D. They do provide the necessary graphics primitives (e.g., points, lines, polygons) to build and manipulate models. Mesa provides the programmer with the ability to perform model building and manipulation completely in three dimensional space. All the details of converting the 3D model to a drawing on a flat screen are handled by the library, including one of the most tedious tasks in 3D programming—removal of hidden lines and surfaces. Mesa also offers “special effects” such as texture mapping, fog or blending.

Mesa's primary ftp site is iris.ssec.wisc.edu, but it can also be found at the usual places for Linux. Installation is easy—first unload the archive file using the command:

```
gzcat Mesa-1.2.8.tar.gz | tar xf -
```

then for a.out give the command:

```
make linux
```

or for ELF give:

```
make linux-elf
```

Executing make will compile the Mesa library, the GL utility library (GLU), the tk and auxiliary libraries, and a whole bunch of example programs. (Mesa's makefile comes configured for 46 different operating systems, including MS Windows.) I have found compilation to be hassle free on at least Linux, AIX, Irix and OSF1. The compiled libraries can be found in Mesa-1.2.8/lib and should be installed in either /usr/lib or /usr/local/lib. The header files (Mesa-1.2.8/include) should also be copied to either /usr/include or /usr/local/include. This step was not included in our make process—the following examples all assume that Mesa is installed in both /usr/local/lib and /usr/local/include.

The compilation produces a total of four libraries.

- 1) libMesaGL.* contains all the basic graphics code.
- 2) libMesaGLU.* provides some higher level functions, such as subroutines to draw geometric objects, splines etc.
- 3) libMesaaux.* is an auxiliary library that is not really a part of Mesa. Since Mesa is window system independent, some simple window manipulation functions were needed. This library was created to demonstrate the features of OpenGL in the OpenGL Programming Guide ("The Red Book"). It is included with Mesa so that all the example programs from "The Red Book" which are included in the Mesa distribution can be compiled.
- 4) libMesatk.* is another window system support library. libMesaaux.* relies on libMesatk.*, so to successfully link a program which uses libMesaaux.* -lMesatak must be added to the command line.

I don't wish to bore you with the usual "Hello, world" program. So since Mesa is a graphics library, we will start with something more appropriate to its function. Let's draw some geometric shapes:

```
#include<stdlib.h>
#include<GL/gl.h>
#include<glaux.h>
void display(void
{
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(0.0, 0.0, 0.0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
  glBegin(GL_LINES);
  glVertex2f(-1.0, -1.0);
  glVertex2f(1.0, 1.0);
  glEnd();
  glBegin(GL_POLYGON);
  glVertex2f(0.25, -0.75);
  glVertex2f(0.75, -0.75);
  glVertex2f(0.75, -0.25);
  glVertex2f(0.25, -0.25);
  glEnd();
  glFlush();
}
void main(int argc, char **argv)
{
  auxInitDisplayMode(AUX_SINGLE | AUX_RGB);
  auxInitPosition(0, 0, 500, 500);
  auxInitWindow(argv[0]);
  auxMainLoop(display);
}
```

As you can see, there is a naming convention for all functions. All Mesa functions start with the letters **gl**. Functions in the auxiliary library start with the letters **aux**. The first two calls to the auxiliary library in **main()** specify the desired frame buffer configuration, i.e. single buffered, rgb mode. (There is also a double buffered configuration for animations and a colormap mode, but rgb mode is preferred and is easier to handle.) The third call opens a window, and

the fourth call enters an infinite loop in which the function **display()** will be called whenever a redraw request is received from X windows. As I mentioned earlier, Mesa does not deal directly with the interface to the windowing system. The auxiliary library provides only the very basics and is not suited for larger programs—more about alternatives later.

The **display()** function starts with two calls to clear the background of the window to white—first we specify the desired color with **glClearColor()** and then we clear the color buffer with **glClear()**. Following that we set the drawing color to black and set up a projection matrix using **glMatrixMode()**, **glLoadIdentity()** and **glOrtho()**. Since Mesa can handle all the necessary mathematics to create a 2D drawing from our 3D world, we have only to give the instructions for making the projections. First we use **glMatrixMode()** to specify that we are going to manipulate the projection matrix. (There is a modeling matrix to translate or rotate objects that we discuss later.) Then we load an identity matrix to initialize the matrix stack, and finally, we use **glOrtho()** to specify an orthogonal projection. Now we draw a line from the lower left to the upper right corner of the window and a square in the lower right quadrant.

All drawing primitives in Mesa are created by embracing their vertex specifications with calls to **glBegin()** and **glEnd()**. In the call to **glBegin()** we specify the primitive we wish to draw. Available primitives are GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_POLYGON, GL_QUADS, GL_QUAD_STRIP, GL_TRIANGLES, GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN.

Finally, we call **glFlush()** to tell Mesa to flush its graphics pipeline and display the objects we have specified. To compile our demo program (assume it has been stored under the name demo1.c), we execute the following command (note that the standard Xlib, the X extension library and the math library are needed to resolve all references from Mesa):

```
cc -o demo1 demo1.c -I/usr/local/include -L/usr/local/lib \
-lMesaaux -lMesatk -lMesaGL -lXext -lX11 -lm
```

Figure 1 shows the result of the execution of our program which is exited by pressing the <ESC> key. This exercise was definitely easier to program using the Mesa libraries than using X window directly.

Figure 1. Demo Program Output

Now, since Mesa is a library for 3D graphics, let's create a three dimensional object. Replace the **display()** function in our first example with the following calls:

```
 void display(void)
{
```

```
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(0.0, 0.0, 0.0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
  glRotatef(45.0, 0.0, 1.0, 0.0);
  glRotatef(30.0, 0.0, 0.0, 1.0);
  auxWireCube(1.0);
  glFlush();
}
```

Recompile. Instead of using **glBegin()/glEnd()** pairs to specify the vertices for an object, we now use one of the auxiliary library functions to draw a wire frame cube. The two **glRotatef()** calls before the drawing change the model view matrix. Since rotations can only change the model view matrix, we are not required to switch to the model view matrix mode explicitly; the switch is made automatically by Mesa. The first call rotates the object 45 degrees about the Y axis, the second 30 degrees about the Z axis. The final letter, **f**, of **glRotatef()** indicates that its arguments are floating point. (Functions that end with the letter **d**, **i** or **s** accept arguments of type double, integer or short, respectively.) Internally, Mesa uses the float version of a function; thus, calling this version directly saves an additional function call within Mesa. Figure 2 shows the output generated by running this version of our program.

Figure 2. Wire Frame Cube

Next, we add interactivity to our program. To allow an interactive rotation of our cube, we have only to add some lines that deal with input:

```
#include<stdlib.h>
#include<GL/gl.h>
#include<glaux.h>
float xangle=0.0, yangle=0.0;
void display(void)
{
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(0.0, 0.0, 0.0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
  glRotatef(xangle, 1.0, 0.0, 0.0);
  glRotatef(yangle, 0.0, 1.0, 0.0);
  auxWireCube(1.0);
  glFlush();
}
void rotX1(void)
{
  xangle+=5.;
}
void rotX2(void)
{
  xangle-=5.;
}
void rotY1(void)
{
  yangle+=5.;
}
void rotY2(void)
{
  yangle-=5.;
}
void main(int argc, char **argv)
```

```
{
  auxInitDisplayMode(AUX_SINGLE | AUX_RGB);
  auxInitPosition(0, 0, 500, 500);
  auxInitWindow(argv[0]);
  auxKeyFunc(AUX_LEFT, rotX1);
  auxKeyFunc(AUX_RIGHT, rotX2);
  auxKeyFunc(AUX_UP, rotY1);
  auxKeyFunc(AUX_DOWN, rotY2);
  auxMainLoop(display);
}
```

The **display()** function is nearly identical to the one in the previous example, we have exchanged the hard coded angle for a variable. Our **main()** function now includes four calls to **auxKeyFunc()** allowing us to specify a callback function that is called when a certain key is pressed (the constants used here refer to the cursor keys). Finally, we need functions that will increase or decrease the rotation angle of the cube depending on which key is pressed. The program is again compiled in the same manner. When this version of our program is running, the cube can be rotated by pressing any of the cursor keys.

Sidebar: Messa/OpenGL Resources

We would probably prefer our application to use the mouse to rotate the cube, but we are currently limited to the functions provided by the auxiliary library. To undertake writing the program to use mouse clicks instead of cursor keys would require us to use one of the more sophisticated X windows interfaces (but thats another article).

Finally, we will add some light effects to our cube demo and show how to remove hidden surfaces. These calculations are also easily handled by calls to Mesa, and the programmer does not have to worry about the underlying, non-trivial mathematics. We again modify the **display()** function as follows:

```
void display(void)
{
  GLfloat light0[4] = {0.5, 0.8, 1.0, 0.0};
  GLfloat color[4]  = {1.0, 0.0, 0.0, 0.0};
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, color);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glLightfv(GL_LIGHT0, GL_POSITION, light0);
  glRotatef(xangle, 1.0, 0.0, 0.0);
  glRotatef(yangle, 0.0, 1.0, 0.0);
  auxSolidCube(1.0);
  glFlush();
}
```

Since drawing a wire frame cube with lighting enabled does not make much sense, we will use a solid cube. For a solid cube to be rendered correctly we need to remove hidden surfaces. In Mesa this can be accomplished by using the z-buffer which stores information about the depth value of a point in 3D

space. Mesa will then automatically only draw pixels which are visible. To initialize the z-buffer prior to the drawing we just add the constant **GL_DEPTH_BUFFER_BIT** to the call to **glClear()**.

For lighting calculations we cannot simply use a drawing color—we have to link a color to an object. Mesa uses "materials" to make this link and allows us to specify the properties of a material. The call to **glMaterialfv()** assigns red as the color for diffuse reflections to both the front and back sides of all polygons. We specify the position of the light with a call to **glLightfv()**. Mesa can use a number of different lights (at least 8 are guaranteed), and the constants **GL_LIGHT0 ... GL_LIGHT7** can be used to reference them. **GL_POSITION** informs Mesa that we are specifying a position (other possibilities include light and color), and the vector **light0[]** places the light on the specified axis at infinite distance. This particular axis is used to achieve different light intensities on the different faces of the cube. Notice that these two functions show another type of naming convention—both names end with the letters **fv**, i.e., the arguments are vectors of floating point values.

We also need to modify the **main()** function to include z-buffering and lighting calculations:

```
void main(int argc, char **argv)
{
  auxInitDisplayMode(AUX_SINGLE | AUX_RGB | AUX_DEPTH);
  auxInitPosition(0, 0, 500, 500);
  auxInitWindow(argv[0]);
  auxKeyFunc(AUX_LEFT, rotX1);
  auxKeyFunc(AUX_RIGHT, rotX2);
  auxKeyFunc(AUX_UP, rotY1);
  auxKeyFunc(AUX_DOWN, rotY2);
  glShadeModel(GL_SMOOTH);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);
  auxMainLoop(display);
}
```

First the constant **AUX_DEPTH** in the call to **auxInitDisplayMode()** instructs X windows to provide a window with a z-buffer. We then use smooth shading (**glShadeModel()**) to draw polygons that have varying color over the face of the polygon. If we used flat shading (the default), the different polygons would be clearly visible. Of course, that would not make a difference in the case of a cube, but would with other objects (e.g., replace the cube with a cone using **auxSolidCone(1.0, 1.0)** and see the result). Finally, we enable lighting calculations, **light0** and depth testing using calls to **glEnable()**. For depth testing we specify a function to compare the depth values so that only smaller values, i.e., closer to the viewer, are considered. Recompile. The lit, rotatable cube shown in Figure 3 is the output of our program after some rotations have been done.

Figure 3. Rotatable Cube

We have now covered the basic drawing operations to produce realistic 3D scenes using Mesa. The auxiliary library used in these examples is insufficient as an interface to the window system for larger scale programs. One alternative is to use the GL Utility Toolkit (GLUT) that transparently provides the same functionality as Iris GL (e.g., window and event handling, menus). GLUT was written by Mark Kilgard at Silicon Graphics and is available free. Another option is to use OpenGL widgets that are provided with the Mesa package in the widgets subdirectory. (This subdirectory must be compiled separately.) A program could then do all the windows and events handling in the normal X fashion and create one or more OpenGL widgets to display 3D graphics. Drawing into these widgets can be accomplished using calls to Mesa. As a final example of what Mesa can do, Figure 4 shows a rendering of a molecular orbital of benzene using my molecular graphics program Viewmol. (The OpenGL/Mesa version of Viewmol has not been released yet, but will appear at the same locations where the Iris GL version can be found today: ftp://ccl.osc.edu/pub/chemistry/software/SOURCES/C/viewmolorftp://ftp.ask.uni-karlsruhe.de/pub/education/chemistry/viewmol_ask.html)

Figure 4.Molecular Orbital of Benzene

Jörg-Rüdiger Hill (jxh@msi.com) was born in Berlin, Germany and holds a Ph.D. in theoretical chemistry. He works for a molecular modeling software company and is currently porting his molecular graphics program Viewmol to Linux. He has been running Linux since version 1.0.9. He much prefers the Southern Californian weather over that in Berlin.

Archive Index Issue Table of Contents

Advanced search

Qt GUI Toolkit

Eirik Eng

Issue #31, November 1996

This GUI toolkit makes porting graphics to multiple platforms a snap.

Developing applications with graphical user interfaces (GUI) takes time and can be hard work. Making these applications work across different operating systems can be even more complex. Traditionally, applications have been developed for one platform, and then large amounts of the code have been rewritten to port the application to other platforms. Multi-platform GUI toolkits have changed that procedure.

A multi-platform GUI toolkit makes it easier to port applications between platforms. Developing applications with a GUI toolkit is also considerably easier and a lot less work than using a window-system directly (e.g., X11 or Windows). The Qt toolkit is a multi-platform C++ GUI toolkit (class library) that has been developed over a 4 year period. The company Troll Tech AS was founded 2 1/2 years ago to secure future development of Qt.

As one of the Qt developers, I can give you an introduction to and overview of Qt. In the process, I'll throw in my 2 cents worth of general GUI-programming techniques.

The following sections can be found in this article:

- The Qt Story—background information about Qt
- Signals and Slots—Qt's object communication mechanism
- The Qt Paint Engine—drawing graphics with Qt.
- Qt Event Handling—how to get those user clicks and window system events in Qt
- Double-buffering—a well known and very useful GUI programming technique

- Making Your Own Software Components—how to code a new software building block
- Dialog Boxes—putting it all together and making it run
- Hints and Tips—my 2 cents worth of GUI-programming experience.

The Qt Story

The first publicly available version of Qt was released in May 1995. Version 0.98 was recently released (July 1996) and included complete source code for X11. Version 1.0 is scheduled for September 1996. Qt for X11 has a non-commercial license which grants any developer the right to use Qt to develop software for the free software community. The non-commercial version of Qt includes the full X11 source code. With this license, Troll Tech hopes to promote the development of high quality free software. Qt is an emulating GUI toolkit which allows programmers a choice between the Motif and the Windows look and feel. It implements its own widgets (user interface elements), and the X11 version of Qt is implemented directly on top of Xlib and uses neither Xt nor Motif. Practically all classes and member functions in Qt are documented. The documentation is available in HTML, postscript, text and as manual pages. The HTML version is fully cross-referenced with links to code examples. In addition, there is a tutorial for the Qt beginner. You can see the documentation on the web at http://www.troll.no/qt/.

Troll Tech has used Linux as its main development platform for over 2 years. All X11 development is done first on Linux, then the source is moved to other platforms for testing and porting. Qt currently runs under several variants of UNIX, Windows 95 and Windows NT.

Signals and Slots

Let's first look at the part of Qt that probably differs most from other GUI toolkits—the mechanism for object communication. One of the most feared and hackish aspects of GUI programming has always been the dreaded callback-function. In most toolkits, widgets have a pointer to a function for each action they trigger. Anyone who has worked with function pointers knows that this can get quite messy. Qt has approached the problem of communication between GUI objects (and other objects for that matter) in a totally new way. Qt introduces the concepts of signals and slots, that eliminate the need for pointers to functions, and provide a type-safe way to send arguments of any type. All Qt objects (classes that inherit from QObject or its descendants, e.g., QWidget) can contain any number of signals and slots. When an object changes its internal state in a way that might be interesting to the outside world, it emits a signal (not to be confused with UNIX interprocess signals), and then goes on happily minding its own business, never knowing or caring if anybody receives the signal. This important feature allows the object to be used as a true

software component. Slots are member functions that can be connected to signals. A slot does not know or care if it has a signal connected to it. Again, the object is isolated from the rest of the world, and can be used as a true software component. These two simple concepts make up a powerful component programming system. They may seem awkward when encountered for the first time, but they are a lot more intuitive and easier to both learn and use than the alternatives. Let's look at how signals and slots are specified in a class declaration. The following class is a stripped down version of the class shown in code Listing 3:

```
class PixmapRotator : public QWidget
{
    Q_OBJECT
public:
    PixmapRotator( QWidget *parent=0,
        const char *name=0 );
public slots:
    void  setAngle( int degrees );
signals:
    void  angleChanged( int );
private:
    int         ang;
};
```

Signals and slots are specified syntactically using C++ categories in the class declaration. This class defined above has a slot called *setAngle*. Slots are normal member functions and must have an access specifier. They are, as with other member functions, implemented by the programmer, and can be overloaded or virtual.

The PixmapRotator class has a single signal, *angleChanged*, which it emits when its angle has changed value. Signals are declared in the class declaration by the programmer but the implementation is generated automatically. To emit a signal, type:

```
emit signal( arguments )
```

The implementation of the slot looks like this:

```
void PixmapRotator::setAngle( int degrees )
{
// keep in range <-360, 360>
    degrees = degrees % 360;
// actual state change?
    if ( ang == degrees )
        return;
    ang   = degrees;            // a new angle
    emit angleChanged( ang );   // tell world
        ...
}
```

Note that *setAngle* only emits the signal, if the value actually changed (as the name of the signal implies). A signal should only be emitted when a state change has occurred.

To connect a signal to a slot the QObject static member function **connect** is used, for example:

```
connect( scrollBar, SIGNAL(valueChanged(int)),
                 rotator,   SLOT(setAngle(int)) );
```

Here the **QScrollBar** *scrollBar*'s signal *valueChanged* is connected to the **PixmapRotator** *rotator*'s slot *setAngle*. This statement assures that whenever the scrollbar changes its value (e.g., if the user clicks on one of its arrows) the angle of the **PixmapRotator** object will change accordingly. The two objects can interact without knowing about each other as long as a connection is set up by a third party.

As you can see, signals and slots can have arguments. The last argument(s) from a signal can be discarded, but otherwise the arguments must match for a connection to be made.

An arbitrary number of slots can be connected to a single signal and vice versa.

Technically, signals and slots are implemented using the Qt meta object compiler (moc). It parses C++ header files and generates C++ code necessary for Qt to handle signals and slots. The *signals*, *slots* and *emit* keywords are macros, so the compiler preprocessor changes or removes them.

Signals and slots are efficient. Of course they are not as fast as a direct function pointer call, but the difference is small. A signal triggering a slot has been measured to approximately 50 microseconds on a SPARC2.

The Qt Paint Engine.

Qt contains a device independent drawing engine, implemented in the class **QPainter**. **QPainter** is highly optimized and contains several caching mechanisms to speed up drawing. Under X11, it caches GCs (graphics contexts), which often make it faster than native X11 programs. **QPainter** contains all the functionality one would expect from a professional 2D graphics library.

The coordinate system of a **QPainter** can be transformed using the standard 2D transformations (translate, scale, rotate and shear). These transformations can be done directly or via a transformation matrix (QWMatrix), exactly as in postscript. Here is a small example taken from www.troll.no/qt showing the use of coordinate transformations:

```
void LJWidget::drawLJWheel( int x, int y, QPainter *p )
{
// set center point to 0,0
        p->translate( x, y );

// 24 point bold Times
```

```
        p->setFont(QFont("Times", 24, QFont::Bold));

// save graphics state
        p->save();

// full circle
        for( int i = 0 ; i < 360/15 ; i++ ) {

// rotate 15 degrees more
        p->rotate( 15 );

// draw rotated text
        p->drawText( 0, 0, "Linux" );
    }
    p->restore();       // restore graphics state
    p->setPen( green ); // green 1 pixel width pen
// draw unrotated text
    p->drawText( 0, 0, "Linux Journal" );
}
```

This member function draws a text “wheel” with the center given at a specified point. First the coordinate system is transformed so that the given point becomes the point (0,0) in the new coordinate system. Next a font is set, and the graphics state is saved. Then the coordinate system is rotated 15 degrees at a time, clockwise, and the text “Linux” is drawn to form a textual “wheel”. The graphics state is then restored , the pen set to a green pen and the text “Linux Journal” is displayed. Note that it is not strictly necessary to save the graphics state since we do a full 360 degree rotation. Saving the graphics state is strictly defensive programming—if we were to change the **for** loop, which is doing the rotation, we could still guarantee that the last text would be output horizontally.

Qt has a font-abstraction implemented in the **QFont** class. A font can be specified in terms of the font family, point size and several font attributes. If the specified font is not available, Qt uses the closest matching font.

The **drawLJWheel** function can be used to generate output on any device since it merely uses a pointer to a **QPainter**. It does not know what kind of device the painter is operating on. The function is put into a widget in code, see http://www.troll.no/qt. Running it produces the result shown in Figure 1.

Figure 1. LJ Widget Output

Support Classes

Qt also contains a set of general purpose classes and a number of collection-classes to ease the development of multi-platform applications. The hardest part of generating portable code has always been operating system dependent functions. Qt has platform independent support for these functions, such as time/date, files/directories and TCP/IP sockets. Sometimes it might be necessary to use the underlying window system resources directly, e.g., when interfacing with other libraries. Qt gives direct access to all low-level window IDs

and other resource IDs. Troll Tech has used this access to write a small widget that makes it possible to use OpenGL/mesa within a Qt widget.

Qt Event Handling

The structure of any GUI program is based on events. This basis is the main difference between GUI programming and non-GUI programming. A GUI program does not have "control" over the application; it merely waits for an event, does something as a response, and then waits for the next one.

A program typically sets up a top level widget to call the main event loop, which then dispatches events as they are received from the user or other parts of the system.

This model can be elegantly applied in an object-oriented language by using subclasses and reimplementation of virtual functions in the classical C++ event mechanism that is also used by Qt. The **QWidget** class contains one virtual function for each event type. A new type of widget is made by subclassing **QWidget** (or one of its descendants). You can simply reimplement an event function for each type of event you wish to receive. The event functions together with the Qt paint engine make up a powerful toolbox for creating custom widgets.

By far the most important event a widget receives is the paint event. It is called by the main event loop whenever the widget needs to draw a part of itself. Below is an example of a simple paint event, taken from code, http:// www.troll.no/qt:

```
void CustomWidget::paintEvent( QPaintEvent *e )
{
// necessary to draw?
    if ( rect.intersects( e->rect() )) {
        QPainter p;
        p.begin( this );         // paint this widget
        p.setBrush( color );     // fill color
        p.drawRect( rect );      // draw rectangle
        p.end();
    }
}
```

CustomWidget contains the member variable *rect* with type QRect, containing a rectangle with a one pixel black outline and filled with a color. Another member variable is *color* with type QColor, containing the color used to fill the rectangle.

First we check if the rectangle intersects the part of the widget that is to be updated. If it does, we instantiate a **QPainter**, open it on the widget, set its brush to the correct color and draw the rectangle (the default pen is one line thick and black).

All event functions take a pointer to an event object as their single argument. **QPaintEvent** contains the rectangular area of the widget that must be redrawn.

In the same widget we also receive resize events like this:

```
void CustomWidget::resizeEvent( QResizeEvent * )
{
// widget size - 20 pixel border
    rect = QRect( 20, 20, width() - 40,
        height() - 40 );
}
```

This event function sets the rectangle to be the size of the widget minus a 20 pixel border on all sides. It is never necessary to repaint a widget in a resize event since Qt always sends a paint event after the resize event when a widget has been resized.

CustomWidget also receives mouse press, move and release events like this:

```
void CustomWidget::mousePressEvent( QMouseEvent *e )
{
// left button click
    if ( e->button() == LeftButton &&
// on rectangle?
         rect.contains( e->pos() ) ) {
// set rectangle color to red
         color   = red;
// remember that it was clicked
         clicked = TRUE;
// repaint without erase
         repaint( FALSE );
    }
}
void CustomWidget::mouseMoveEvent(QMouseEvent *)
{
// clicked and first time?
    if ( clicked && color != yellow ) {
        color = yellow;    // set color to yellow
        repaint( FALSE );  // repaint without erase
    }
}
void CustomWidget::mouseReleaseEvent(QMouseEvent *e)
{
    if ( clicked ) {       // need to reset color
        color   = green;   // set color to green
        repaint( FALSE );  // repaint without erase
        clicked = FALSE;
    }
}
```

The mouse press event sets the rectangle color to red if the left mouse button is clicked inside the rectangle. When the mouse is moved after a click on the rectangle the color will change to yellow. Finally the color is reset to green when the mouse button is released.

The calls to repaint cause the entire widget to be redrawn. The FALSE argument instructs Qt not to erase the widget (fill it with the background color) before sending the paint event. We can use FALSE, because we know that **paintEvent** will draw a new rectangle covering the old one. Painting in this manner reduces flickering considerably; otherwise, double-buffering should be used.

The full widget code can be found in www.troll.no/qt . Running it produces the result shown in Figure 2. Custom Widget Output

Double-buffering

Flickering is a common problem in graphics programming. Some GUI programs do updating by clearing the area of a widget and then draw the different graphics elements. This process normally takes enough time for the eye to notice the clearing and drawing process. The widget flickers, the program looks unprofessional and fatigues the eyes of the users.

A technique called double-buffering can be used to solve this problem. A pixmap (i.e., pixel map—an off-screen memory segment used as if it were a part of the screen raster buffer) is used, and all drawing is done off-screen on this pixmap. The pixmap is then transferred to the screen in one lightning-fast operation. This pixmap transfer is normally so fast that on most systems it appears instantaneous to the human eye.

Sometimes a pixmap the size of the widget to update is used, in other cases, only certain parts of the widget are double-buffered. Which method will be most effective must be considered in each case.

Pixmaps often contain large amounts of data and are often slow to create and handle (in CPU-time). A good technique is to store the buffer pixmap as a part of the widget. When the widget needs to update itself (in Qt, whenever it receives a paint event), it simply copies the required part of the buffer pixmap to the part of the widget that must be repainted.

Often, it is useful to include a dirty flag as a part of the widget. All state changes (i.e., changes to member variables) that affect the visual appearance of the widget can then simply set this flag to TRUE telling the widget to repaint itself. The paint event function then checks the dirty flag, and updates the buffer pixmap before it updates the screen. This ensures that all widget painting code is in one place, making the widget easier to maintain and debug.

I've found this technique to be very useful and powerful and have used it on a variety of GUI-systems, as well as Qt.

Making Your Own Software Components

OK, now that we've looked at different parts of Qt, let's use it to build a custom-made software component that can display an image and rotate it by an angle. This widget should contain slots with instructions to let the user choose a file on disk, and it should have the ability to print the rotated image on a printer.

As a start we decide to give it the following signals and slots:

```
public slots:
    void  setAngle( int degrees );
    void  load();
    void  print();
signals:
    void  angleChanged( int );
    void  filePathChanged( const char * );
```

We can now set the rotation angle *setAngle*, let the user choose a new image file *load*, and print the image *print*. We choose to implement the functionality we need for the first version first. Later this component can be expanded to include slots like setPixmap(QPixmap) or setFilePath(QString).

The two signals tell the world about a change in the rotation angle (*angleChanged*) or the image file being displayed (*filePathChanged*).

Next, we include two member functions to fetch the angle and file path:

```
public:
    int   angle()          const { return ang;  }
    const char *filePath() const { return name; }
```

And we include the following member variables:

```
private:
    int         ang;
    QString     name;
    QPixmap     pix;
    QPrinter    printer;
    QFileDialog fileDlg;
    QPixmap     bufferPix;
    bool        dirty;
```

By setting these variables, we store an angle, file name, pixmap, printer and file selection dialog in the component. We want the widget to update itself smoothly and have decided to use the double-buffering technique, so we store a buffer pixmap. In addition, we have a dirty flag which is set when the widget needs to update the buffer pixmap. The combination of a buffer pixmap and a dirty flag is a very useful and powerful GUI technique that can be used in a large range of widgets.

Also, the widget has a paint event function and a resize event function, see code Listing 3 for the full class declaration.

Since we want to be able to paint to both the screen and a printer, we put the drawing code in a private member function that operates on a **QPainter**:

```
void PixmapRotator::paintRotatedPixmap(QPainter *p)
{
// need device width and height
    QPaintDeviceMetrics m( p->device() );
// center point
```

```
    p-$gt;translate( (m.width())/2,
                (m.height()) / 2 );
    p->rotate( ang );
    p->drawPixmap( - (pix.width())/2,
                - (pix.height())/2, pix );
}
```

First we fetch the metrics of the device the painter is operating on. We use the width and height of the device to put the center (0,0) of our coordinate system in the middle of the device. Next we rotate the coordinate system by the wanted angle and draw the pixmap with its center point at (0,0). In other words, the center point of the pixmap is put at the center point of the device.

The paint event function looks like this:

```
void PixmapRotator::paintEvent( QPaintEvent *e )
{
    if ( dirty ) {      // buffer needs update?
// same size as widget
        bufferPix.resize( size() );
// clear pixmap
        bufferPix.fill( backgroundColor() );
        QPainter p;
// paint on buffer pixmap
        p.begin( &bufferPix );
        paintRotatedPixmap( &p );
        p.end();
        dirty = FALSE;  // buffer now new and clean
    }                   // update exposed region:
    bitBlt( this, e->rect().topLeft(), &bufferPix,
                e->rect() );
}
```

If the widget is “dirty”, we need to update the buffer pixmap. We set its size to the size of the widget, clear it and call our local painting function. Don't forget to reset the dirty flag when a buffer pixmap has been updated.

Finally, we use the **QPaintEvent** pointer to find out which part of the widget must be updated and call *bitBlt*. *bitBlt* is a global function that can transfer data from one paint device to another as fast as possible. *bitBlt* is common GUI shorthand for “bit block transfer”.

With double-buffering and a dirty flag, the resize event function becomes trivial:

```
void PixmapRotator::resizeEvent( QResizeEvent *e )
{
    dirty = TRUE;               // need to redraw
}
```

Again, it is never necessary to repaint a widget in a resize event, since Qt automatically sends a paint event after the resize event.

With a common drawing function doing printing is also easy:

```
void PixmapRotator::print()
{
// opens printer dialog
    if ( printer.setup(this) ) {
```

```
        QPainter p;
        p.begin( &printer );    // paint on printer
        paintRotatedPixmap( &p );
        p.end();                // send job to printer
    }
}
```

First we let the user setup the printer, then we open a painter on that printer, and finally, call the drawing function.

Loading a new image takes a bit more code:

```
void PixmapRotator::load()
{
    QString newFile;
    QPixmap tmpPix;
    while ( TRUE ) {
// open file dialog
        if ( fileDlg.exec() != QDialog::Accepted )
            return;     // the user clicked cancel
// get the file path
        newFile = fileDlg.selectedFile();
// is it an image?
        if ( tmpPix.load( newFile ) )
            break;      // yes, break the loop
        QString s;      // build a message string
        s.sprintf("Could not load \"%s\"",
                newFile.data() );
// sorry!
        QMessageBox::message( "Error", s );
    }
    pix   = tmpPix;     // keep the pixmap
    name  = newFile;    // new file name
    emit filePathChanged( name );  // tell world
    dirty = TRUE;       // need to redraw
    repaint( FALSE );   // paint the whole widget
}
```

We set up a loop that opens the file dialog box. If the user has selected a file that cannot be loaded, we tell her and let her try again. If a valid file has been selected, we copy the new pixmap and the file path. Finally, we emit a signal to tell the world, mark the widget as dirty and repaint all of it (the FALSE argument means that Qt should not clear the widget before sending the paint event).

We have now made a new software component which can be connected to others through the signal/slot mechanism. See www.troll.no/qt for the full code of the PixmapRotator widget.

Dialog Boxes

Finally, let's use our component to put together a dialog box. Most GUI applications contain these boxes. Dialogs are windows with a number of widgets as children. A typical example of a dialog is an input form for a database with a text field for each database field.

Qt contains the standard widgets needed to build dialogs for most purposes. This custom built dialog box class has been taken from www.troll.no/qt

```
class RotateDialog : public QDialog
{
    Q_OBJECT
public:
    RotateDialog( QWidget *parent=0,
                const char *name=0 );
    void resizeEvent( QResizeEvent * );
private slots:
    void updateCaption();
private:
    QPushButton   *quit;
    QPushButton   *load;
    QPushButton   *print;
    QScrollBar    *scrollBar;
    QFrame        *frame;
    PixmapRotator *rotator;
};
```

The **RotateDialog** class inherits from **QDialog** and contains 3 pushbuttons, a scrollbar, a frame and the custom made pixmap rotator. The dialog only has three member functions. The constructor initializes the different widgets in the dialog, *resizeEvent* sets the position and size of the widgets, and the slot updates the caption text of the dialog.

Let's take a look at the constructor:

```
RotateDialog::RotateDialog( QWidget *parent, const char *name )
        : QDialog( parent, name )
{
    frame = new QFrame( this, "frame" );
    frame->setFrameStyle( QFrame::WinPanel |
                          QFrame::Sunken );
    rotator = new PixmapRotator("this, rotator");
    rotator->raise();  // put it in front of frame
    quit = new QPushButton("Quit", this, "quit");
    quit->setFont(QFont("Times", 14,
                QFont::Bold));
    load = new QPushButton("Load", this, "load");
    load->setFont( quit->font() );
    print = new QPushButton("Print", this,
                        "print");
    print->setFont( quit->font() );
    scrollBar = new QScrollBar(QScrollBar::Horizontal,
                        this, "scrollBar" );
    scrollBar->setRange( -180, 180 );
    scrollBar->setSteps( 1, 10 );
    scrollBar->setValue( 0 );
    connect( quit, SIGNAL(clicked()), qApp,
                SLOT(quit()) );
    connect( load, SIGNAL(clicked()), rotator,
                SLOT(load()) );
    connect( print, SIGNAL(clicked()), rotator,
                SLOT(print()) );
    connect( scrollBar, SIGNAL(valueChanged(int)),
             rotator  , SLOT(setAngle(int)) );
    connect( rotator, SIGNAL(angleChanged(int)),
                SLOT(updateCaption()) );
    connect( rotator,
                SIGNAL(filePathChanged(const char *)),,
                SLOT(updateCaption()) );
    setMinimumSize( 200, 200 );
}
```

The different widgets have now been instantiated and initialized. We also want to put a frame around the pixmap rotator, so it is raised (popped to the front of the window stack) in order to make sure it is in front of the frame.

The scroll bar is set up to represent a value in the range [-180,180] with line and page steps set to 1 and 10 respectively. A line step is used when the user clicks on a scroll bar arrow, page step when the user clicks between the arrows and the scroll bar slider.

Then the different widgets are connected. The quit pushbutton is connected to the applications quit slot (qApp is a pointer to Qt's application object). The load and print pushbuttons are connected to their respective slots in the pixmap rotator, and the scrollbar is connected to the pixmap rotators angle value.

Next we connect the rotator's signals to the private slot *updateCaption*. We are here using a connect function that only takes three arguments, where the *this* pointer is implicit as the receiver. Note that the slot we connect to has fewer arguments than the signal. All or some of the last arguments of a signal can always be discarded in this way, if we are not interested in receiving them in a slot.

Note how we use the standard widgets to control our custom made widget. PixmapRotator is a new software component which can be plugged into many different standard interface controls. We could easily have added a menu in addition by plugging it into the rotator's slots. Keyboard accelerators or a text field, used to enter a numerical value for the angle, could likewise have been added without a single change to PixmapRotator.

Finally, we tell Qt that this widget should never be allowed to have a size smaller than 200x200 pixels. Note that the class does not have a destructor. Child widgets are always deleted by Qt when the parent is deleted.

There is no interactive dialog builder for Qt at the time of this writing (July 1996), but there is one in the works that will probably be ready by the time you read this article. Check Troll Tech's home page for details (http://www.troll.no/). The resize event is implemented as follows:

```
const int border     = 10;
const int spacing    = 10;
const int buttonH    = 25;
const int buttonW    = 50;
const int scrollBarH = 15;
void RotateDialog::resizeEvent( QResizeEvent * )
{
    quit->setGeometry(border, border, buttonW,
                      buttonH );
    load->setGeometry((width() - buttonW)/2,
                      border, buttonW, buttonH );
    print->setGeometry(width() - buttonW - border,
                      border, buttonW, buttonH );
    scrollBar->setGeometry( border,
                quit->y() + quit->height() + spacing,
                width() - border*2, scrollBarH );
    int frameTop = scrollBar->y() +
                scrollBar->height() + spacing;
                frame->setGeometry( border, frameTop,
                width() - border*2,
```

```
                height() - frameTop - border );
    rotator->setGeometry( frame->x() + 2,
                frame->y() + 2, frame->width() - 4,
                frame->height() - 4 );
}
```

Each widget is moved and resized according to the dialogs width and height. The three buttons are placed 10 pixels from the top of the dialog, one on each side and one in the middle. The scrollbar is placed right beneath them, followed by the frame. The rotator is put inside of the frame.

Writing code like the above is not difficult, but it's not always easy to read. A geometry manager solves resizing of dialogs in a more elegant way. Qt's geometry manager is currently under internal testing at Troll Tech and will soon be added to the toolkit. The slot is implemented like this:

```
void RotateDialog::updateCaption()
{
    QString s;
// we do not want the full path
    QFileInfo fi( rotator->filePath() );
    s = fi.fileName();          // only the filename
    if ( rotator->angle() != 0 ) {   // rotated?
        s += " rotated ";
        QString num;
// convert number to string
        num.setNum( rotator->angle() );
        s += num;
        s += "degrees";
    }
    setCaption( s );    // set dialog caption
}
```

We build a message string by using the image file name and its rotation angle. The string is then used as the dialog's caption.

See www.troll.no/qt for the full code. When run, the user can press on the load button and Qt's standard file dialog will pop up. If the print button is clicked, the standard print dialog pops up letting the user choose if the output should go to a file or to a printer. Under X, Qt generates postscript printer output; under Windows, the Windows printer driver system is used.

In Figure 3 you can see a screen shot of the RotateDialog together with the print dialog.

Hints and Tips

If you are new to GUI programming, you might find that your first widget comes up blank on the screen or not at all. There are two very common reasons for this to happen.

First of all, a widget's window on the screen can be overwritten at any time by the window system. The window system does not store the contents for you, it just calls the widget's paint event function when the widget needs to be

refreshed. Thus, if you draw on the widget but the paint event function doesn't reproduce the drawing exactly, you might not see any effect on the widget. Generally, the best method is to do all your drawing in the paint event function, and just instruct the widget to do a repaint whenever its state changes.

Secondly, when you create a widget, it is not visible. In Qt, you have to call the *show* member function to make widgets visible; other toolkits have a similar function.

Finally, I would like to mention a few points I consider important when designing and implementing GUI programs:

- Keep it simple. When you build new widgets and dialogs try to keep their interfaces as small and elegant as possible. Don't add a lot of functions that might be nice to have in the future.
- Don't crowd the screen. Try to keep your dialogs as intuitive and minimalistic as possible. A dialog should be both functional and pleasing to the eye.
- Use double-buffering. At least when all or parts of a widget will change significantly over time. Programs that don't flicker look a lot more professional than the ones that do.
- Cache like crazy. If there are parts of your program that do time-consuming operations, e.g., generation of pixmaps, save the result so that you don't have to do the same operations over and over.
- Keep your member variables private—this is good object-oriented practice. In GUI programming it is even more important. A change in a variable often means that you have to update the screen. If you set variables via member functions, you can guarantee that the screen is always up-to-date.
- Put your drawing code in one place—preferably in a single member function. Bugs in the relationship between the values in your member variables and what the widget displays are then located in one place. (You will be glad you did.)
- Use standard types as arguments to signals and slots. If your signal contains an *int*, it can be connected to a large number of slots. If it contains an argument of the type *MyNumber*, the widget or dialog will not be as useful as a component.

Where to Find Qt

The Qt source and binary versions for several platforms (including Linux of course) can be downloaded from the net at ftp://ftp.troll.no/qt/.

Eirik Eng (iriken@troll.no) is a co-founder of Troll Tech AS, and works there as a developer. He has a siv.ing. (M.S.) degree from the Norwegian Institute of Technology and has worked with GUIs and OOP since 1991. His main hobby is office gardening (perfect for people who spend a lot of time at the office). This year he is especially proud of his 4 foot high (and still growing) eggplants and his tamarind tree.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

Graphics Tools for Linux

Michael J. Hammel

Issue #31, November 1996

You CAN do progressional graphic art on Linux systems. The tools are available, the documentation is growing, and the user communities offer a wealth of information and help.

In the Beginning...

Ever since I wrote my first basic program to peek and poke my way through a series of animated block graphics on a TRS-80 Model 1, I have been intrigued by the use of computers for graphics. I learned every new language by modifying a graphics program I had written in some other language. In high school and my early days of college these programs were simple games—later I learned of the Mandelbrot and Julia fractal algorithms. These algorithms were relatively simple to program and so became my tool for learning new languages, environments and, more importantly, graphics systems.

Over the years this fascination would wax and wane. Not surprisingly, my interest would peak when some new movie with the latest, greatest special effects was released. *Star Wars* came out when I got my hands on the TRS-80, so naturally I had X-Wing fighters evading a simulated crosshair radar written in BASIC, albeit by the time the radar locked, the pilot of the X Wing had probably died from old age.

Late in 1995, a co-worker and I began to work on our company's web pages. I had done my own pages, but they lacked good graphics. He was convinced he could do the graphics while I did the HTML. His enthusiasm became annoying when he began to laud the merits of MS-Windows-based graphic tools like Adobe Photoshop, and went on about how Linux, and Unix in general, lacked any such tools. Since I knew most of the special effects I was seeing in movies were coming from Silicon Graphics or other Unix systems, I knew comparable tools must be available for Linux.

And then came *Toy Story*. I was fascinated. I was enthralled. I was completely hooked. I set out to find Linux tools that I could use to update my web pages and to show the world what could be done on a Linux box.

This article introduces the tools I found, tells how the information I gathered grew into the Linux Graphics mini-HOWTO (see Resources box), and explores how the graphics arena is evolving on Linux. I assume the reader is a graphics beginner, and I focus on tools currently available. I touch only briefly on the programming environments and libraries available to do graphics, since these require a more in-depth discussion; however, I also don't want to rehash the Linux Graphics mini-HOWTO. After reading this article, you can check out the mini-HOWTO to get more details.

Graphics Basics

When you get right down to it, there are really only four types of tools available for graphics on Linux:

> **Viewing** - tools which display images but don't change them
>
> **Creation** - tools which create images
>
> **Manipulation** - tools which change existing images
>
> **Conversion** - tools which convert images to new formats

Various tools are available for each of these types. Many tools fit into more than one category. The dividing line between categories is not always clear, but the use of categories helped in organizing the Graphics mini-HOWTO.

Creation tools also provide one of two basic functions: they draw or they render images. *Drawing* tools provide the ability to interactively create shapes, called primitives, such as boxes, circles, cones or torii. Creation tools also include *paint* programs. Rendering tools build images based on model information, i.e., information that describes the shapes of primitives and their relationships in a *scene*. The simplest way to distinguish between the two is that drawing programs generally deal with 2D images, and rendering tools deal with 3D images. This is an oversimplification but it will suffice for this article.

Images are pictures, basically, which come in a number of formats. GIF, JPEG, PNG and TGA are some of the more common static image formats, but there are literally hundreds of others. No one format is the standard, so there isn't one format that every tool supports. GIF and JPEG are the most widely supported formats for web browsers, but you can expect to see PNG support in

many browsers and other tools soon. Which format you use depends on the tools you use and the way you intend to use the images. For example, GIF isn't good for large posters, because it lacks support for more than 256 colors per image. TGA, on the other hand, provides a large range of colors but isn't supported by web browsers.

Static images, like photographs, are single, independent pictures. Animated images can be strings of static images or images created on the fly by programs written using special languages or programming libraries. A lot of interest in computer graphics has been generated by the use of animated images; however, it is important for new users to understand static images before moving into animation. For that reason, animation tools are not discussed in this article.

A number of programming libraries are available for use in supporting the various graphics formats and certain types of primitives, functions and algorithms. Libraries already exist for TIFF, JPEG, PNG, and many other image formats, so programmers don't have to reinvent the wheel with each new program they write. Also available are a number of languages and programming interfaces for creating 2D and 3D graphics and runtime animation, including VRML, OpenGL and PHIGS.

Figure 1. AC3D Screen with Imported DXF Model for a Sailplane

Figure 2. GIMP Windows

Viewing Tools

In my early days on Compuserve, GIF files were abundant and were viewed with programs like picem, which had few capabilities beyond displaying the image. The majority of tools available for Linux today do much more than view images.

A few tools are still around that do little more than view images. *xwud* (X Window Undump) is a program delivered with the base set of X11 clients for displaying files in the X-Windows Dump format created with a companion tool called *xwd* (X Window Dump). However, neither of these tools is feature-rich, and I use them only when I have no other way to do a window screen capture under X.

Creation Tools

Most of the graphics tools available for Linux fall into the category of creation tools and allow you to draw or render images, interactively or through some form of scene description language. One of the most widely known of these is XPaint, a tool similar in style to the old MacPaint or MS Paint without as many

features. Recent updates to the program have helped by providing additional options, but the rapid growth of tools like Photoshop have made the XPaint style tools less attractive. XPaint does provide some basic functions useful for creating texture maps for other tools. For example, one nice feature of XPaint is the ability to edit individual bits in an image.

A disadvantage to XPaint and similar tools is that it is not designed to deal with pixels as groups or to allow the modification of pixels by hue, saturation or intensity. Thus, XPaint gives you a canvas on which to paint, but not the ability to mix your colors before you paint or to blend colors (or shades of gray) on the canvas itself.

The first tool I discovered after seeing *Toy Story* was POV-Ray, the Persistence of Vision Raytracer. POV-Ray, another form of creation tool, is a 3D photorealistic renderer (not a drawing program). It works by reading in a text file that describes the scene to render. The scene describes objects made up of primitives (boxes, cones, torii, triangles and/or combinations of these and other shapes), descriptions of the textures to place on these objects, the lighting in and around the scene and the point of view of the "camera". Programs that do raytracing use one particular method for determining how light acts on the objects, and thus, how the objects will look to the viewer. This method traces the path of light beams through the scene; how this is done is really unimportant to all but the most experienced raytracing artist. Other methods for computing the way light behaves in a scene are radiosity and the REYES algorithm, the method used in making *Toy Story*. Tools using these methods are discussed in the Linux Graphics mini-HOWTO.

The disadvantage to tools like POV-Ray is the lack of user interaction. POV-Ray is a sort of batch processor—it processes a file and produces output. No feature of POV-Ray allows you to create input files. Input files must be created either by hand or by the use of a modeller. Modellers, like CAD systems, use wireframe representations of the objects in their scenes. The use of wireframes helps to visualize the scene from various points of view without the overhead of adding the textures (which generally take a long time to compute, even on fairly fast systems).

There are three modellers available for Linux: SCED, Midnight Modeller, and AC3D. SCED, available as source, is portable across multiple Unix platforms and makes use of the Athena widget set. I found its interface a little difficult to use, but the ability to group objects, called CSG (Constructive Solid Geometry), was very nice. Midnight Modeller is a tool ported from the MS-Windows environment. The author does not release source code, so the program is available only in binary format. The interface is very CAD-like, but the colors tend to be harsh, making the program difficult to use. The window features,

such as menus and dialog boxes, are very DOS-like, which I find a distraction on an X-Window system. Neither of these is the quality of some of the better modellers for MS-Windows.

AC3D is a new tool with a very nice 3D (Motif-like) interface that uses front and side view windows and can do some real-time rendering. I came across this tool just as I was finishing this article, so I haven't had time to give it a proper review. Of the three, it appears to be the most user-friendly with the most intuitive interface. AC3D is shareware for Linux (about $15 US) and comes in binary format only at this time.

When I first started gathering information about graphics tools, I was primarily using POV-Ray, as I had found the POV-Ray web site with its large collection of tools. However, the tools were mostly DOS/Windows binaries or written by DOS/Windows users and included the C source. Since I wasn't running DOS or Windows and there was little information on which tools would work on Unix, I started the Unix Graphics Utilities web page (this page is not exclusively for Linux). I used the information I gathered as the basis for and the incentive to write the Linux Graphics mini-HOWTO, since all my testing was done using Linux systems.

Many other tools can fit into the creation category. For example:

- BMRT is a tool that conforms to the Renderman specification, put out by Pixar. PRMan, which was used to create *Toy Story*, is another Renderman-compliant tool (although it is not available on Linux systems).
- Rayshade - another raytracing utility.
- TGIF - a much more sophisticated version of XPaint that allows for the organizing of primitives into groups and layers, similar to the old MacDraw tool for the Macintosh.
- HF-Lab is a tool for creating 3D landscapes.

Again, you can get more detail on these tools from the Linux Graphics mini-HOWTO, including where to obtain the tools.

Figure 3. The POV-Ray Home Page

Manipulation Tools

A manipulation tool that has gained a rather large following in a fairly short time is the GIMP, Generalized Image Manipulation Program. The GIMP is being developed primarily by Peter Mattis and Spencer Kimball at UC Berkeley. This tool, designed as a Unix-based counterpart to the Photoshop-caliber tools available on other operating systems, has a limited set of base features at this

time; however, the design includes the ability to add plug-ins which expand the features of the program. The result of this design philosophy has been a very large collection of plug-ins from a growing developer base. I've added my own plug-in, the Sparkle plug-in based on John Beale's Sparkle utility. There are tutorials and tips and tricks web pages (see the Resources box) as well as a relatively strong effort to organize the registration, format and distribution of the plug-ins.

The GIMP provides both user interaction and the ability to combine images through various blending techniques (nearly all of which are done with plug-ins). By combining two images, say by subtracting one from the other, it is possible to take 2D black and white text and turn it into a full-color 3D image resembling anything from a plasma field to a glowing, tube-shaped jelly. The GIMP is a major step beyond the basic box, circle and fill capabilities of XPaint.

Combining images is done through what is known as “Channel Operations” (commonly referred to as channel ops). These are operations on individual pixels based on the intensities (brightness) or color of the pixel and/or surrounding pixels. In short, a channel operation is the blending of colors in one or more images. Channel ops are what make the GIMP a more attractive tool than XPaint to the graphics artist. (Note that the basic capabilities of XPaint are not all implemented in the GIMP, so XPaint is still useful.)

The disadvantages of the GIMP are twofold: it is in early development, and its GUI base is in the process of changing. The reliance on Motif is being removed and a new toolkit, gtk (the GIMP Toolkit), has been developed. This toolkit is in the early stages of testing. The base GIMP features and most of the plug-ins lack any detailed documentation. However, these problems are relatively minor and are being addressed. The GIMP's advantages are that, at this time, it is the only freely available program of this caliber, and it has a large support base from the plug-in development and user communities.

XV is another tool with wide audience appeal. XV, by John Bradley, is a shareware program that supports a large number of file formats for reading (displaying) and writing. A large printed document is available to those who register their copy of the software. Although the number of algorithms for manipulating images that XV supports is smaller than the plug-in base for the GIMP, the ability to control the images' colors is much greater in XV. You can control the hue, saturation and RGB (red, green and blue) levels interactively in more ways than the GIMP provides (although there is at least one GIMP plug-in that allows this control to some degree).

From my point of view, the main difference between the GIMP and XV is that the latter is designed for scientific image processing and the former for artists.

I'm sure there are a few people who would argue, But I've seen television interviews of JPL and NASA employees who were using XV to display planetary images.

Figure 4. Image Produced Using GIMP Plug-ins

Animation Tools

A couple of animation tools are worth mentioning. The first is mpeg_play, a tool for viewing MPEG-formated animation files. Its companion, mpeg_encode, creates MPEG files. It can also display XING files, which uses a variation of MPEG encoding. However, mpeg_play doesn't manipulate the image in a way that can be saved back to file—it modifies only the displayed image.

Xanim is similar to mpeg_play, but supports a much wider set of input file types and capabilities for resizing images on the fly.

Conversion Tools

One of the problems you'll encounter when working with graphics tools is support for specific image file formats. For example, while working with the GIMP, you may want to work with TGA images, since TGA provides as many as 24 bits of color while GIF formats only provide 8 bits of color. Having a greater range of colors gives a very smooth blending of colors when you add, subtract, blur or otherwise manipulate the images with the GIMP. However, if the image you are creating is destined for a web page, you need to convert it to GIF or JPEG format. If the GIMP didn't provide a way of doing this (which it does, but this is just an example), you would need a tool for converting the image formats.

PBMPlus and NetPBM are a set of tools for converting images between various formats. PBMPlus tools, originally written by Jef Poskanzer, take one image file of a particular format as input and convert it to an intermediate format called PPM (Portable Pixmap). Another PBMPlus tool is then used to convert the PPM format to the target format. For example, to go from TGA to GIF you might use

```
tgatoppm input.tga | ppmquant 256 | \
    ppmtogif -interlace > output.gif
```

The ppmquant is another type of tool in PBMPlus used not to convert the file to another format, but to alter the image in the same way. In the case of ppmquant, the input image is quantified down from some large number of colors to 256. Doing so makes it possible to reduce the size of the data file, and depending on the image, may not alter the appearance so much as to make it unusable. You may choose to reduce the number of colors because computers

can record more subtleties of color than the human eye can distinguish, and on a web page, small data files make for faster loading.

PBMPlus tools work by processing data from standard input and writing to standard output, which allows the user to string a collection of tools together in a series of pipes (as in the above example). All the tools are meant for command line use, i.e., no graphical interface is available for these tools.

NetPBM is a later incarnation of PBMPlus, after Jef stopped working on it and development was picked up by another team of developers. However, I believe Jef has once again returned to working on PBMPlus.

Another conversion tool is ImageMagick. This set of tools has a graphical front end, but they can also be used as command line tools, similar to the PBMPlus tools. I haven't used these tools much, but I know they have a number of supporters.

Programming Interfaces

Programming interfaces aren't really end-user tools, but I'd like to mention a few, since these interfaces are likely to be the basis of the next generation of end-user tools.

One of the most popular and widespread programming interfaces is OpenGL, from Silicon Graphics. OpenGL is a hardware-independent programming interface that provides commands and operations to produce 3D, interactive applications. This technology is fairly new and requires support in your X server to make use of hardware accelerations provided by video adapters. Although the hardware support is not required in order to use OpenGL, without it or a relatively fast computer, OpenGL applications will run a bit sluggishly. A freely available version of OpenGL, known as MesaGL, implements most (if not all) of the OpenGL command set. Also, OpenGL has a very low level interface, much like Xlib is to X-Windows; therefore, higher level toolkits, such as GLUT or aux, are more appropriate for writing applications. [For more information on OpenGL/Mesa, see *Linux Goes 3D* elsewhere in this issue—ED.]

VRML is the Virtual Reality Modeling Language, designed to provide a 3D interface using a markup language similar to HTML. The current VRML 2.0 specification is based on Silicon Graphics Moving Worlds specification. Only a few VRML-capable browsers are available; however, there is strong industry interest in this language, and I expect many browsers will be supporting it in the near future. A recent report by C|Net cited both Netscape 3.0 and MS Internet Explorer as supporting VRML to some degree.

Java is not specifically graphics-oriented; its real purpose is to provide a platform-independent programming language. Java has gained its early reputation by virtue of a set of graphics applications and toolkits available via the World Wide Web. My own web pages make use of a scripting language known as JavaScript (from Netscape).

A number of other programming libraries have been announced on the comp.os.linux.announce newsgroup:

- SRGP - the Simple Raster Graphics Package, a toolkit for use with *Computer Graphics: Principles and Practice*, 2nd Edition, by Foley, van Dam, Feiner and Hughes.
- lib3d - an X-based 3D rendering library
- EZWGL - a Motif-like widget library that includes some GL support
- YGL - emulates SGI's GL (Graphics Language) under X11

Web Pages

As I mentioned earlier in this article, web browsers support the GIF image format, and some JPEG; most now support both formats. In choosing one of these formats, you should consider the following:

- How big is the image in bytes?
- Do you want the outline of the image to be visible or do you want the background to show through?
- Do you want the image to fade into view while the surrounding text is displayed, or do you want the image displayed before the text?

The answers to these questions help determine which image format to use. JPEG will usually compress images without losing too much of the information and appearance of the image, than GIF. Therefore, if your images take up a lot of disk space (a problem since my ISP doesn't provide much disk space for private web pages), or you don't want to force your readers to download large images, JPEG images would probably be better. However, if your image has a background you do not want displayed but you do want the background image or color of the page to show through, you need to use GIF-formatted images, as JPEG does not provide transparency in images.

If you want your images to fade into view as the text around it is displayed, you need to use GIF images. This trick is called interlacing, and JPEG does not support it. If you have large images (not necessarily in bytes, but ones that take up a lot of screen space), it is more user-friendly to use interlaced GIF images.

Once you've decided on the type of images you'll use, you can start creating them. There are many tricks and tips, but for starters I suggest you get hold of the following tools:

- XV
- XPaint
- The GIMP
- PBMPlus/NetPBM

You can use these tools, along with a CD-ROM of images (readily available from computer software stores) to create your own little web world. When you get adventurous, you might try adding some complex 3D images using POV-Ray. If you're interested in using special fonts with the GIMP, I suggest looking for packages containing Adobe Type 1 fonts. Once you have these you should check out the typ1inst package, which will allow you to install the fonts for use with your X server.

The Future

As you can see, there is really too much information about graphics tools to cover in a single article. Nearly all these tools are still evolving, adding new features and capabilities through the combined efforts of many people. Tools based on the more powerful programming languages, such as Java and OpenGL, are not far off, and it is only a matter of time before commercial versions begin to appear. Support for live video capture is available for some X servers and support for hardware accelerations and video capture boards is forthcoming. I intend to cover all of these in the Linux Graphics mini-HOWTO as they evolve.

You **can** do professional graphic art on Linux systems. The tools are available, the documentation is growing, and the user communities offer a wealth of information and help. And what's best of all—you can now find the tools you need using a single reference: The Linux Graphics mini-HOWTO.

Michael J. Hammel (mjhammel@csn.net) is a transient software engineer with a background in everything from data communications to GUI development to Interactive Cable systems—all based in Unix. His interests outside of computers include 5K/10K races, skiing, Thai food and gardening. He suggests if you have any serious interest in finding out more about him, you visit his Home Pages at www.csn.net/~mjhammel. You'll find out more there than you really wanted to know. He also requests that any commercial vendors of graphics systems contact him, as he'd like to include these in the mini-HOWTO in the future.

Archive Index Issue Table of Contents

Advanced search

OpenGL Programming on Linux

Vincent S. Cojot

Issue #31, November 1996

Use Linux and get good grades? Read on to learn how Linux helped one student with a major school project.

This article is not intended to be an OpenGL tutorial or introduction. There are people far more competent in this area than myself, and they have written a number of articles and even books about this subject. Also, even though the project discussed in this article was written and built mostly using Xinside's OpenGL, it is not my intention to discuss the superiority of any of the Linux OpenGL ports or implementations over another. This article tells a cool story about using Linux, and I thought it was worth contributing to the Linux community.

Introduction

“And, thus,” the professor concluded, “Instead of working on perfecting our home-made ray-tracer—something we have been doing for years—this semester, we'll start something new. You will have to build a 3D, network-capable tank game using OpenGL. You'll be in teams of two or three students. For that, you'll use the RS/6000 workstations we have here, and we'll give you an introduction to OpenGL.”

My ears! They could not believe what I had just heard. As a student in Computer Engineering at the Polytechnical School of Montréal and a computer graphics fan with a good background in ray-tracing, I had been waiting for years to be advanced enough in my studies to be able to take that course in advanced computers graphics. That semester I had finally been able to take the difficult 4th year course, and I had just heard, to my immense disappointment, that instead of working with a ray-tracer and producing high-quality ray-traced pictures, I would have to work on OpenGL. My morale was not high, and fear was making its way along my stomach as I realized I knew a lot less about OpenGL than about ray-tracing. But as one LinuxDoom aficionado would put it:

“Armed solely with my Linux Box and my OpenGL Beta product, I plunged into the hostile mass of GL intrinsics, prepared to fight with every last GLfloat variable I had.”

What is OpenGL?

More seriously, OpenGL is a graphics library designed from the start as a hardware-independent interface to be implemented on many different platforms. It uses a client-server approach, similar to the X client-server approach, to provide display of graphics primitives on the chosen windowing system. The server sends commands to the client, and the client displays them.

On X-capable Unix workstations, OpenGL has an extension to the X server named GLX. You can run your OpenGL program on one computer and display it on another, but it requires that the server machine has the needed OpenGL libraries and that the client has the GLX extension. Since those two packages usually come together, this means that both the server and client must be “OpenGL-capable”.

In short, OpenGL is capable of displaying simple geometric objects, showing orthogonal and perspective projections, performing back face removal, doing shading and anti-aliasing, and applying textures to objects. If you want to do something complex—like display a car or a plane—you have to build those objects yourself, and use OpenGL to render them the way you like.

On Linux, to my knowledge, you have the choice of several commercial implementations and one free implementation:

- Xinside's and Metrolink's OpenGL ports for Linux, each of which requires that you install its own X server to provide the GLX extension and generally higher performance.
- Portable Graphics, whose product runs directly on XFree86.
- Brian Paul's Mesa library, which is GPLed and available for free, but has no GLX extension. It's impressive and affordable.

My personal experience was that the product I was using (Xinside's OpenGL second beta, and later, the final product, which was even faster) was of very high quality. It was faster and more compatible than Mesa. Speaking about commercial applications running on a free operating system is a sensitive and slippery issue, especially when there are freely available equivalents (Mesa) and even more so when you happen to find yourself very (or at least more) satisfied by a commercial tool. I found Mesa to be an impressive piece of software, but Xinside's OpenGL beta was noticeably faster and more OpenGL-compatible, since it is a true OpenGL implementation.

Back to a Dearly Loved Linux Box

So, here I was, a few days later, in front of an RS/6000 workstation, writing the first few lines of code of that soon-to-be tank game and wondering if it was going to run on my Linux box. You see, I had subscribed to Xinside's OpenGL beta program a few months before as a means to remotely run OpenInventor from my Linux box, and thus, I found myself with an OpenGL-capable Linux computer. Later that same day I went home—while my Linux box was retrieving the sample code by FTP—and got ready to compile it under Linux.

The project we were building was using a freely available auxiliary library named libaux. “Fine,” I thought, and I FTPed its source code from the RS/6000 lab and compiled it on my Linux box. It's also available from ftp.sgi.com under the OpenGL sub-directory, along with all the examples from the OpenGL programming guide. With a lot of hope and increasing excitment I got ready to start the sample code and...it crashed, generating a panic file and killing the X server.

The team later figured that this problem was caused by a small bug in the Beta OpenGL release I was using which caused it to misbehave when using a color-indexed color mode and single-buffering. The program, however, ran fine as soon as I switched to use RGBA (for Red, Green, Blue and Alpha) color mode—it even ran slightly faster than on the older RS/6000 workstations we were using!

Granted, those RS/6000 were basic entry-level workstations, and their age (about two years), combined with poor 3D hardware accelerated video cards, proved they were no real match for my P133 with its Matrox Millennium (although OpenGL on Linux only provided software 3D acceleration). For someone who has been used to “This hot stuff runs on workstations—W-o-r-k-S-t-a-t-i-o-n-s—don't even think about running it on your home PC!” this OpenGL on Linux experience was like a dream come true.

To Port or Not to Port, That Was No Question

The days went by, and we started incorporating more and more code into the project. My team-members had more course work than I did, so I found myself leading the team—writing most of the code in the first part of the project and all of it in the second and third parts. Of course, I was writing it all on Linux—but always verifying later that it ran on the RS/6000 workstations (Murphy, you know?).

Of course, that did not go unnoticed, and some of the students in the class started exploring ways to build and develop their own projects at home on their PCs using NT or that OS-with-an-expiration-date-in-its-name (Windows 95). Others followed my advice that it would probably be better to use a Unix

because of portability problems (I thought...er envisioned...er imagined that the Win32 API could be quite different from that of most Unices) and got Mesa running. After all, if you have the choice and if you can do the same things you do at your university at home, would you rather spend nights in a freezing-cold computer lab with armless wooden chairs or work on your home computer?

Problems started to appear just a few weeks after that when we were required to implement and use a timer within the game. That was the first blow for the NT/95 people because, unless you're familiar with the Windows API or have some sample source code, changing Unix's gettimeofday() to a Windows API call is not trivial. After all, if your virtual tank is going at 10 m/s, it should do so no matter what hardware you have, be it a 16 CPU SGI workstation or a poor 80486. Some people got tired of putting **#ifdef**s and **#ifndef**s in their code and decided to spend nights in the lab instead.

Then came the network daemon. The idea (mostly at my suggestion) was that the game client running on a particular workstation would **fork()** a daemon at initialization. The daemon would share one or more memory segments with the client and would have the task of listening on certain ports for broadcast messages sent by other possible network players. Needless to say, these Unix intrinsics marked the end of the Windows port; even if you could run a part of the 3D engine on Win32, you'd still have to do all the network and final debugging on the RS/6000 workstations at school.

But during all this time there was at least one happy Linux user who did not change a single line of code when sending it from his home Linux box to the Risc workstations. And the only time he actually had to put an **#ifdef** was when the endianness difference between the Pentium and the RS/6000 processor started to show in the byte ordering of the TARGA files he was loading and using for textures. Rumour even has it that he debugged his network code without actually entering the computer lab: in the darkest hours of the night he used two workstations to run his program on and exported the display to his Linux box (which was slow, but functional enough to track down some bugs).

Performance: Hardware and Software

Speaking about performance in OpenGL is, for those of us who don't use a middle to high-end SGI workstation at home or at work, about as important as speaking about OpenGL itself. In our case, around the middle of the semester, it became obvious to both professors and students alike that the RS/6000 workstations we were using were not fast enough for what we were doing with them.

Eventually we switched to another lab of RS/6000 workstations which belonged to the Mechanical Engineering Department. People there ran CATIA—like

AutoCad but with ten times the features and the memory requirements. Those workstations were still not inherently faster than a good Linux Pentium PC with enough RAM; most tests, gcc, xv, etc, showed my P133 was about 50-60% faster doing generic operations. But their hardware-accelerated OpenGL graphics allowed my game to run on them at 25 frames per second with 512x384 pixels. By comparison, I was getting a maximum of only 9-10 frames per second with 320x240 pixels on my Linux box, where OpenGL rendering was done by software alone on the main CPU and FPU.

The program still ran, and it was fast enough to allow me to work out most of the bugs and implement new features, but I would personally have enjoyed it a lot more if the Linux OpenGL port I was using had been able to take advantage of the 3D features on my video card to make my programs run even faster. On my end, I tried removing as much un-optimized stuff as possible from the game's main loop to make it run as fast as possible on all platforms.

Here are some stats about the project:

- Lines of code: about 7600 (game and daemon) + 900 (explosions renderer)
- Number of textures and ray-traced rendered explosions: 34
- Number of different object lists used: 38
- Number of possible network players or automated opponents: 20
- Features only available on Linux: basic sound!
- Time spent pulling our hair out on that game: around 200-250 man-hours

Conclusion

The project, for all the work-teams in our course, is now finished as far as it involved students working on a programming project whose results professors would evaluate. I finished the final “product” pretty much alone and about a week before all the other teams. In the end, our game client probably had the greatest number of features, the most complex graphics, the nicest explosions and the most reliable motion engine—and we got the highest marks possible on the final evaluation by the professors. Somehow, I think that if I had not been able to run everything on my home Linux machine, and do everything when I wanted it and how I wanted it, I probably would not have reached this level of achievement.

Other than showing that some Computer Engineering students are definitely more productive on their home machines than on most computers you can give them access to, this somewhat extraordinary adventure shows that some fields—which, until now, were reserved for high-end workstations—can be

explored with something as simple as a good Linux box and some relevant software.

Future Directions

UPDATE: 11/29/2006

If you want to see the pictures and code relating ot this article go to this link: www.step.polymtl.ca/~coyote/graphics_tank.html

Vincent Cojot is a student in Computer Engineering at the Polytechnical School Of Montréal. He enjoys Computer Graphics, Xview/OL programming (under Linux, of course) and miniatures painting.

Archive Index Issue Table of Contents

Advanced search

The Java Developer's Kit

Arman Danesh

Issue #31, November 1996

Are you an absolute beginner? Here's a brief introduction to using the JDK.

Java has taken the Internet and programming communities by storm during the past year with its promise to enable the creation of software that can run on any platform from a single binary file and be used securely in a distributed network environment.

The Java concept is simple: a single source code file is compiled to a single pseudo-binary file containing Java byte codes. This binary file can be run on any platform for which a Java interpreter or other runtime engine exists.

In order to develop and test these Java applications, it is necessary to have access to several critical development tools, including a compiler, a debugger and a Java interpreter for testing applications. Numerous Java development environments already exist—primarily for MS-Windows systems. These full professional development tools are produced by the likes of Borland and Symantec. For Linux, and Unix in general, this type of commercial development tool hasn't become available.

Nonetheless, this doesn't mean that Linux users are unable to develop Java applets and applications. Sun Microsystems has developed a free Java Developer's Kit (JDK) which includes a compiler, a debugger, a runtime environment and an applet viewer for testing Java applets embedded in web pages.

Obtaining the JDK

The Java Developer's Kit was originally developed by Sun for SPARC, Solaris and Windows NT/95. These versions of the kit, along with more recent versions for x86 Solaris and MacOS, are available from the JavaSoft web site at http://www.javasoft.com/

Sun has also allowed other organizations to port the JDK to other platforms. The Blackdown Organization (Randy Chapman) has ported version 1.0.1 of the JDK to Linux and makes binaries available for x86 versions of Linux. The JDK distribution for Linux is available from ftp://ftp.blackdown.org/pub/Java/linux/. Blackdown also has a web page at www.blackdown.org/. On the web site you will find three files:

```
linux.jdk-1.0.1-try3.common.tar.gz
linux.jdk-1.0.1-try3.static-motif-bin.tar.gz
linux.jdk-1.0.1-try3.shared-motif-bin.tar.gz
```

It is necessary to download two of these three files:

```
linux.jdk-1.0.1-try3.common.tar.gz
```

plus either:

```
linux.jdk-1.0.1-try3.static-motif-bin.tar.gz
```

or:

```
linux.jdk-1.0.1-try3.shared-motif-bin.tar.gz
```

The last file requires an ELF binary of version 2.0 of the Motif libraries (libXm.so.2). If you don't have Motif, download linux.jdk-1.0.1-try3.static-motif-bin.tar.gz instead—it is a larger file but will work regardless of whether you have the Motif libraries or not. Throughout this article we will be using linux.jdk-1.0.1-try3.static-motif-bin.tar.gz.

All versions of the Java Developer's Kit also require the following libraries, many of which may already be on your system:

- /lib/libc.so.5.2.16
- /usr/X11/lib/libX11.so.6.0
- /usr/X11/lib/libXt.so.6.0
- /usr/X11/lib/libXext.so.6.0
- /usr/X11/lib/libXpm.so.4.3
- /lib/libdl.so.1.7.9

If you are missing any of these libraries, the JDK will not work. These libraries are all freely available on the Internet.

Installing the JDK

Once the “common” tar file and one of the Motif tars have been downloaded, they need to be uncompressed. The documentation with the Linux JDK recommends installing the JDK in the /usr/local directory (although it can be

installed elsewhere). To do this, copy the two compressed tar files to /usr/local with:

```
cp linux.jdk-1.0.1-try3.common.tar.gz /usr/local
cp
linux.jdk-1.0.1-try3.static-motif-bin.taR.GZ
/usr/local
```

Then, the files can be uncompressed and untarred with:

```
tar xzvf linux.jdk-1.0.1-try3.common.tar.gz
linux.jdk-1.0.1-try3.static-motif-bin.tar.gz
```

This will create a directory java/ under /usr/local which contains four subdirectories: bin, demo, include and lib. /usr/local/java will also contain a zip file with the source and various README and HOWTO files. The bin/ directory contains the scripts used to execute all the components of the JDK. The components are:

- **appletviewer**: A viewer to test applets embedded in HTML documents.
- **javac**: The Java compiler: compiles Java source code to Java byte code binary files (known as class files). The class files produced by javac can be run by a Java interpreter on any platform.
- **java**: The Java interpreter: used to execute Java class files under Linux.
- **jdb**: The Java Debugger: a command-line debugger which is in alpha development.

Each of these scripts in java/bin actually call executable files in java/bin/i586. These scripts expect certain tools to exist in fixed locations in your system. Specifically, the appletviewer script expects mkdir to be in /usr/bin and pwd to be in /bin. On some systems, this may not be true (for instance, in RedHat-derived systems, you may find mkdir in /bin). There are two solutions to this problem. One is to edit java/bin/appletviewer and replace the incorrect occurrences of /usr/bin/mkdir or /bin/pwd with the correct full paths of these programs.

The second solution is to create symbolic links in the directories expected by appletviewer. For instance, on RedHat systems where mkdir is in /bin, a symbolic link could be created in /usr/bin with the command:

```
ln -s /bin/mkdir /usr/bin/mkdir
```

If you expect to be using the JDK components frequently, you will probably want to add the java/bin directory to your path. Assuming you installed the JDK under /usr/local and you are running the bash shell (the default shell for many Linux distributions), you could add the following lines to the .bashrc file in your home directory:

```
PATH=/usr/local/java/bin:$PATH
export PATH
```

If you are running C shell, the following line at the end of your .cshrc file in your home directory will do the job:

```
setenv PATH /usr/local/java/bin:$PATH
```

Using the JDK

Although the details of programming and developing Java applications and applets are beyond the scope of this article, we will briefly cover how to go about compiling and running Java applications and applets. Java source code is saved in files with the .java extension. Once compiled, a class file (with a .class extension) will be created. Assuming the java/bin directory is in your path as outlined above, a Java source file can be compiled with:

```
javac filename.java
```

Class files for applications can be executed using:

```
java filename.class
```

Applets are a little more complicated. Applets are run as embedded pieces of web pages and are included in web pages with a special <APPLET> tag. You can test an embedded applet with appletviewer or Netscape Navigator 2.0 or 3.0. With Navigator, simply choose Open from the File menu and then open the HTML file with the embedded applet.

With appletviewer, simply type:

```
appletviewer filename.html
```

Running applets as well as some Java applications requires that you are working in an X-Windows environment. If you don't have X-Windows installed on your system you won't be able to test applets or any applications which make use of Java's GUI development capabilities.

Troubleshooting Your Installation

On most Linux systems the steps outlined above should be all that is required to get the JDK up and running. However, on some systems you may experience

some difficulty; some of the common errors and their solutions are outlined below:

- You get an error message referring to /dev/zero. The device /dev/zero needs to have world read and write permissions. Set these permissions using:

```
chmod 666 /dev/zero
```

- You get “dirname: too many arguments” or “cannot find class” errors. The component you are trying to run cannot find the native Java class files. The JDK uses the environment variable CLASSPATH to find these files. This variable is set in java/bin/.java_wrapper and java/bin/appletviewer. However, with your shell, these scripts are having trouble determining the correct directory. You can edit these files so that the CLASSPATH gets set correctly.

In .java_wrapper, change the line which reads:

```
J_HOME=`dirname $PRG`/..
```

to

```
J_HOME=/usr/local/java
```

(or wherever you installed the JDK) A similar change needs to be made in appletviewer.

Getting More Information

Once you have installed the Java Developer's Kit, you will probably want more information about developing Java applications and applets. Aside from Sun's official Java home page at http://www.javasoft.com/, the Gamelan directory (http://www.gamelan.com/) provides an extensive collection of applets and applications—many with source code—as well as pointers to other reference material. The comp.lang.java newsgroup is a high-volume newsgroup which is actively used by many Java experts and novice programmers.

Arman Danesh (armand@juxta.com) is a technology journalist who contributes regularly to several publications around the world. He writes a weekly Internet column in *The South China Morning Post* called “The Other World” and a regular column called “Trawling the Net” in *The Dataphile*. He is the author of *Teach Yourself JavaScript in a Week* from Sams.net Publishing.

Archive Index Issue Table of Contents

Advanced search

LJ Interviews Larry Gritz

Amy Wood

Issue #31, November 1996

A Techmical Director of Toy Story gives us the scoop from Pixar Studios.

Amy Wood, the graphics/layout artist for *Linux Journal* interviewed Larry Gritz of Pixar Animation Studios on August 16.

AmyI understand that you were a Technical Director for the latest great animation feature film, **Toy Story**. Can you tell us what you did in that role?

LarryI was one of 30 or so technical directors (TDs) who worked on that film. TD is the job title for people who create the models, write shaders and light the shots. Essentially, they are technical folks responsible for the visual look of the film. Another group, the animators—typically with classical animation rather than technical backgrounds—is responsible for the motion or acting of the characters. There are also countless other people writing the story, designing the look, painting, developing software and so on. In all, it's quite a big team of incredibly talented people. I came into the project fairly late in the production, after modeling was mostly done, but I got to work on shaders and lighting.

What is your background? How did you get involved with graphics?

I started out interested mainly in compilers, but I took a course in computer graphics when I was an undergraduate at Cornell, and I've been hooked ever since. I started tinkering around with writing renderers, and concentrated on graphics in graduate school at George Washington University, doing my MS thesis about a new way of calculating a particular kind of light propagation. I stayed for a PhD (which I am still in the process of wrapping up), doing more research in animation techniques, among other things.

Can you tell us about your Blue Moon Rendering Tools software?

BMRT is a high quality renderer which supports ray tracing and radiosity, area lights, volumetric effects and other advanced features. It runs on several Unix platforms, including SGI, Sun, HP, NEXTSTEP, and of course, Linux. It's cheap shareware, and is free for academic and non-commercial usage. Features that set it apart from most other renderers include support of curved high level surfaces such as bicubic patches and trimmed NURBS, good anti-aliasing support and programmable shading (users can write little programs called “shaders” which control the appearances of surfaces and lights). These features aren't found in many renderers (including big commercial packages), but they are essential to high-end, professional quality rendering.

BMRT is fully compliant with the RenderMan Interface Specification, developed by Pixar. RenderMan is a standard way for modelers to talk to renderers, sort of like PostScript, but for describing 3-D photo-realistic scenes. By being RenderMan compliant, BMRT is largely compatible with Pixar's PhotoRealistic RenderMan product (PRMan, for short), which is probably the most commonly used renderer for feature film effects work (and of course, was used to render *Toy Story*).

BMRT is not particularly easy to use; it's really more of a developer's product. But it's extremely powerful—much more so than any of the other free renderers out on the net, which are more oriented toward hobbyists.

The BMRT home page, www.seas.gwu.edu/student/gritz/bmrt.html, has several pictures that have been rendered using my software.

Why did you pick Linux as a platform for Blue Moon?

A couple of years ago, I was introduced to Linux by Youngser Park, a fellow graduate student at GWU. He asked me to port BMRT to Linux so that he and other students could run the renderer (as well as our other lab tools) at home. I remember the first time I was at his place and saw Linux running on his computer. I'd heard of Linux before, but never imagined that it could be a robust implementation. When I saw X11 running and realized how easy it was to set up an environment just like I was used to on the SGI, I knew I needed to be running it on my home machine, as well.

What are some interesting projects where the BMRT have been/are used? Is BMRT a popular package? Do you know of any studios that use it running under Linux?

BMRT is fairly popular in the high end. It's rather hard for beginners to use, so it doesn't come close to say, POV-Ray, in terms of the number of people who use it. But because it's so powerful, and RenderMan compliant, it's gotten noticed

by a lot of production houses. Judging by the mail I receive, several well-known studios have at least tried it out. Pixar's renderer is much faster, and is less prone to some very subtle artifacts, so no studios would want to use my software instead of PRMan. But since the algorithms are very different, many houses use them together—PRMan for the bulk of the work, and BMRT for those pesky scenes when they just have to have ray tracing or area lights or something. I can't name the studios, but I know BMRT has been used for a couple TV commercials and for an episode of *Star Trek: Voyager*. I don't think it's been used for final frames of any feature films yet, but I wouldn't be surprised if that happened soon.

I don't know any studios currently using Linux “officially”, but many people who work at production houses run Linux at home and like to be able to continue to tinker with shaders and such.

How do you think Linux compares to other platforms?

I think it's a more robust Unix-like OS than many I've seen from the big commercial workstation vendors. I also like the spirit of community and the kind of high-quality, low cost software that is available for Linux. I've tried to contribute to that with the availability of my software for Linux.

Do you think your decision to offer a commercial graphical/rendering tool for Linux will inspire others to make more packages available?

I hope so. With high end Intel chips being a very cost-effective way to get lots of computational power, I wouldn't be surprised to see studios using large farms of Intel-based hardware for their rendering or other graphics tasks. If this is the case, I'd much rather see these machines running Linux than NT.

And in the vein of Barbara Walters, if you had to be one character in **Toy Story**, who would you be?

Probably Sid, though perhaps without the sadistic streak. I like the tinkerer in him. He has sort of a God complex, but he sure does make interesting toys.

Note that RenderMan is a registered trademark of Pixar, and **Toy Story** is registered and copyrighted by Walt Disney Corporation. **The RenderMan Companion** by Steve Upstill (Addison-Wesley, 1990) is a good reference for more information about the RenderMan standard.

Larry Gritz (lg@pixar.com) is a Technical Director for Pixar Animation Studios in Richmond, California. He holds an MS in Computer Science from George

Washington University. Visit him on the web at http://www.seas.gwu.edu/student/gritz/, and check out his Blue Moon software.

Archive Index Issue Table of Contents

Advanced search

Linux-GGI Project

Andreas Beck

Steffen Seeger

Issue #31, November 1996

The Linux-CGI Project goals are explained—what it intends to accomplish and how it will do it.

Introduction

In this article, we will explain the intentions and goals of the Linux-GGI Project along with the basic concepts used by the GGI programmers to allow fast, easy to use access to graphical services, hide hardware level issues from applications and introduce extensible support for multiple displays under Linux. The Linux-GGI project wants to set up a General Graphical Interface for Linux that will allow easy use of graphical hardware and input facilities under the Linux OS. Already existing solutions and standards like X or OpenGL do deal with graphic's issues, but these current implementations under Linux have several (sometimes serious) drawbacks:

- Console switching is not deadlock-free, because the kernel asks a user-mode application to permit the switch causing a problem in terms of security. Since **any** user-mode application can lock the console, the kernel has to rely on the application to allow a user-invoked switch. For stand-alone machines, if the console locks in an application without a switch, a system reboot will have to be done.
- The Secure Attention Key (SAK), which kills all processes associated to the current virtual console might help with the above problem, but for graphics applications the machine might still remain locked, because the kernel has no way to do a proper reset of the console—after all, it has no idea which video hardware is present.
- Any application accessing graphical hardware at a low level has to be trusted as it **needs** to be run by root to gain access to the graphical hardware. The kernel relies on the application to restore the hardware

state when a console switch is initiated. Relying on the application might be okay for an X server that needs superuser rights for other reasons, but most of us would not want to trust a game that is available to us only in binary form.

- Input hardware (such as a mouse or a joystick) can be accessed using the current approach, but it can't easily be shared between several virtual consoles and the applications using it.
- No clean way is available to use more than one keyboard and monitor combination. You might think that this is not possible on PC hardware anyway; but in fact, with currently existing hardware there are ways to have multi-headed PCs, and the USB peripheral bus to be introduced soon may allow for multiple keyboards, etc. Besides, other architectures do support multiple displays, and if Linux did also, it would be a good reason to use Linux for applications like CAD/CAE technology.
- Games cannot use the existing hardware at maximum performance, because they either have to use X, which introduces a big overhead (from a game programmer's point of view), and/or access the hardware directly, which requires separate drivers for every type of hardware they run on.

GGI addresses all these points and several more in a clean and extensible way. (GGI does not wish to be a substitute for these existing standards nor does it want to implement its graphical services completely inside the kernel.) Now, let's have a look at the concepts of GGI—some of which have already been implemented and have shown their usability.

Video Hardware Driver

The GGI hardware driver consists of a kernel space module called Kernel Graphical Interface (KGI) and a user space library called libGGI. The KGI part of GGI will consist of a display manager that takes care of accessing multiple video cards and does MMU-supported page flipping on older hardware. This method allows for incredibly fast access to the frame buffer from user space whenever possible. (This technique has already been proven—the GO32 graphics library for DJPGG, the GNU-C-compiler for DOS, uses this method and has astonishingly fast graphical support on older hardware.) If this memory-mapped access method can be used in GGI, there will be no loss in performance as the application reads or writes the pixel buffer directly.

Each type of video card in the system has its own driver, a simple loadable module that registers as many displays as the card can address. (Video cards exist that support two monitors or a monitor and a TV screen.) The driver module gives the system the information needed to access the frame buffer and to access special accelerated features, the setup of a certain video mode and the limits of the hardware (e.g., the graphic card, the monitor, and any

other part of the display system). The module can either be obtained from a single source file or be linked using precompiled subdrivers for each graphical hardware subsystem (ramdac, monitor, clock chip, chipset, accelerator engine). This last option is the favourite approach, since it allows support for new cards to be added quite easily, as only the subdrivers for hardware not already supported need to be implemented and tested. (The others are already in use or bug fixes there will improve all drivers using them.) This scheme has been used to derive support for many of the S3 accelerator-based cards, and has proved to be very efficient and easy to use. It also allows for efficient simultaneous development for several graphic cards. The subdrivers to be linked together are now selected at configuration time, but they can also be selected after automatic detection or according to a database (not yet built). Note that the subdrivers do not need to be in source form; as a result, precompiled subdriver object files can be linked together during installation.

As each subdriver knows the hardware exactly, it can prevent the display hardware from being damaged due to a bad configuration and make suggestions about the optimal use of the hardware. For example, the current implementation has drivers for fixed- and multisync monitors that allow optimal timings for any resolution to be calculated on the fly without any further configuration. Of course, completely user- configurable drivers are also possible. In short, in addition to the hardware level code, the subsystem drivers provide as much information about the hardware as possible. This way the kernel will have sufficient methods to initialize the card, to reset consoles and video modes when an application gets terminated, and to make optimal use of the hardware. The KGI manager will allow a single kernel image to support GGI on all hardware, as any hardware-specific code is in the loadable module and only common services (such as memory mapping code) are provided from the kernel. The KGI manager will also provide data structures and support to almost any imaginable kind of input devices.

The user space library, called libGGI, will implement an abstract programming interface to applications. It interfaces to the kernel part using special device files and standard file operations. Applications should use this interface (or APIs provided by applications based on it) to gain maximum performance; however, other APIs can be built accessing the special files directly. Understand that in this case the X server will just be a normal application in terms of graphic access. Since X is considered to be the main customer for graphical services, the API will be designed according to the X protocol definition and will implement a set of low level drawing routines required by X servers. The library will use accelerated functions whenever possible and emulate features not efficiently supported by the hardware found. An important feature of future generation graphical hardware is 3D acceleration which easily fits into the GGI point of view. We plan to provide support for 3D features based on MESA,

which is close to OpenGL and ensures compatibility with other platforms than Linux.

Another issue when dealing with graphics is game programming as games need the highest possible performance. They also need special support by the video hardware to produce flicker-free animation or realistic images. The current approaches can't support this need in a reasonable way, since they cannot get help from the kernel (e.g., to use retrace interrupts). GGI can provide this support easily and give maximum hardware support to all applications.

Input Hardware Driver

There are many ways to interact with computers—a keyboard, a mouse, even a cybersuit. All of these methods have special protocols to report user actions and even need special hardware to be accessed. GGI will allow any kind of input to be supported without recompiling the kernel for each new device, thus allowing for flexibility and easy configuration. This support is achieved by having a loadable module for each device or device class. Just like the video card drivers, any input device driver will register abstract input devices that convert user actions to events.

For example, an application might query for the registered devices and select the events it wants to receive, so that a game program could default to use joysticks or keyboard input depending on the environment. Installing a game or an X server will not require any further configuration other than copying the binary to its destination directory and starting it. Please note that this methodology will also considerably reduce the effort required to maintain several differently-equipped machines as the application binaries will be the same for all machines and can be shared via network file systems. Only the GGI modules to be loaded will differ from machine to machine.

A New Way of Understanding Consoles

GGI defines a console as a pair—a display and a (mandatory) character input device. Optionally, other input facilities like a pointing device or controllers attached to a console may be present. The display is capable of providing alphanumeric data or graphics while the character device provides character input (just as the name implies). We use these diffuse terms as the display actually might be something other than a monitor, e.g., braille lines or other devices that help disabled or handicapped people to work with computers. Similarly, the input might be a keyboard or a voice recognition program or hardware—just about anything you can imagine. However, the character input device is mandatory, because it focuses on one and only one virtual console that is shown on one of the displays registered by the loaded modules. Any other devices are associated with one of the keyboards, and any user activity is

reported to applications running on the focused console. Thus, it is not only possible to have multiple virtual consoles but also (in conjunction with multiple displays) to have several real consoles.

If the user wants to switch between two virtual consoles, the keyboard driver will tell the KGI manager to bring the specified virtual console on the display assigned to it and then report any keyboard, pointer and controller events to the application. One problem arising from the virtualization is that an application accessing accelerated features might first have to terminate the current command or that the frame buffer needs to be preserved even if the application goes into background mode. GGI will effectively hide this operation from the application. Applications can be placed into one of the following categories with examples given:

- The application can redraw its screen without noticeable overhead at any given time, e.g., X server.
- The application can be programmed considerably more easily when a back-up buffer is provided in case the frame buffer needs to be accessible at any time, e.g., a ray tracer or any other program that needs to do a lot of calculations to draw an image. This back-up would also allow running the application in background mode while continuing to draw to its frame buffer.
- The application can skip output or simply sleep, if not in foreground mode, thereby reducing system load significantly, e.g., games or software video decoders. SVGALIB works in this manner.

Class one and three are easy to virtualize—they just have to redraw their buffers when switched to foreground mode, and therefore, when switching to background mode, the screen contents are discarded and drawing requests are ignored. The only difficult class is class two. However, since the kernel knows the exact state of the hardware, it can tell a user space daemon to allocate sufficient memory, save the frame buffer there, redirect the memory mapping of the application and tell the library to use optimized drawing methods for memory- mapped buffers instead of accelerated drawing functions.

GGI plans to add powerful graphical hardware support to the Linux operating system. As with any hardware driver, it needs to have a kernel segment that is kept to a minimum (currently the modules are about 30K in size, and should not become greater than 80K). If accepted by the Linux community, GGI can provide a clean method of dealing with multiple display and input hardware as well as an architecture-independent programming interface that will give good performance on any platform. Also, it will allow hardware manufacturers to provide optimized drivers for their hardware if they wish. During development much care has been and will continue to be taken to isolate machine or

hardware-dependent code, whenever possible, in order to provide good portability.

Sidebar: Linux-GGI Project Resources

As GGI is still under development, several features are not yet implemented, but there is a first implementation that demonstrates that our concepts are capable of providing easy access to video hardware and solving all of the points addressed in this article. Currently being worked on is support for multiple displays and libGGI. Of course, introducing a new concept to the kernel to access video hardware will cause several (non-X) applications to be incompatible, but on the other hand, adding this concept will ease the configuration of Linux, and open up new vistas to game programmers with an operating system and graphical support that will allow maximum performance on any system.

Andreas Beck (becka@sunserver1.rz.uni-duesseldorf.de) studies physics at the University of Duesseldorf, Germany and started the GGI project. He developed the memory mapping code for GGI, worked on the library implementation and made major contributions to the concepts used.

Steffen Seeger (seeger@physik.tu-chemnitz.de) also studies physics at the University of Technology at Chemnitz-Zwickau. He wrote most of the S3 driver code and made major contributions to the console concepts and the kernel drivers.

Archive Index Issue Table of Contents

Advanced search

Java and Postgres95

Bill Binko

Issue #31, November 1996

First in a series of articles detailing the creation of a Java interface to Postgres95.

Java's native methods are functions written in C (or another compiled language) and dynamically loaded by the Java interpreter at run time. They provide the means to access libraries that have not been ported to Java, and also allow fast compiled code to be inserted at critical points in your system.

In this article, we will walk through the complete process of writing native code. We will create a Java interface to Postgres95 by writing wrapper classes around the libpq library. Postgres95 is a free database system (licensed under the GPL) that runs on most varieties of Unix, including Linux

While written (and tested) solely on Linux, the principles of this article should apply to any version of Unix and (with the exception of how to build the shared library) the code should be easily ported. To get the most out of this article, you should have some Java experience, or be very familiar with C++ and OO principles.

An Introduction to Java

Recently, Java has received a great deal of attention (and quite a bit of hype) as a fantastic WWW tool. “Java Powered” pages with animations and interactive interfaces have popped up all over the Web, and everyone, including Microsoft (gasp!), is clamoring to add Java capabilities to their browsers. What many people don't realize is that Java is much more than that: it is a complete programming language suitable for use in standalone and, in particular, client-server applications.

Java offers several features that make it ideal for an application language. First among these is obviously portability. With Java there is no need to write

Windows95, Mac, and several Unix versions of your application. Since the code is run by the Java Virtual Machine (VM), all that is necessary is that the VM (and any native libraries you want to use) be ported to that platform.

Another compelling reason to write in Java is the depth of its libraries ("packages" in Java-speak): networking, I/O, containers, and a complete windowing system are all integrated. Many of these capabilities are "crippled" when running a Java applet, but applications are free to make complete use of all of them. Java is a multi-threaded environment, allowing safe use of threads on platforms that don't currently support them natively. Java has a garbage collection system that eliminates the need for explicit freeing of memory. Exception handling is built in (and its use is actually required by many of the libraries, including the one we will write), and its true OO nature eases inheritance and re-use.

Sidebar: Class Repositories: the Motivation behind Jgres

Interfacing Java with Existing Systems

Even with all these things going for it, using Java for an application still has one major drawback: many systems don't yet have a Java interface, and writing them from scratch is often difficult, or even impossible.

This is the problem I faced when I wanted to access a Postgres95 database from Java. There was an excellent (and simple) C library (libpq) that shipped with Postgres95, but no support whatsoever for Java. Since the source (in this case) was available, I considered recreating libpq under Java, but this proved to be a substantial chore, and required intimate knowledge of Postgres internals. (In fact, as of this writing, John Kelly of the Blackdown Organization is writing just such a beast. It's called the Java-Postgres95 project, and you can find an alpha version at ftp://java.blackdown.org/pub/Java.

Then I decided to simply write wrapper classes for libpq. There are several drawbacks to this approach: First, it cannot be used in an applet. Browsers explicitly disallow any access to native code (except those provided with the browser), so these classes simply will not work. Second (and more importantly), this solution is not as portable as one written in straight Java. While libpq is portable to all major flavors of Unix, and the code we'll write will be as well, there is currently no libpq for Windows95/NT or the Mac.

Apart from being simpler, there is one other advantage to writing this in native code: When the Postgres95 project releases bug fixes or changes their communication protocol, little or no change will be required to our code.

Sidebar: How to get Postgres and Java

The Battle Plan

We will proceed in three steps, providing examples of how to use each part along the way.

First, we'll create wrappers for libpq's PGconn, and PGResult. This will allow us to connect to the database, issue queries, and process the results.

Then, we'll write a new interface to Postgres95's Large Objects (or blobs in other databases), using Java's Stream classes.

Finally, we'll use Java's threads to provide an easy, behind the scenes interface to Postgres95's asynchronous notification system.

Using Native Methods in Java

Java methods (class functions) that have been declared “native” allow programmers to access code in a shared library. Theoretically, this code can be written in any language that will “link with C” (but in general, you'll probably want to stick to C, or perhaps C++).

When a Java class is loaded, it can explicitly tell the Java system to load any shared library (.sos in Linux) into the system. Java uses the environment variable LD_LIBRARY_PATH (and ldconfig) to search for the library, and will then use that library to resolve any methods that have been declared “native”.

The general procedure for writing native code is as follows:

- Write the .java file, declaring all native methods as “native” (The .java file must compile cleanly at this point, so insert dummy methods if you need to)
- Add the **loadLibrary()** command to your .java files to tell Java to load the shared library
- Compile the class:

```
javac [-g] classname.java
```

- Generate the headers and stubs:

```
javah classname (no extension)
```

```
javah -stubs classname
```

- Use the declarations in the classname.h file to write your C code (I use the file classnameNative.c, as it seems popular, and the stubs file uses classname.c)
- Compile the .c files using the **-fPIC** (position independent) flag:

```
gcc -c -fPIC -I/usr/local/java/include
filename.c
```

- Generate the shared lib (these flags are for gcc 2.7.0):

```
gcc -shared -Wl,-soname,libFOO.so.1 -o
libFOO.so.1.0 *.o -lotherlib
```

- Put the .so file somewhere in your LD_LIBRARY_PATH (or add it to /etc/ld.so.conf).

An Example: The PGConnection Class

The PGConnection class is a wrapper for libpq's PGconn. A PGconn represents a connection to the backend Postgres95 process, and all operations on the database go through that connection. Each PGConnection will create a PGconn and keep a pointer to it for future use.

Let's walk through the steps above:

First, we write our PGConnection.java file (Listing 1). Remember that it must compile cleanly in order to generate our header and stubs, so if you refer to any Java methods that you haven't written, create dummy methods for them. We will need a constructor, a finalizer, and all of the operations that libpq allows on a PGconn. We declare most of these operations as native methods (see Listing 1—**exec()** and **getline()** are special cases that we'll consider later).

Listing 1. PGConnection.java

The PGConnection Constructor

To get a PGconn, libpq provides the function:

```
PGConn *setDB(char *host, char *port, char *options, char *tty,
char *dbName)
```

Since this in effect “constructs” the connection to the database, we'll use this as a model for our constructor (See Listing 1, line 18). The constructor simply calls **connectDB()** (Listing 1, line 21; a native method that calls **setdb()**—we'll define it in a moment), and throws an exception if the connection is not made. Doing the error checking in the constructor guarantees that no connection will be returned if the call to **setdb ()** fails.

Now let's look at our first native method, **connectDB()**. We declare it as native at line 70 in Listing 1. Note that no Java code is provided.

There are several important things to notice about this declaration. The “private” keyword makes this method accessible only from the PGConnection class itself (we want only our constructor calling it). The “native” keyword tells Java that code from a shared library should be loaded for this method at runtime. Since libpq is not “thread-save”, we want to make it impossible for two

threads to be making calls to libpq at the same time. Making all of our native methods “synchronized” goes a long way towards this goal (we will return to this when we tackle the asynchronous notification system). Finally (Listing 1, lines 70-73), the declaration states that **connectDB()** takes five Java strings as arguments and doesn't return anything.

Figure 1. How Types Convert to and from Java and C

The remainder of the native calls follow this same pattern, with the exception of **exec()** and **getline()**. Again, we'll put these off a little longer.

Before we continue, let's add the loadLibrary call. We place it at the end of the class, in a block marked “static” (Listing 1, line 92) with no method name. Any blocks such as this are executed when the class is loaded (exactly once) and libraries that have already been loaded will not be duplicated. In our example, we'll name the library libJgres.so.1.0, so we need to use loadLibrary (“Jgres”) (See Listing 1, line 94).

With our .java file complete, we are ready to write the C code. First, we compile the .java file with:

```
javac PGConnection.java
```

Then, we create the “stubs” file and the .h file with:

```
javah PGConnection
javah -stubs PGConnection
```

At this point you should have PGConnection.h and PGConnection.c in your current directory. PGConnection.c is the “stubs” file, and should not be modified. For our purposes, the only thing you must do to the stubs file is to compile it and link it into your shared library.

PGConnection.h is a header file that must be included in any C file that accesses PGConnection objects. At line 14 (see Listing 2) you will find the declaration of a struct corresponding to the data for our object. Below that you will find prototypes for all of the native methods we declared. When writing the C code for native methods, you must match these signatures exactly. Listing 2. PGConnectionNative.c (includes PGConnection.h)

Now, let's (finally) write the C code.

The code for **connectDB** is very straightforward, and demonstrates the majority of the issues involved in writing native code. Notice that the first argument to **connectDB** is not listed in the PGConnection.java file. Java automatically passes a “handle” (a fancy pointer) to the object you are dealing with as the first

parameter of every native method. In our case, this is a pointer to a struct HPGConnection (defined in PGConnection.h), which we name “this” (Listing 2, line 14. If you're working in C++, you may want to use “self” since “this” is a keyword). Any access to the object's data must go through this handle.

The remainder of the parameters are the Strings we passed in (see PGConnection.java). These are also passed as handles, or pointers to the struct Hjava_lang_String (defined in java_lang_string.h, included by native.h). We could access these structures like any other handles (see below), but Java provides several convenient functions that make it much easier to work with strings.

The most useful of these functions are **makeCString** and **makeJavaString**. These convert Java's Strings to char *s and vice versa, which use Java's garbage collector to handle memory allocation and recovery automatically. (

Beware of a major pitfall here!

You must store the value returned by **makeCStrin**g in a variable. If you pass the return value directly to a function, the garbage collector may free it at any time. The same is not true of **makeJavaString**.) Lines 30-34 in Listing 2 show the use of **makeCString** and we use **makeJavaString** first at line 51. Lines 41-42 in Listing 2 show our call into the libpq library. It is called exactly as normal, and the resulting pointer is stored in the variable tmpConn. You may notice that we don't do any error-checking here: we do that in the Java code for our constructor, where it is easier to throw exceptions.

As I mentioned above, **PGConnection** needs to keep the **PGconn** pointer around, so that it can use it in later calls—all later calls, in fact. In order to do this, we will store the 32 bit pointer in a data member with Java type **int** after casting it to a C long to avoid warnings (see Table 1 for a list of type conversions).

To access this member, we must use Java's “handles”. Handles are used to access data in a Java object. When you want to access a data member, you simply use **unhand(ptr)->member** rather than **ptr->member** (where ptr is the handle). We do this on line 42 of PGConnectionNative.c (Listing 2) to save the pointer returned by setDB in a Java int (note: if you forget the unhand() macro, you will get a warning about incompatible pointer types).

This function has covered almost all you need to know to call C functions from Java (calling Java methods from C is possible, but the interface is clumsy at best at this point, and where possible, I'd avoid it). Most of the rest of the methods (host, options, port, etc.) simply convert the data and make the C call. We'll just take a look at one of these, **PGConnection.db()**.

The only significant portion of the C function **PGConnection_db()** is its first line (Listing 2, line 46). It needs a **PGconn** to pass to **PQdb()**, so it must get it out of the **PGConnection** member, **PGconnRep**. It uses **cw[unhand()** to get the pointer as a long, then casts that to a **(PGconn *)**. Since this line is so messy (and is starting to look like lisp!) I created a macro, **thisPGconn**, to clean up the code a little. It is used in the remainder of the file, and its definition is at the top of the file (don't put it in PGConnection.h, since that is machine-generated).

All of the native methods in the Java class PGResult follow the same basic structure, and there is no reason to go over them.

Jumping through Some Hoops

There are some places where Java and C just don't get along. The rest of this section will touch on the few I found, and how I avoided them.

Hoop #1: Returning Java Objects (exec() Explained)

The **exec()** method (see, I told you I'd get to it) needs to return a PGResult object. This is in keeping with libpq's structure, and the OO nature of Java. However, returning an object from a native method can get pretty hairy. The “official” way to do it is to call the function:

```
HObject *execute_java_constructor(ExecEnv *,
                                  char *classname,
                                  ClassClass *cb,
                                  char *signature, ...);
```

and return the HObject * it returns. Personally, I find this interface extremely clumsy, and have managed to avoid it. However, for completeness, the actual call in our case would be:

```
return execute_java_constructor(EE(), "classPGResult",
                                0, "(I)LclassPGResult;",
(long)tmpResult);
```

I found it far easier to create a buffer between the call to **exec()** and the call to **PQexec()** that could call the constructor from Java. This is where the **nativeExec()** method comes from. **exec()** simply passes the string to **nativeExec()**, which returns an int (the PGresult pointer that **PQexec()** returned). Then it calls PGResult's constructor with that int.

The extra layer will also come in handy when we add the asynchronous notification system.

Hoop #2: Append to Strings in Java, not C (getline() Explained)

PQgetline() expects the user to continually call it while it fills in a static buffer. This is simply not needed in Java. A much nicer interface is to just have **getline()** return a String. However, building the String (appending each return value from **PQgetline()**) required calling Java methods from C—which, as we saw in Hoop #1, is very messy. By using a StringBuffer (a String that can grow) and doing the work in the Java code, it's much easier to understand, if a little slower.

The flip side of this is that the return value is now the String, so there must be another way to tell if an error has occurred or an EOF has been reached. One solution (I'm looking for a better one), and the one we use, is to set a data member flag. If the flag has been set to EOF, we simply return a Java null String. So once again, an extra layer saves us from a lot of truly gross code!

Hoop #3: You Can't Get a Stream's FILE*; (trace() and formatTuples() Explained)

This is one hoop I think the JavaSoft team should've solved for us. There is simply no way to get a FILE * (or a file descriptor) from a FileStream. **PQtrace()** expects a FILE *, so we simply open one, based on a filename passed in by the user. We check to see if it's “stdout” or “stderr”, and act accordingly.

We see the problem again when we try to implement Postgres95's **printTuples** (or **displayTuples** for 1.1). It also expects a FILE*, but this time the solution is a little messier. Here, we want the output in a String, so we open a temporary file, send it to the libpq function, rewind it, read it, and close it. This is pretty messy, but it does work, and is actually pretty quick about it. If we wanted to write a cleaner version, we could certainly rewrite **displayTuples()** completely in Java code, using PGResult's native methods **fname()** and **getValue()** that we have already defined.

The Finish Line

:

After writing all the C code, we are ready to generate our shared library.

First, we have to compile the .c files:

```
gcc -O2 -fPIC -I/usr/local/java/include/  \
              -I/usr/local/java/include/solaris \
              -c PGConnectionNative.c
gcc ...                                 (repeat for each .c file)
```

Then we link them:

```
gcc -shared -Wl,-soname,libJgres.so.1
-o libJgres.so.1.0 *.o -lpq
```

The -lpq tells the dynamic loader to load **libpq.so** when Java loads this library.

And finally, put them somewhere the dynamic loader can find them (in your LD_LIBRARY_PATH, or in a standard location (i.e. /usr/local/lib) and rerun /sbin/ldconfig -v).

That's all there is to it. Now we can use PGConnection and PGResult just like any other Java classes.

A Simple libJgres Example

To finish up this section, let's use our new classes to implement a simple SQL client. The client will connect to a database “foo” and accept query strings from standard input. **PGConnection.exec()** will process the queries, and print the results to the terminal using **formatTuples()**. The connection to the database is made on line 17 in Listing 3 (QueryTest.java).

We use the libpq convention of sending NULL (the empty Java string **""** translates into a NULL char *****) for any parameters we don't know. Notice that the call to PGConnection's constructor is surrounded by a “try” block. If an exception is thrown within this block, we have a problem with the connection and exit nicely (lines 54-58, Listing 3).

At line 24 of Listing 3, we test some of the simple functions to print out information about what we're connected to. We then read a query string and quit if it is “q” or “Q”.

We process the query on line 33 of Listing 3, by calling **exec()**. Note that we nest another “try” block here, because if we get a PostgresException on an **exec()**, we want to simply print the error and continue (we handle the exception on lines 43-46). If we reach line 34, we know that the PGResult is valid. We check to see if it returned any tuples, and use **formatTuples()** to print them if it did. If not, we simply print the current status and continue.

Conclusion

In this segment, we've shown how to create simple Java wrappers for C library functions. In the next installment, we'll show how to use Java's Streams to wrap Postgres95's Large Objects, and finally, we'll create a multi-threaded interface to its Asynchronous Notification system.

Charles “Bill” Binko graduated from the University of Florida with a BS in Computer Engineering in 1994. Currently a software engineer in Atlanta, GA, he has been a Linux enthusiast since 1993. His main computer interests are in

simulation, genetic algorithms and distributed programming, and he finds Java an excellent platform for all of these.

Archive Index Issue Table of Contents

Advanced search

Letters to the Editor

Various

Issue #31, November 1996

Readers sound off.

Several readers asked *Linux Journal* about the registered trademark symbol after Linux, in particular after noting the R[registered] symbol after Linux on IDG Books' *Linux Secrets*, written by Naba Barkakati. The book's cover says: “Linux is a registered trademark of William R. Della Croce, Jr.” Is there really a registered trademark on the word Linux?

Funny You Should Ask…

IDG Books Worldwide, Inc. told *Linux Journal* they did a trademark search as they always do when deciding what to put on a book cover, and although surprised to find a registered trademark on Linux, they printed the information resulting from their search. Their intent was in no way to reinforce the registered mark, but to comply with trademark requirements.

In July 1996, we at *LJ* tried to contact the person who had filed for the trademark, Mr. William R. Della Croce, Jr., via phone and left a message giving our e-mail address and telephone number. Mr. Croce responded by e-mail with a brief note, stating that “LINUX” was proprietary to him and that we would be hearing from his attorney.

We e-mailed Linus Torvalds about the matter. Linus reiterated his determination that Linux remain in common use or be trademarked by some trustworthy organization or individual.

We investigated the trademark, which was filed for August 15, 1994 and registered September 5, 1995, with a first use date of August 2, 1994. Since this date is long after others have used the term “Linux”, it seemed there were ample grounds for protesting this trademark and we began gearing up to do so.

In August 1996, *Linux Journal* and other Linux companies reported that they had received letters from Mr. Croce informing them that:

> LINUX ® is proprietary. Information about obtaining approval for use and/or making payment for past use may be obtained by writing to the following address:...

Yggdrasil Computing filed for a trademark on their book title Linux Bible *in March 1995. Their trademark was turned down because Linux was already a trade name registered to Mr. Croce. In March 1996, Yggdrasil Computing filed a letter disputing Croce's trademark and showing that Linux was a generic term and that Yggdrasil's use was prior to Croce's in any event. By the time you read this, we may know the results of this action. Other companies and individuals are getting involved in the trademark issue as well, and we will try to keep you informed.*

Check our web site at http://www.ssc.com/lj/ for the latest update on the Linux trademark.

—Belinda FrazierAssociate Publisher

X-cellent Resource

In your September issue's *Letters to the Editor* Ethan Wellman wrote that he had problems with X. So have I, and so it seems, have a lot of people. Your reply was appropriate, but would have been more helpful if you had suggested he contact the XFree people at http://www.XFree86.org/.

—John Palsedge jpalsed@uswest.com

More Coverage of Various Platforms

I am an experimental physicist and much of the work I do involves data analysis and simulations on computers. I have recently begun using Linux on my home PC and on a PC at work and I have really become a big fan of Linux. However, much of my “real” work is still done on commercial workstations (with commercial OSs) from DEC and SUN. It appears to me that Linux could definitely become a low cost alternative to these workstations.

There are two things I would really be interested in seeing in *Linux Journal*:

- Some kind of comparison of Linux on various platforms to commercial workstations, i.e. benchmarks, software and hardware availability, etc.
- Comparison of Linux on Intel Pentium, Pentium Pro and on the DEC alpha chips. Now that several commercial vendors are advertising systems that

> run Linux on alpha chips in your journal, I think it would be very useful to people interested in buying these to have an idea of the pros and cons of Linux/alpha vs. Linux/Intel.

I have been very impressed with the (VMS) alpha machines in our lab and I am seriously considering the purchase of an Linux/alpha system.

—Frank Moore moore@pyvsfm.physics.ncsu.edu

Coming Up in *LJ...*

We agree with your observations, and The May 1997 issue of *Linux Journal* will focus on the various platforms available for Linux.

Korn Shell Bin for Free

The July 96 issue of *LJ* presents the new Korn shell (ksh93). What is not mentioned (and is not widely known) is that users who are not interested in commercial support can get Linux, Sun and other binaries for free (src is not available). This includes not only the ksh binary but also shared libraries and the Tksh extension for Tcl/Tk. Just check the URL http://www.research.att.com/orgs/ssr/book/reuse.

—Alexandre avs@daimi.aau.dk

Archive Index Issue Table of Contents

Advanced search

Is This Any Way to Run a Railroad?

Phil Hughes

Issue #31, November 1996

We take the tag line on the cover (“The Monthly Magazine of the Linux Community”) seriously.

While I have never run a railroad, I am guessing that publishing a magazine is like running a railroad. You have an assortment of diverse customer needs, you have a limited budget, and you need to offer the best possible service to everyone.

But the parallel goes beyond that. In the railroad business you can make up your train out of an assortment of different cars, and based on the number of cars and the terrain you must travel through, you can pick the number of engines to pull it. As publisher of *LJ*, I get to pick the article mix, pick the number of pages, and pick who receives the magazine. If I do my job right, *LJ* gets more customers, which gets it more revenue—revenue from advertisers as well as readers—and everyone benefits.

All that said, I want to tell you what has changed at *LJ*, why it has changed and what you will see in the future. And, for our old customers, I want to assure you that you will continue to get the service you expect.

We take the tag line on the cover (“The Monthly Magazine of the Linux Community”) seriously. I fought for this before Issue 1 was published, and I continue to fight to make sure we stay on track. Today, however, that community is changing and we need to respond to those changes. While there is still a large Linux development community, there are other easily-identifiable “communities” needing a reliable source of Linux information. Here are a few:

- Applications developers
- ISPs
- Linux (and Unix) novices

- Web developers
- Embedded systems builders

Let's take one example, web developers, and see why it is important that we give them the information they need. Linux is an ideal platform for developing web content and Linux systems make ideal web servers. But web developers have choices. When someone says, “Why should I use Linux instead of NT for my web server?” we need to have a good answer. Being able to point at books like *CGI Programming in C & Perl*, by Thomas Boutell, a Linux user himself, helps. Being able to show the person that a monthly magazine called *Linux Journal* will answer ongoing questions, offer sources for essential hardware and software, and generally offer needed support is another important part of the answer.

What's Changing?

Linux Gazette

The first major change is that we are taking over the *Linux Gazette*. For those of you not familiar with it, *LG* is a newsletter. *LG* has offered an assortment of quick tips and articles that, while useful, have appealed to a smaller segment of the Linux community. We have always considered their work to be complementary to ours.

John Fisk, the creator of the *Gazette*, has run out of time to produce it and we struck a deal whereby *LG* can continue as a vendor-independent source of information. Its new home will be http://www.ssc.com/lg/, and its new editor will be Marjorie Richardson. She can be reached at info@linuxjournal.com.

In addition, a new editor and a new home, there will be other changes to the *Gazette*. While we will continue to offer an on-line version, we intend to include part of the *Gazette* in the pages of *Linux Journal*. We will offer the information we consider of the greatest interest, and pointers to additional on-line information.

Novice-to-Novice

The way our community grows is by getting new people up to speed. We used to have a novice column. It just sort of faded away. We knew it was needed and with the introduction of “Novice-to-Novice”, we've done something about it. John Fisk is writing some articles for the series, as well as at least one other author. You can suggest new topics by sending e-mail to info@linuxjournal.com or by writing to us.

Tech Answers

When each new Linux distribution comes out there is a flood of new questions. We have started a tech answers column where vendors and consultants will answer the common questions that arise. If you have a question, you can send it to info@linuxjournal.com, mail it in or fill out a form on our web page.

More product reviews

The most common question for a beginner is: “What should I buy?” For someone who has been working with Linux for some time, this is a common question as well—the query is just more likely to be about an ISDN board or scanner than a Linux distribution. We are encouraging vendors to send us products to review and hope to keep up with the new product releases so we can help you make product selections.

More pages

Finally, we are producing a larger magazine. That means we will have more space for the new columns and more reviews. What makes this possible is more income. We are growing as fast as we can right now, but we need you to do your part. Buy your boss a subscription. Tell a fellow Linux user about us. And when you buy something for Linux tell the vendor that you found out about him in *Linux Journal*. That's the way it all happens.

And more

We have other changes in the works as well. Watch for information on them in the December issue, or watch for news flashes on our web site. With your help we can continue to be the Linux resource you need as well as a tool that helps show others that Linux has become a real part of the computing industry.

Archive Index Issue Table of Contents

Advanced search

Keyboards, Consoles, and VT Cruising

John M. Fisk

Issue #31, November 1996

There are times when the command line interface is still a very good choice for getting things done.

"It's a GUI, GUI, GUI, GUI world!"—or so the major OS manufacturers would have you believe. The truth is that while this is increasingly the case, there are times when the command line interface (CLI) is still a very good choice for getting things done. It's fast, generally efficient, and is a good choice on memory or CPU constrained machines. And don't forget, there are still a lot of very nifty things that can be done *at the console*.

In this spirit, I'd like to start by following up on a delightful and informative article written by Alessandro Rubini entitled "The Best Without X" in the November 1995 issue (#19) of *Linux Journal*. Among a wealth of helpful ideas, Alessandro suggested converting the numeric keypad into a *console-switch scratch pad* to allow single key switching from one virtual terminal (VT) to another. We'll begin by looking at how this conversion can be done. We'll also look at:

- Getting from Here to There: handy methods for VT cruising
- The Useful Unused VT: where to put all that logging information, and where X-Windows really ends up

By the time that you get through tinkering around with these things I think you'll agree that the CLI isn't such a bad place after all. Also, the good news is that the programs you'll need to do this conversion are standard inclusions in most recent Linux distributions and include:

- kbd 0.91 (keyboard font and utility programs)
- utils 2.5 (Rick Faith's huge collection of utilities)
- GNU shell-utils 1.12 (shell utilities including the **stty** program)

A listing of Linux FTP archives where these utilities can be found is included in the sidebar.

The Keypad VT-Switcher

The numeric keypad is an ideal candidate for re-mapping into a virtual terminal-switching scratch pad since most of us have never learned “ten-key by touch”. In addition, the non-numeric functions on a 101-key keyboard are already duplicated by the home, end, page up, page down, insert, delete and arrow keys. Since there may be occasions in which we still want to use the keypad for numeric input, let's see how to set it up as a VT switcher while retaining numeric input ability. You'll need to have the kbd package installed on your system. The two programs we'll be using are **showkey** and **loadkey**. To check whether they are installed on your system type:

```
$ type loadkeys showkey
```

if you're using the BASH shell, or:

```
$ which loadkeys showkey
```

The **which** program or the BASH shell built-in function **type** will both print the path to the executable if they exist in the PATH search path. On my machine this produces:

```
$ type showkey loadkeys
showkey is /usr/bin/showkey
loadkeys is /usr/bin/loadkeys
```

```
$ which showkey loadkeys
/usr/bin/showkey
/usr/bin/loadkeys
```

If you don't have these programs installed, you'll need to get the the kbd package source, and install it yourself. This package is available only as source code, but installation is as simple as un-archiving it into a temporary directory, then typing:

```
$make && make install
```

Converting the keypad into a VT switcher involves defining a keyboard mapping and using loadkeys to actually load this information into the kernel keyboard translation tables. It's easier than it sounds—although you must keep in mind that indiscriminate tinkering can render your keyboard useless (requiring one of those dreaded cold reboots), and that changing the keyboard translation tables affects **ALL** VTs, not just the one you're working on. The kbd package's default installation location is under /usr/lib/kbd, with the key mapping files in the keytables subdirectory. Change to this directory and make a copy of the

defkeymap.map file, which contains the default keyboard mapping and is a useful place to begin. You can name the new file anything you'd like—e.g.,

```
cp defkeymap.map custom.map
```

Use your favorite editor and load up the copied file. At this point it's probably helpful to have a look around at the current contents. The experience is rather like visiting one of those fine old curio shops—look, but don't touch! The first few lines may look something like this:

```
keycode   1 = Escape  Escape
    alt keycode   1 = Meta_Escape
keycode   2 = one  exclam
    alt keycode   2 = Meta_one
    shift   alt keycode   2 = Meta_exclam
keycode   3 = two   at   at
    control keycode   3 = nul
    shift   control keycode   3 = nul
    alt keycode   3 = Meta_two
    shift   alt keycode   3 = Meta_at
```

I won't go into all the gory details of how to re-map the keyboard except to say that the basic format to use is:

```
keycode keynumber = keysym
    modifier keycode keynumber = keysym
```

in which *keynumber* is the internal identification number of the key and *keysym* represents the action to take. Now, before you bail out on me, let's put this into simple terms. Each key on the keyboard is identified by a unique number which is represented by *keynumber*. When a key is pressed or released, the press or release event is passed to the operating system, which responds by performing the appropriate action—represented here by *keysym*. The *modifier* is a key which is held down at the same time that the key is pressed. These *modifier* keys include the well-known control, alt and shift keys. The ability to define multi-key combinations extends the mapping available for each key.

So, using the example above, pressing the key associated with keynumber 3 actually causes the number **2** to be printed to the screen. If the shift key is held down at the same time as the key is pressed, the **@** sign is printed to the screen, and if the three key combination shift-alt-3 is pressed, the output is the Meta_at (whatever that looks like).

Getting back to the task at hand, we want to change to a specified VT when we press one of the keypad keys: i.e., pressing keypad 1 should switch to VT number 1, pressing keypad 2 should switch to VT number 2, etc. In your customized key map file find the section that defines the keypad keys—it should look similar to this:

```
keycode  71 = KP_7
    alt keycode  71 = Ascii_7
keycode  72 = KP_8
    alt keycode  72 = Ascii_8
keycode  73 = KP_9
    alt keycode  73 = Ascii_9
[...]
```

Now, edit this section so that it reads something like Listing 1.

Before continuing, let's make a couple of observations. First, it's not a bad idea to comment the file as you go. What seems clear and obvious now fades into obscurity as the weeks pass. Adding comments now will prevent your having to pore over manual pages, program documentation and magazine articles later, looking for the correct syntax or usage. Second, notice that with each entry there are *sub-stanzas*, beginning with the words **alt keycode**, **shift keycode**, etc. These stanzas define multi-key combinations in which a *modifier* key is pressed at the same time as the key being defined. A common example of this is the crtl-c combination used to terminate a program during execution.

Finally, you may be asking yourself how you're supposed to know which keynumber is associated with a key. Does anyone know off-hand what keynumber goes with the **;** key? You can find this out by using the showkey program. After you invoke the program, showkey will print the keynumber for any key you press and will quit after 10 seconds of no input. So, now that we've edited the pertinent section in the custom.map file, let's see how we'd arrive at this *from scratch*. The basic steps would be:

- Find the keynumber for the keypad keys.
- Edit the customized mapping for the keys so that pressing them would change to the appropriate VT.
- Edit the customized mapping for the keys so that the keypad could still be used for numeric input (using a modifier key combination in this case).
- Load the customized mapping and see whether it works.
- Optionally, have the default key mapping loaded at system boot.

To do this, let's begin by invoking the showkey program:

```
$ showkey
```

Now, any key you press causes showkey to print the keynumber. On my machine, invoking showkey and pressing keypad keys 1 through 9 results in the output shown in Listing 2. You can see that both key press and key release events are detected. Also note that the numbering of the keypad keys is not sequential. The numeric keys have the format shown in Table 1:

Table 1

```
Actual Key:                Keynumber:
7    8    9                71   72   73
4    5    6                75   76   77
1    2    3                79   80   81
```

Table 1 shows that keypad number 1 has keynumber 79, keypad number 2 has keynumber 80, etc, Knowing this, we can set up the appropriate key map entry for each of these keys. The keysym event that we're interested in is Console_*x*, in which *x* is the number of the VT to which the view is switched. A simple entry to map keypad number 1 to switching to VT 1 would look like:

```
keycode 79 = Console_1
```

If you look at Listing 1, you'll notice that this is what we've done. Suppose, however, we wanted to switch to a VT greater than 9—how are we to do that? The solution is to use a modifier key combination. Looking again at the example above, using the shift key with the keypad allows us to use Console_10 through Console_19. We also wanted to be able to use the numeric keypad as just that—a means of entering numeric data. In the example above, notice that the modifier **alt** was used to do this:

```
keycode  71 = Console_7
    shift       keycode  71 = Console_17
    alt         keycode  71 = KP_7
    alt control keycode  71 = Console_7
```

In this stanza for the keypad_7 key, the first entry maps the keypad_7 key to switch to VT 7. The second line maps shift-keypad_7 to switch to VT 17 and the third line maps the alt-keypad_7 combination to KP_7 which is the keysym for numeric output when num lock is *on*. Thus, to use the keypad as a numeric keypad, press the num lock key so that it toggles to *on*, then hold down the alt key while you enter numbers at the keypad. Note, too, that alt-crtl-*keypad* was defined to switch to the same console as simply pressing the keypad key itself. In this case, it acts in exactly the same fashion as the alt-f*n* (alt-*Function_key*) or alt-crtl-f*n* (alt-crtl-*Function key*) combination. You may have noticed that using the function keys is how one is typically instructed to switch from one VT to another. Looking at the stanzas for the function keys you'll notice entries such as the following:

```
keycode  59 = F1    F13    Console_13
    control keycode  59 = F25
    shift   control keycode  59 = F37
    alt keycode  59 = Console_1
    control alt keycode  59 = Console_1
```

Note that both alt-f1 and alt-crtl-f1 are used to switch to VT 1. Those of you using X will probably already have found that switching to a VT from X requires the three key alt-crtl-f*n* key combination while the two key alt-f*n* key combination is used at the console. Although you can change this default behavior, it's best not to. At this point, we've defined mappings for the keypad

keys such that each key acts as a switch to the VT of the same number. Using shift-*keypad_key* switches to VT (10 + keypad number) and using alt-*keypad key* with the num lock *on* outputs the numeric value of the key. The final step is to actually load the new mapping and give it a try. The loading is done using loadkeys and can be done without logging on as root. To load the customized keymap, enter:

```
$ loadkeys /usr/lib/kbd/keytables/custom.map
```

This will print a message indicating that the custom.map file is being loaded. After this, you're all set! Give it a try. To revert back to the default mapping simply enter:

```
$ loadkeys /usr/lib/kbd/keytables/defkeymap.map
```

and the default mappings will be loaded once again. You can use this edit -> load customized map -> test -> load default map cycle to obtain the desired mapping. Once you've created a custom map file and wish to have it loaded at boot, you can add an entry to one of the rc.* files, such as rc.local, to have loadkeys automatically load your customized mapping:

```
if [ -r /usr/lib/kbd/keytables/custom.map ]; then
     loadkeys /usr/lib/kbd/keytables/custom.map
fi
```

This entry ensures that the file is present and readable and invokes loadkeys to load the file. Again, keep in mind that loading a key mapping changes the keytable information for *all* VTs, not just your current one.

Getting from Here to There

Now that we're on a bit of a roll, let's look at another method for moving from one VT to another. The utility of being able to quickly switch from one VT to another should be obvious: you can be compiling a program on VT 1, editing a file on VT 2, reading program documentation on VT 3 and having a manual page displayed on VT 4. Now that you've re-mapped the keypad, switching from one VT to the next is as simple as pressing the keypad keys. But there are other handy means of getting around as well and these include:

- Using the keysym functions Last_Console, Incr_Console and Decr_Console
- Using the chvt program (which is part of the kbd package)

The Incr_Console and Decr_Console keysym functionsdo as their names imply: they switch to (VT + 1) or (VT - 1) respectively. So, if you were currently working at VT 3, the Incr_Console keysym would switch you to VT 4 while the Decr_Console keysym would switch you to VT 2. The Last_Console keysym also does as its name implies: it switches to the last VT used. If you were working at

VT 3 and switched to VT 6, the Last_Console keysym would switch you back to VT 3. You can map a key or modifier+key combination to invoke any of these keysym functions. I've mapped these functions as follows:

```
Ctrl+left arrow  = Decr_Console
Ctrl+right arrow = Incr_Console
keypad 0         = Last_Console
```

Obviously, you can map these functions in any manner you wish, but the relevant entries to map the above actions would be:

```
#keycode  82 = KP_0
keycode  82 = Last_Console
    shift keycode  82 = Console_10
    alt   keycode  82 = KP_0
[...]
keycode 105 = Left
    alt keycode 105 = Decr_Console
keycode 106 = Right
    alt keycode 106 = Incr_Console
```

These entries map the keypad 0 key to the Last_Console keysym and the alt-[left arrow] or [right arrow] to Decr_Console or Incr_Console keysyms. The good news is that these last two are already the default so that you have to edit only the stanza for the keypad 0 key. Now, you can quickly cycle through all the VTs by holding down the alt key and repeatedly pressing the left or right arrow. To alternate between two VTs you have only to repeatedly press the keypad 0 key. I've found these particular mappings to be quite useful but, as I mentioned before, they can be customized to anything you wish. The last bit of VT cruising magic is the chvt program included with the kbd package. Its use is quite simple:

```
$ chvt 3
```

would change to VT 3. Substituting another number for **3** allows you to change to that VT. A foreshortened version of this can be set up using a shell alias:

```
$ alias vt='chvt'
```

so that entering:

```
$ vt 3
```

would switch you to VT 3.

So, now that we've defined several methods of getting from VT to VT it is important to note that this works only at the console and not under the X Window System. Under X, the X server takes control of the keyboard, mouse, and display: setting up customized keyboard mappings is performed using the ~/.Xmodmap file or the program xkeycaps and is a subject for a later article.

The Useful Unused VT

Having the capacity to open multiple VTs and to have programs running on them in the foreground or background is one of the things that makes running Linux such a huge amount of fun. As the old Surgery Prof used to harangue his interns, “Help me, help me! If I had another set of hands I'd help myself!” Linux gives you that extra “set of hands”. Generally, most VTs, to be useful, must have a **getty** process running on them in order to log in. A getty is a program associated with a terminal that:

- Opens the tty line and sets its mode.
- Prints the login prompt and gets the user's name.
- Initiates the login process for the user.

Without going into all the details (again, a subject for a later article), suffice it to say that this program is set up in the /etc/inittab file. An entry for a getty might look like Listing 3.

The important thing to note in this listing is that the agetty program is run on each of the tty devices from tty1 to tty6. Thus, at system startup there are a total of six gettys running, allowing you to log into VT 1 through 6. So what about VT 7 and beyond? Are they still usable in any way? If you've re-mapped your keyboard—try pressing keypad_7—alternatively, press alt-f7—and see what happens. In general, the screen is blank with the cursor positioned at the upper left corner. You can type at the keyboard, and the output is displayed on the screen. Despite this, there is no way to execute programs at this terminal. A terminal you can't log in on isn't much use. There are, however, two important exceptions to this statement.

So Where Did X Go?

The first exception to note is that when the X Window System starts, it is displayed on the first unused tty—one that doesn't have a getty running on it. Since the first six ttys had gettys running on them, X would, in the example above, start on tty 7. Now we know the solution to the great riddle, “So where is X ?”, when you switch from X to a console. Pressing crtl-alt-f1 in X would switch you to VT 1. If you wanted to get back to X, simply:

- press keypad_0 if you've mapped this to the Last_Console keysym.
- press keypad_7 to switch to VT 7 on which X is running.
- press alt-f7 to switch to VT 7.

Putting That Unused VT to Work

The other exception to note is that while you can't run programs on a VT without logging in, you can still send output there. As a simple experiment try the following:

```
$ echo "This is a test" > /dev/tty7
```

Switching to VT 7, you'll see the words “This is a test” displayed. This ability becomes useful with system logging. Without going into an exhaustive discussion of system logging and configuration, it is worth noting that the output of all logging facilities can be “dumped” to an unused VT which allows quick perusal for events such as logins, kernel messages, mail logging, etc. To do this simply add the following line to the /etc/syslog.conf file (after logging in as root):

```
# this one will log ALL messages to the
#/dev/tty9 terminal since this is an unused
# terminal at the moment. This way, we
# don't need to hang a getty on it or take up
# a lot of system resources.
*.*                     /dev/tty9
```

Once you've added this stanza to /etc/syslog.conf, you'll need to either kill and restart the syslog daemon or else send it the HUP (hang up) signal. Since this latter method is fairly easy let's do it:

```
$ ps -x | grep syslog
28  ?  S     0:01 /usr/sbin/syslogd
```

will output the PID (process ID number) of the syslog daemon which in this case is 28. Now, just type in:

```
$ kill -HUP 28
```

in which **28** is the PID number. The syslog daemon will re-read its initialization files. From here on, all logging that occurs, regardless of its source, will be output to tty9 (or whichever tty device you specify).

Switching to VT 9 you might see something like the following:

```
Jul  1 10:11:37 FiskHaus kernel: Max size:342694   Log zone size:2048
Jul  1 10:11:38 FiskHaus kernel: First datazone:68   Root inode number 139264
Jul  1 10:11:38 FiskHaus kernel: ISO9660 Extensions: RRIP_1991A
Jul  1 12:21:50 FiskHaus login: ROOT LOGIN ON tty2
Jul  1 17:26:56 FiskHaus login: 1 LOGIN FAILURE ON tty5, fiskjm
```

The first three lines represent kernel messages that occur when a CD is mounted. Root logins are noted by the **login** program as well as login failures—in this last case I purposely entered an incorrect password.

The value of all of this logging may not be immediately evident, but if you've ever noticed your machine thrashing about and swapping like crazy, or, while on-line, your hard drive lights begin to light up when you're not doing anything —a quick switch to VT 9 can often give you an idea about what's going on. These instructions should get you started. The manual pages for loadkeys, showkey and keytables have much more complete technical descriptions of key mapping. Also, the kbd package comes with a good deal of helpful documentation in its /doc subdirectory. And finally, don't forget the *Keyboard-HOWTO* which can be found among the growing number of Linux HOWTOs http://www.ssc.com/linux/howto.html).

John Fisk (fiskjm@ctrvax.vanderbilt.edu) After three years as a General Surgery resident and Research Fellow at the Vanderbilt University Medical Center, he decided to “hang up the stethoscope” and pursue a career in Medical Information Management. He's currently a full time student at the Middle Tennessee State University and working on a graduate degree in Computer Science before entering a Medical Informatics Fellowship. In his dwindling free time, he and his wife Faith enjoy hiking and camping in Tennessee's beautiful Great Smoky Mountains. An avid Linux fan since his first Slackware 2.0.0 installation a year and a half ago

Archive Index Issue Table of Contents

Advanced search

Debian 1.1

Phil Hughes

Jonathan Gross

Issue #31, November 1996

The Debian packaging system makes it possible to do add, remove and upgrade automatically.

With Linux 2.0 out, commercial vendors are offering new products based on the 2.0 kernel. We want to keep everyone up to date, so we will be reviewing the various distributions. An updated chart (based on the one that appeared in Issue 29 of LJ) and new descriptive text will be made available on the *Linux Journal* web page at http://www.ssc.com/lj/distable.html.

by Phil Hughes and Jonathan Gross

The first distribution made available with the 2.0 kernel was Debian, and thus it is the first we will review. We would like to thank iConnect for making the CD available to us. iConnect's web page is located at http://www.i-connect.net/i-connect/services/cdrom.html.

Some Background...

> From the Debian FAQ:
>
> Debian GNU/Linux is the result of a volunteer effort to create a free, high-quality Unix-compatible operating system, complete with a suite of applications. The idea of a free Unix-like system originates from the GNU project, and many of the applications that make Debian GNU/Linux so useful were developed by the GNU project. Debian was created by Ian Murdock in 1993, initially under the sponsorship of the Free Software Foundation's GNU project. Today, Debian's developers think of it as a direct descendent of the GNU project.

Installing Debian

Enter. Enter. Enter. Enter. Enter. Enter. You could pretty much train a chicken to install Debian: “Peck the enter key. Wait....okay, peck enter again...wait...okay, now peck enter...” I had time to think about this as I was installing Debian for the first time on an old 386 with no math co-processor. By the time I had installed it on my P60, I had an entire training session for my chickens, with plans to take over entire networks. (“Now it's upgrade time! Select FTP from the Access menu and peck enter....now peck enter again...”)

Debian installation begins with a set of floppies. I needed six: a base system spans three of them, a root, a boot and a sixth to make a backup boot disk.

The base system is installed, and from there, you go through the package selection process.

I've installed a fair number of Linux distributions, built a couple from scratch (no chicken pun intended) and bollixed up more than one installation—Slackware, Red Hat, SLS (shudder) and others. I've noticed that one of the things I do that causes problems is deciding to not install package foobar. It's big, I don't like the package name, the version number is 13, so I don't install it. Then I run Idependonfoobar, and it complains, “...can't find library libfoobar.” Phooey. So I waste a lot of time only to discover that the packages I left out contain files that other programs depend on.

Debian solves this problem for me, or at least warns me that I am making a mistake. Debian's package dependencies are very, very cool, and will be discussed at length.

Installation was self-explanatory until I came to the package selection screens, where I stumbled a bit, forging ahead without *any* documentation. I found I had to rethink my chicken training regime. The selection process reminds me of reading mail with trn: there are a couple of different screens that do different things, and a slew of keyboard controls for doing them. Is this part of the cost of having dependencies? I don't know, but I think that part could have been clearer. Once you figure out how things work, it **is** very helpful, but it is certainly not intuitive, like the rest of the installation.

For example: I decided not to install Tex. It's big, and I would rather have used that disk space for my large collection of gifs. So I left Tex unselected. Later, I chose apsfilter, a print filter that needs dvips to work its magic. Debian's package selection tool, **dselect**, told me that apsfilter depends, among other things, on dvips. That's cool. But even cooler is the fact that all these dependencies are presented in a list form that allows me to select packages

from the list during the install, solving the dependency problems as they arise and without losing my place in the installation. Very cool.

Another example: I initially chose the Debian default mail transfer agent, smail, but then decided I really wanted sendmail in all its glory. I went back into dselect, and selected sendmail.

dselect complained that smail conflicted with sendmail, but I ignored it and tried to install it anyway. deselect wouldn't do it because of the conflicts with smail. You may consider it a blessing or a curse when software makes decisions like this for you. I admit it was slightly irritating that deselect wouldn't let me hang myself. Nevertheless, this sort of thing prevents damage to innocent hardware when people cannot get their new systems to work because of software conflicts and heave the machines out windows.

When I went back and heeded the warning messages, deselected (dselected?) smail, and tried again, smail was removed, and sendmail was installed and configured. During installation, you are prompted to configure sendmail to send mail to a hub, or run as a host. This option is nice for people who have simple mail needs.

Another of the best features of this distribution is the upgrade method—truly the most impressive feature of Debian. After you have the system basics installed and running, you can fire up dselect and do upgrades to your system. If you are connected to the Internet, you can do this via FTP. dselect allows you to select where you are getting your new packages—so you point it at ftp.debian.org. The upgrade acts just like the installation, the system updates the packages database, and gets all the newest packages from the FTP site, and continues with the install.

No more guessing which parts of the distribution you need to upgrade to run the new version of foobar—you just point dselect at the Debian FTP site and let the system worry about it—this is what computers are for—keeping track of all the “this package depends on this stuff, and you'll need to get the new version of blah-blah-blah.” Debian does all that for you—**very impressive**.

Debian allows you to install (and update) from CD-ROM, NFS, hard drive, previously mounted partition, floppy, or via FTP. I installed from the CD-ROM obtained from iConnect.

However, there are problems with the system. Apparently the installation procedure doesn't install LILO correctly. Unless you run LILO by hand before rebooting, it won't re-boot from the hard drive at all, even if you told the installation scripts that was what you wanted. Also, the modules do not load

into the kernel. All these problems need to be dealt with before Debian is ready for the Big Time.

I also tried to get and install the kernel sources for 2.0.6 (the latest package as of this writing). The kernel building procedure is a little different for Debian than for other distributions, and I still don't have it working properly. The kernel builds, then some scripts run to make sure that if you upgrade your kernel packages, you don't overwrite your custom kernel configuration. These scripts don't seem to work correctly—the kernel building procedure aborted and I had to finish it manually, which is not a big deal *if you know what you are doing*. But these bugs are not trivial and should be fixed (and may be, by the time this article is printed).

More on the Debian Packaging System

The Debian packaging system makes it possible to do add, remove and upgrade automatically. You may have noticed that I said “makes it possible”, not that it always works. If the package is properly constructed, it will automate all these tasks for you. But it's up to the person or vendor building the package to ensure it works. The scripts in the package are also very important to help you with package management. In addition to the pre-install and post-install scripts mentioned earlier, there are also pre-remove and post-remove scripts.

Commercial Debian?

Debian remains an “almost-commercial” distribution. By this I mean that you can buy it on a CD, but there is no company that sells and supports it, and as far as I know, no commercial software packages are available in the Debian format.

Sidebar: More Debian Information

This situation could change. As I write, InfoMagic is considering making Motif available in Debian format. Also, as the official keeper of Matt Welsh's *Linux Installation and Getting Started*, we are considering writing an installation chapter for Debian.

Only time will tell if the Debian format will make serious inroads into the commercial Linux market. I hope it does.

Phil Hughes is the publisher of Linux Journal and Jonathan Gross is a technical editor at Seattle Software Labs.

Phil Hughes is the publisher of *Linux Journal*

Jonathan Gross is a technical editor at Seattle Software Labs.

Archive Index Issue Table of Contents

Advanced search

LINUX JOURNAL

MkLinux—Linux Comes to the Power Macintosh

Richard C.S. Kinne

Issue #31, November 1996

Macintosh users can now come in from the cold—MkLinux has arrived. Here's how it happened, and how you can get it.

May 17, 1996 was a day of celebration among people who are both Macintosh and Linux devotees, for it is the day Apple and the Open Software Foundation let loose the first Developer's Release of the anxiously awaited MkLinux. MkLinux is a PowerPC-native version of Linux that operates on top of a MACH kernel. Back in February, Apple and the Open Software Foundation announced they would work together to port Linux to the PowerMacintosh platform. Since then, the progress of the teams working in Grenoble, France and in Cupertino, California has been anxiously followed.

Keeping up with the project was made easier with the creation of mailing lists and a web site run by Nick Stephen, one of the project's programmers laboring in Grenoble. While obviously very busy, Nick kept an impatient community at bay with progress reports and patient answers to both simple and complex questions.

While being harassed on an almost weekly basis to “release the code so we can begin hacking on it”, the OSF and Apple teams were adamant that nothing would be released until it was at least marginally stable—the growing MkLinux community would have to wait.

On May 17, that wait ended as MkLinux DR1 was released, both on CD-ROM to the attendees at Apple's World Wide Developer's Conference and to the world at large via the Internet. Proving they were on top of what was to become a small whirlwind of discussions and reports, Apple switched the old OSF mailing list to one of their servers and created a bevy of additional mailing lists devoted to specific topics. Apple also hosted the project's main web site, http://www.mklinux.apple.com (Figure 1), and one of several FTP sites where the MkLinux DR1 release could be downloaded onto waiting hard drives. For a

developmental release, it was a professional roll-out. Now the question on everyone's mind—"Was the wait worth it?"—could be answered.

Figure 1. MkLinux World Wide Web Site

The Price of Admission

The admission price, in terms of computing resources needed for the first MkLinux Developmental Release, is somewhat steep. This is not a Linux you are going to be able to cram into 2MB of RAM and a 40MB hard drive. Apple recommends that those brave persons who try installing DR1 have at least 16MB of RAM and 400MB free on their hard drives. Also, according to Apple, an entire hard drive should be devoted to your MkLinux partitions, if possible. Your choice of platforms is also somewhat limited. At this time, MkLinux runs only on NuBus PowerMacintoshes using the PPC 601 RISC chip. The 603 or 604 chips are not supported, nor is the PCI bus. That situation will soon change.

The packages that have to be procured for the installation are also huge. Apple originally put them on the Internet at ftp://mklinux.apple.com/pub/MkLinux_DR1 for transfer and within a week, other sites had them available for download. Packaged as stuffed MacOS SIT files, the initial MkLinux distribution file, MkLinux_DR1.sit, is 42MB in size. Uncompressed, this expands to over 120MB! Additional packages, all several megabytes in size, are also available for the MACH source code, the Red Hat Package Management (rpm) source packages, the rpm binaries, and the X11 distribution. All in all, while you only *need* the 42MB MkLinux_DR1.sit file to get started, the full distribution weighs in at nearly 200MB of material.

Realizing that this could prove to be a severe problem to many of the people who want this initial distribution, Apple made a deal with Prime Time Freeware (http://www.ptf.com) to release to the general public the same CD-ROM given out to the attendees of Apple's World Wide Developer's Conference. By sending $10 plus shipping and handling to Prime Time Freeware, you'd be saved the time and aggravation of having to download several megabytes from an increasingly fickle Internet. Given the weight of the distribution, many people opted for the CD-ROM in getting this first version of MkLinux.

Of Partitions and Installations

Once you have the required packages, either after downloading them from the Internet or getting them off the CD-ROM, what happens? The first job is to format and partition your drive. The Apple/OSF team wrote MkLinux to use the same type of partitions as Apple's A/UX product in order to make it (theoretically) easy to partition your drives, since most Mac partitioning software supports creating A/UX partitions. As with any developmental release,

for some of us this worked, and for some of us it didn't. As an example of the support the community gave not only itself but the MkLinux team, Philip Machanick of the University of the Witwatersrand in South Africa kept track of those types of hard drives and partitioning programs people reported were working and not working with the DR1 release. By the time you read this, the need for keeping such a list should be long past. Apple recommends that you partition at least a 300MB root partition. This can be divided into a 100MB root partition and a 200MB /usr partition, if you want to. You must also have a swap partition that is at least 32MB in size, but not larger than 64MB. Because of how Uhix type “upgrades” are done, it is best to separate the root and /usr partitions.

Once the drive partitioning is completed—a job that Apple's release notes call the most difficult process of the installation—you run the “Install MkLinux” application to begin the actual file copying process. When this Macintosh application runs, it asks several questions. Most importantly, the installer needs to know the SCSI drive number of the hard drive with the MkLinux partitions. Once that is secured, it goes on to find each of the partitions you've created and asks if you wish to use them. The application is so careful in making sure you want to do what you say that some users felt they were put through an electronic version of the Inquisition during installation. However, once all the questions are answered to the satisfaction of the installer, it copies the needed files to the newly created MkLinux partitions. It also modifies the Macintosh System Folder by adding one Control Panel and two System Extensions. Together, these files act as the Macintosh version of LILO, enabling you to specify which OS will boot up on system startup. The entire installation, once started, takes about fifteen minutes to complete.

Once the files are copied, you need to invoke the new MkLinux Control Panel to tell the Macintosh which OS you'd like to use for booting. At present, the options for the Mac LILO program are rather limited. When you bring the Control Panel up, you have a choice of selecting one of two radio buttons: MacOS or MkLinux (Figure 2). Once you make this selection, there is a button available to enable rebooting right then and there.

Figure 2. MkLinux Control Panel

Depending on which radio button you choose in the MkLinux Control Panel, a couple of different things can happen. If you choose to reboot MacOS, you won't even know MkLinux is available to you (save for the space its partitions consume on your hard drive). If, however, you're brave and elect to boot MkLinux, very early in the Macintosh boot process, the MkLinux LILO screen appears on the screen. This colorful and well-constructed dialog box (Figure 3) gives you the choice of booting MkLinux or booting MacOS. The countdown in

the lower right-hand corner of the dialog box gives you ten seconds to make up your mind before, by default, the machine boots MkLinux.

Figure 3. MkLinux Dialog Box

Assuming you boot MkLinux, your screen turns black, and the white-lettered lines of a Linux system booting up scroll up your screen. After a short pause, the screen will again clear and print:

```
MkLinux for Power Macintosh. Brought to you by Apple Computer, Inc.
Kernel 1.2.13 on a osfmach3_ppc
login:
```

Congratulations! You're now running native Linux on a PowerMacintosh!

Your Mileage May Vary

Despite being distributed by Prime Time Freeware, MkLinux is not yet ready for prime time. As already noted, many people have had problems right from the start, being unable to even boot from their drives. Sources at Apple say this problem will be solved by the time you read this article, but if you're considering purchasing a new drive, you will want to check whether the one you're considering has been used successfully by someone else before spending money on it.

Video support in the first Developer's Release is a bit sparse. Only on-board video and the HPV card are supported, and this has caused some problems with people who have “AV” Macintosh systems.

Floppy drives and, more unfortunately, serial output, are not supported with this first release. Thus, while you can play around with networking if you have access to Ethernet, those of us who connect to the Internet via PPP will have to wait a bit. The lack of serial support also limits printing options.

On the SCSI bus, only hard drives and CD-ROMs are supported at the moment. The release notes say other devices, such as the Iomega ZIP drive, have not been tested, but I have not gotten mine to work, and I know of no one on the Internet who has.

Finally, as with any developer's release, your mileage may vary with respect to getting various programs and systems working. For example, while I have not gotten Emacs to work, I know of several people who've had no problem with it. On the other hand, Apple's own Errata, as of May 25, mentions a problem regarding a shell script that will cause you to be logged out the first time you log on as root; I have never encountered this problem.

However, the MkLinux teams at Apple and OSF got a lot of things right. The installation procedure (assuming you have a MkLinux-friendly drive) is one of the smoothest installations I've ever been through for a software package of this size. Considering this is a developer's release, it has been remarkably stable. While there have been some surprises, usually either some work-around has been developed or the situation is put right on the “to-do” list by the Apple/OSF teams.

MkLinux's Future Shines Bright

According to Michael Burg at Apple, MkLinux will go through at least one more developer's release, scheduled near the end of the summer, before the Reference Release is distributed in September. The MkLinux world has proven that it moves as quickly as the Intel Linux world, with updates and patches appearing on Apple's FTP site ftp://ftp.mklinux.apple.com/pub/) on a weekly basis. According to a schedule that Michael Burg released to the Internet in early June, most of the bugs and omissions from DR1—such as video console and driver issues, SCSI driver bugs, and the lack of serial support—should be solved and implemented by the time you read this article. PCI bus support is scheduled for the Reference Release with support for the PPC 603e platforms coming some time in autumn.

Sidebar: Prime Time Freeware

MkLinux Discussion Lists from MKLinux FAQ

After autumn, what's next? To a large extent, like any Linux, that depends on us. Apple and the OSF have released the full source code for this project to anyone who wants it, respecting the spirit that has guided Linux since Linus Torvalds first released it. Some Intel Linux hackers have wondered whether there is enough of a critical mass of MkLinux programmers to keep the project alive. Based on the beginnings of the community that has come alive around this first developer's release, I don't think we'll disappoint our Intel brethren.

Sidebar: As We Go To Press

The Macintosh is a computer which, through its eleven years of life, has inspired a lot of love and dedication. With MkLinux, we have the opportunity, as the saying goes, to “fall in love all over again.”

Richard Kinne (kinnerc@snymorva.cs.snymor.edu) is using the MkLinux project to re-acquaint himself with the Unix operating system after having been exiled to VAX/VMS-land for ten years. He works as the User Services Consultant for the State University of New York at Morrisville. When not writing or hacking

with his significant other, he enjoys *Star Trek*, *Babylon 5* and playing with his cats.

Archive Index Issue Table of Contents

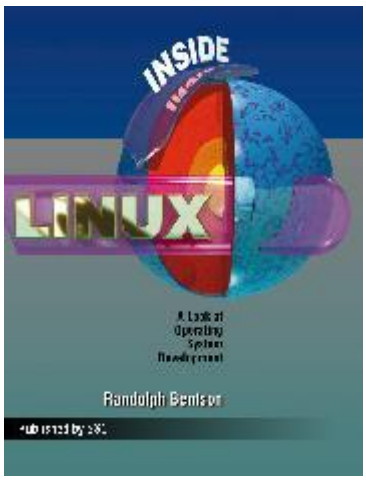
Advanced search

Inside Linux: A Look at Operating System Development

Richelo Killian

Issue #31, November 1996

I was expecting a book that really digs right into the Linux kernel, and would give me some insight into why Linux works as great as it does, but that's not what I found.



Author: Randolph Bentson

Publisher: Specialized Systems Consultants (SSC)

ISBN: 0-916151-89-1

Price: $22.00

Reviewer: Richelo Killian

The title of this book, *Inside Linux: A Look at Operating System Development*, is a bit misleading. I was expecting a book that really digs right into the Linux kernel, and would give me some insight into why Linux works as great as it does, but that's not what I found.

Please don't misinterpret the above paragraph—Dr. Randolph Bentson has written an excellent book here. I just think the title is a bit misleading. Now that I have that off my chest, let's have a look at what the book **does** offer. The book is divided into three parts:

- Computing Today
- The Linux Programmer's View
- The Advanced User's View

Chapter 1 discusses why Linux is becoming more popular, pointing out both its advantages and its shortcomings, although even those are looked at from a Linux-fanatic's point of view. This chapter alone makes buying the book worthwhile, if only as ammunition to convince your boss you need Linux in your organization.

In the next three chapters, Dr Bentson gives a fairly detailed history of the development of Linux, covering, among other topics: security rings, the shell, file system, timesharing, virtual memory, the X client, the X server, scheduling, synchronization, memory management and kernel security. These issues are discussed from the initial development of Linux, and why many of these elements are implemented differently in Linux than in any other system. Each system element is covered in sufficient detail, but not so much as to become boring. Dr. Bentson has done an excellent job of filtering through all the material available for these different areas of operating system Development, but just in case there are some people out there who want more information on one or more of the topics discussed, he includes more than enough pointers to sites on the Internet to satisfy even the most avid reader.

Part II, “The Linux Programmer's View”, gets a bit deeper into the actual kernel. Chapter 5 discusses the operating system kernel, concentrating on issues such as: user interface, process control, input/output, memory management, security, standards and “bootstrapping the kernel”. Each section is full of example code which better explains the concepts presented. All of this explanation is done in a mere 80 pages—I would have liked a more in-depth discussion of the inner workings of the kernel.

Chapter 6 looks at the networking side of the kernel and gives a brief overview of the ISO OSI Model, TCP/IP, UDP/IP and IP & ICMP. This chapter is very informative, mostly covering information that can also be found in *The Linux Network Administrator's Guide*. Part III starts off by looking at development tools, but does not discuss any specific tools, concentrating instead on the philosophies behind the tools.

Chapter 7 discusses languages, editors and file management tools. Chapter 8 goes into quite a bit of detail on the hardware supported by the Linux kernel. These chapters would be very useful to a new user in making decisions as to hardware and software tools. The book concludes with a quick run-down of available distributions—a good reference for someone looking to buy, as long as the rapid rate of change in software today is kept in mind. Covered in this last chapter are dual tracked kernel releases, distribution kits and CD-ROM publishers. Appendix B contains a concise list of system calls, and Appendix C consists of a nice time-line of Linux development to date. Overall I would say that this is an excellent read for the accomplished Unix User. Part II, however, could be a bit too advanced for the newcomer to Unix.

Dr. Bentson emphasizes that the book was written in LaTeX, and converted to HTML with LaTeX2HTML. It would be nice if SSC could include the book on CD in HTML format to make following the URLs easier.

Richelo Killian (ftcs@icon.co.za) is the Unix system administrator at the University of Cape Town, Electrical Engineering Department, South Africa. He looks after and administers Sun, IBM, HP and Linux machines. He started in Linux with the first Yggdrasil release. He is also the coordinator of the Linux Promotion Project, hosted by Linux International.

Archive Index Issue Table of Contents

Advanced search

etags

Dave Thomas

Issue #31, November 1996

Have you gotten used to hypertext? Want the same mobility in your source code and other documentation? etags may be your answer.

You've probably used hypertext links to browse the Web, skipping easily between pages with simple keystrokes or mouse clicks. With etags you can browse your source programs and documentation just as easily. In this article I'll describe tag files and the etags and **ctags** commands that generate them. I'll also look at how **less**, **Emacs** and **vi** use tag files to make your editing easier and more productive.

etags and Emacs

By default, etags takes a collection of source files and generates an index of the definitions of all the *global* symbols of interest. For C and C++ programs, this index includes global and member functions, classes, structures, enums, typedefs and **#define**s. For (La)TeX documents, it indexes chapters, sections, subsections, figures, equations etc. etags also includes built-in support for assembler, Fortran, lex, Lisp, Pascal, Sheme and yac. **etags --help** will give a complete list of languages supported by your version.

Try running **etags *.c *.h** in one of your source directories. After it finishes processing, you'll find that it has produced a file called **TAGS**. This is a simple text file, with a block of entries for each file parsed. Each block contains a line for each definition in the file, with the text of the definition and the line number and absolute character position on which it was found.

How do you use this? Well, the simplest way is from within Emacs. You start editing as normal in the source directory. When you come across a call to one of your functions, and want to see how it was defined, position the cursor on the call, and enter the Emacs command **M-.** (press the **esc** key followed by a period, or hold **alt** and period down at the same time). Emacs will display the

name of the function in the mini-buffer. If you press **return** to confirm, Emacs will automatically open the file containing the definition and position the cursor at its start. Suppose while looking through that function, you are puzzled by a declaration which uses a typedef name. Move the cursor to it, press **M-H.** again, and up pops the typedef. If you can't find the a **#define** you created, use **M-.**, then type the macro name in the minibuffer. Emacs looks it up in the tags table and takes you straight to it.

Sometimes you'll have more than one tag containing the same text, and **M-.** will take you to the wrong place. Simply keep pressing **C-u M-.**, and Emacs will move through all possible tag matches. If you're finger-tied like me, you may find **M-0 M-.** a simpler way to enter a prefix for the **find-tag** command. Entering a negative prefix (**C-u -M.**) takes you back to a previous definition.

That's not all you can do with the TAGS file. Imagine that when you were typing in some source you decided to give your functions long, descriptive names. It seemed like a good idea at the time but now the glow is wearing off as you're typing a call to that function for the nth time. Again, tags can help. Type the first few characters of the name, then press **M-tab**. Either the full function name will appear, or a window will pop up displaying a list of possible completions. (This might not work for XEmacs users, and you may have to re-bind **M-tab** to **tag-complete-symbol**.)

You can also use tag completion in the minibuffer whenever a command prompts for a tag—type in the first few characters, press **tab**, and Emacs will do the rest.

If you're an XEmacs user, the **M-?**i command displays a single line summary of a function or typedef in the modeline—really useful if you've forgotten the calling sequence!

Tags are also a great way to perform search and replace operations across all your source files. The **tags-search** command prompts you for a regular expression, then displays the first match in any of the files in your TAGS table. You can move on to successive matches using **M-,** (that's *meta-comma*). The **tags-query-replace** performs a egex-replace across all the files in the TAGS table. You can stop it at any point (with **escape** or **C-g**) and later resume it with **M-,**.

Finally, the **tags-apropos** lists in a separate window all tags matching the regular expression that you enter—a great way to scan for stuff in a hurry.

ctags, vi and less

In the same way that etags works with Emacs, the ctags program generates tag files that can be used with vi, **view** and less. The basic operation of the command is the same, but it generates a file called **tags** (in lower case). Once it's finished, you can go straight from the command line to a particular function in either vi or less using **vi -t***name* or **less -t***name*. You don't even have to give a file name!

If you're editing with vi, you can move to a tag using the **:tag** command.

Keeping up to Date

Surprisingly, the **TAGS** and **tags** index files remain valid even if you insert and delete lines in the files they reference. You really need to run etags/ctags only when you add or remove functions or files. I find it convenient to have a **tags:** target in my Makefiles for this:

```
tags:
	etags $(SRC)
```

If you have files in many directories, you *could* generate a single tags file covering them all by specifying directory names on the command line. This works fine in vi and Emacs, but you'll need to set up either the **tags-file-name** variable or **tag-table-alist** if you're an XEmacs user. I personally find this pretty clumsy, and tend to stick to a TAGS file per directory.

More Information

man etags and **man ctags** are the obvious starting places. You'll also find good information in the Emacs info pages, and using the Emacs **?H-a** command.

ctags and etags are both included in Emacs and XEmacs distributions. You can also get various other tags programs from the Internet—**archie -c ctags** will find a site near you.

Dave Thomas (davet@gte.net) is an independent consultant specializing in complex Unix, OS/2 and Windows developments. He's forever grateful for all the work that's gone into Linux and XFree—it lets him work from home in Dallas on client systems in Florida, New Hampshire, Atlanta, Toronto... The phone company is happy too.

Archive Index Issue Table of Contents

Advanced search

New Products

Margie Richardson

Issue #31, November 1996

Absoft Fortran 77 for Linux, Spyglass Client Web Technology Kit, and more.

Absoft Fortran 77 for Linux

Absoft Corporation now provides a commercial grade, native Fortran 77 compiler and debugger for Linux/Intel systems. The compiler includes Pentium optimizations and strong support for mainframe/workstation extensions. It is compatible with gcc and all system tools. Absoft offers educational and site discounts. For pricing contact them at the address below.

Contact: Absoft Corp., 2781 Bond Street, Rochester Hills, MI 48309, Phone: 810-853-0050, Fax: 810-853-0108, E-mail: sales@absoft.com, URL: http://www.absoft.com/.

Spyglass Client Web Technology Kit in Red Hat Linux

Red Hat Software, Inc. has announced that its Linux operating system will now include the Spyglass Client Web Technology Kit (WTK) at no additional cost. Spyglass WTK is the most comprehensive web technology offering available today, covering multiple platforms and offering a complete, high-quality set of premium web technologies that developers can “mix and match” to web-enable applications, services and devices.

Contact: Red Hat Software Inc., 3201 Yorktown Rd. Durham, NC 22713, Phone: 203-454-5500, Fax: 203-454-2582, E-mail: info@redhat.com, URL: http://www.redhat.com/.

BLAST Communications Software for Linux

BLAST, Inc., a provider of cross-platform asynchronous communications software, announced the release of BLAST for Linux (Intel-based), version 10.7. BLAST includes the following file transfer protocols: Xmodem, Ymodem,

Zmodem, Kermit and BLAST's own proprietary file transfer protocol. It also supports TTY and Passthru terminal emulations. BLAST for Linux runs on Linux version 1.2.12-ELF and is priced at $495.

Contact: BLAST Inc., P.O. Box 808, Pittsboro, North Carolina 27312, Phone: 800-242-5278, Fax: 919-542-0161, E-mail: sales@blast.com, URL: http://www.blast.com/.

TenXpert CD Server Upgrade

Ten X Technology, Inc. announced an upgrade to its TenXpert CD server that increases both performance and capability without raising the price. The TenXpert-1, entry-level server with a 1GB hard disk cache, now supports 42 CDs, up from 14. The midrange server, TenXpert-4 expands its hard disk cache from 1 to 2 GB, increases CD support from 42 to 168 discs, and adds support for writing CDs over the network to a CD-Recordable drive. The top end server, TenXpert-8 increases its hard disk cache from 2 to 4 GB, handles 250 CDs and supports NSM jukeboxes with its internal CD-Recordable drive. TenXpert servers have a list price beginning at $2995.

Contact: Ten X Technology, Inc., 13091 Pond Springs Road, Austin, Texas, 79729, Phone: 800-922-9050, Fax: 512-918-9495, E-mail: greg@tenx.com, URL: http://www.tenx.com/.

TransactNet Web Interface Toolkit

TransactNet, Inc. announced the availability of their Web Interface Toolkit (WIT), the first Java tool to automate the web. WIT was developed using a Linux system, and contains a set of Java class libraries, code generators and an intuitive user interface, allowing it to generate Java applets, stand-alone applications or “servlets”. The WIT class libraries, encapsulating HTML parsing, a Javascript compatible HTML document object model, and HTTP and CGI access can also be used directly by the developer. A free beta version of WIT 1.0 is available now and can be downloaded from http://www.transactnet.com/. Pricing for the production release scheduled in late September has not been announced.

Contact: TransactNet, Inc., 4094 Majestic Lane, Suite 226, Fairfax, VA 22033, Phone: 703-426-0386, Fax: 703-426-0387, E-mail: info@transactnet.com, URL: http://www.transactnet.com/.

32 bit ODBC Driver for c-tree Plus

The FairCom Corporation announced the release of its 32 bit Open Database Connectivity (ODBC) Driver Kit for c-tree Plus and the FairCom Servers. The

ODBC Driver v1.6A is a single-tier driver that interfaces directly with the c-tree Plus programming interface. This new release operates with the Linux X-Windows compatibility software and offers full read and write capabilities, support for multiple communication protocols and additional data alignment options. The ODBC Driver is a dynamic link library which supports both multi-user non-server and client/server modes of operation, and can be purchased for $59.00 per node.

Contact: FairCom Corporation, 4006 W. Broadway, Columbia, MO 65203, Phone: 573-445-6833, Fax: 573-445-9698, E-mail: faircom@faircom.com, URL: http://www.faircom.com/.

Phonetics Data Remote

Phonetics, Inc. announced the addition of Data Remote to its family of intelligent communication devices. Data Remote produces and communicates reports from remote equipment, and can be attached to any device with an RS232 output port to give instant reporting capability over standard phone lines. Data Remote is priced under $1000.

Contact: Phonetics, Inc., 901 Tryens Road, Aston, PA 19014, Phone: 610-558-2700, Fax: 610-558-0222, E-mail: sales@phonetics-monitoring.com, URL: http:/www.phonetics-monitoring.com/.

Archive Index Issue Table of Contents

Advanced search

Best of Technical Support

Various

Issue #31, November 1996

Our experts answer your technical questions.

Bootable Kernels and Slackware Installation

I have a new scsi controller (aha2940) which my only hard drive is connected to. I can get the latest boot disk for my controller, but the kernel that gets installed by Slackware is unbootable. How do I use the kernel on my boot disk as a kernel for my hard drive? How can I boot from the floppy and compile a kernel for my hard drive?

Manni Wood

A Work-around

During the Slackware installation procedure, a kernel is installed from the distribution set instead of from the boot disk that you used to start your i system. Handling the installation this way has the unfortunate side effect of making certain hardware devices unrecognizable to your new system since only two kernels are included in the distribution set—one for IDE and another for SCSI-based systems. The installation is handled this way because the boot disk kernels are “all-in-one” packages that have device drivers for every imaginable piece of hardware. This is quite inefficient for normal use, since many of the drivers are unused and these extra options will waste memory. After you install Linux you should compile and install a new kernel image with only the options you really need.

While you are setting up your system, you can use a temporary work-around that will let you use the boot disk's kernel to boot your system. Slackware boot disks prompt you for a set of options to pass to the booted kernel. One of the options will allow you to boot a system that has no working kernel image installed. At the prompt, type **mount root=/dev/X** , where X specifies the drive and partition where you installed Linux, e.g. sda, hdb2. This boot disk can be

quite handy to have in case you forget to re-run LILO after installing a new kernel, because your system will be unbootable without it. When it is used in conjunction with a root diskette that also contains some diagnosis and recovery tools, you will have a powerful pair of emergency utility disks.

Once Linux is running you need to copy a working boot kernel to wherever LILO (assuming you installed LILO) is looking for your current image. As a general practice you'll want to keep a backup copy installed as well. You can control LILO by editing **/etc/lilo.conf**. The default file should be fairly well commented and you can consult the LILO documentation for more details. If you don't have the original file you can copy the boot diskette onto your drive as a kernel image with the command:

```
dd if=/dev/fd0 of=/tmp/myimage bs=8192
```

*replacing **if** and **of** with the appropriate input and output locations. After the new kernel file is in place, rerun LILO by typing **lilo** so it can rebuild its boot tables. If you forget to take this step you will not be able to boot your system! To recompile and install a new kernel, obtain a kernel and extract the archive into **/usr/src/linux**. Users with Slackware distributions set up for kernel version 1.2 need to beware. Many things have changed as Linux has grown to version 2.0, so many things can break. You may wish to make this step later. Slackware 3.0 comes with the complete set of the newest version 1.2 kernel package, in the K disk set. Either install that or unpack your desired package into **/usr/src/linux**.*

The easiest and safest (though not the nicest looking) way to rebuild the kernel is to then **cd** into **/usr/src/linux**, type **make config,** and answer all the questions. Then type **make dep; make clean; make zImage**. If you are running on an Intel platform your new kernel image will be produced in **/usr/src/linux/arch/i386/boot/zImage** . Be prepared to wait, especially if you have a slow machine. If you are using a newer kernel package, you might type **make menuconfig** or one of the other combinations (see Makefile for details) for a better-looking configuration process.

—Chad RobinsonBRT Technical Services Corporation chadr@brttech.com

Drivers for 8 or 16 Port Serial Cards

Do you know where I could find a driver for a Jaws (extinct?) JCom-8 eight port serial card? What other 8 or 16 port cards would allow me to operate 8 Wyse 150 terms from Linux? —Gary Richardson

Here's One Source

That's not a card I've ever even heard of. To answer the second part of the question, there are several cards out there that can do what you need. The kernel has direct support for all of the Cyclades boards. We use a 16 port PCI Cyclades at Red Hat and it worked right out of the box (though it requires a kernel recompile or a module to be built).

—Donnie Barnes, Red Hat Software djb@redhat.com

Formatting Back-up Tapes

Is there any program/utility that will format, read the contents of and selectively back up or restore information on tapes?

—Dave Blondell

A Utility for Backing up and Restoring

There's a powerful utility called

Taper

that's able to selectively back up and restore information on tapes, with or without verifying. It's very easy to use as you can just tag the files or directories from menus. [See “Tar and Taper for Linux”, by Yusuf Nagree, in *LJ #22—Ed] Unfortunately, it cannot format tapes, so they must be bought preformatted (or formatted under DOS). I'm not aware of any utility that lets you format tapes under Linux.*

—Flavio Villanustre flavio@newage.com.ar

XF86Cig File under X-Windows

I am now trying to set up X-Windows, but I have no idea how the sections for “Device” and “Screen” of XF86Config file should be described. If you have any concrete example for my card, will you kindly let me know? My video card is: Canopus Power Window 968PCI-4M (S3)

—Hiroshi Shibata

Try This Instead

Rather than hacking the XF86Config by hand, have you tried using xf86config? (It should be in **/usr/X11/bin**.) The copy of xf86config that I have here (from 3.1.2D) lists an S3-968 (generic) option and that should work for you. The

xf86config that comes with the latest XFree86 might even list your card specifically. If it doesn't work, you might want to try using some of the options at http://www.xfree86.org/3.1.2/S3-1.html for other cards using the same chipset.

—Steven Pritchard,President, Southern Illinois Linux Users Group

Setting up Usenet News

How do I set up Usenet news (with CNews or INN)? What documentation/books are available?

—Koen Rousseau kobalt@innet.be

First, Obtain a News Feed . . .

First, if you want to carry the "real" Usenet, you must obtain a news feed from somewhere. Your ISP should be able to point you in the right direction, or sell you one themselves. Note that you don't necessarily need a news feed to use INN or CNews. If you only want to support some some local news groups, within a company intranet for example, then you don't need an outside feed. Looked at in that light, a Linux PC and INN can provide one of the most-touted features of a product like Lotus Notes (group conferencing and company-wide discussion forums) at a fraction of the cost. Once you've made arrangements for a feed, then you need to install the software. I recommend INN for new sites. You have a Red Hat distribution, and Red Hat has an RPM (Red Hat Packaging System) for INN on their ftp site under ftp://ftp.redhat.com/pub/contrib/RPMS/inn-1.4unoff4-2.i386.rpm. Download, then install with:

```
rpm -i inn-1.4unoff4-2.i386.rpm
```

It works right "out of the box". You will, of course, have to configure it for your site—add your feed site to ***/etc/news/newsfeeds, nntpsend.ctl*** *and* ***hosts.nntp****. Verify that the groups you want to carry are in* ***/var/lib/news/active*** *and newsgroups, and then configure* ***nnrp.access*** *to allow reading/posting from the proper IP addresses.*

After that is all working add **/usr/lib/news/bin/news.daily** and nntpsend to **/etc/crontab**. News.daily and nntpsend should be run as user "news", not as root. These programs expire old news and transmit your site's outgoing posts, respectively.

The RPM installs a FAQ under /usr/doc that should answer most of your questions.

—Bob Hauck, Wasatch Communications Group bobh@wasatch.com

Archive Index Issue Table of Contents

Advanced search

Consultants Directory

> This is a collection of all the consultant listings printed in *LJ* 1996. For listings which changed during that period, we used the version most recently printed. The contact information is left as it was printed, and may be out of date.

ACAY Network Computing Pty Ltd
Australian-based consulting firm specializing in: Turnkey Internet solutions, firewall configuration and administration, Internet connectivity, installation and support for CISCO routers and Linux.

Address:
Suite 4/77 Albert Avenue, Chatswood, NSW, 2067, Australia
+61-2-411-7340, FAX: +61-2-411-7325
sales@acay.com.au
http://www.acay.com.au

Aegis Information Systems, Inc.
Specializing in: System Integration, Installation, Administration, Programming, and Networking on multiple Operating System platforms.

Address:
PO Box 730, Hicksville, New York 11802-0730
800-AEGIS-00, FAX: 800-AIS-1216
info@aegisinfosys.com
http://www.aegisinfosys.com/

American Group Workflow Automation
Certified Microsoft Professional, LanServer, Netware and UnixWare Engineer on staff. Caldera Business Partner, firewalls, pre-configured systems, world-wide travel and/or consulting. MS-Windows with Linux.

Address:
West Coast: PO Box 77551, Seattle, WA 98177-0551
206-363-0459
East Coast: 3422 Old Capitol Trail, Suite 1068, Wilmington, DE 19808-6192
302-996-3204
amergrp@amer-grp.com
http://www.amer-grp.com

Bitbybit Information Systems
Development, consulting, installation, scheduling systems, database interoperability.

Address:
Radex Complex, Kluyverweg 2A, 2629 HT Delft, The Netherlands
+31-(0)-15-2682569, FAX: +31-(0)-15-2682530
info@bitbybit-is.nl

Celestial Systems Design
General Unix consulting, Internet connectivity, Linux, and Caldera Network Desktop sales, installation and support.

Address:
60 Pine Ave W #407, Montréal, Quebec, Canada H2W 1R2
514-282-1218, FAX 514-282-1218
cdsi@consultan.com

CIBER*NET
General Unix/Linux consulting, network connectivity, support, porting and web development.

Address:
Derqui 47, 5501 Godoy Cruz, Mendoza, Argentina
22-2492
afernand@planet.losandes.com.ar

Cosmos Engineering
Linux consulting, installation and system administration. Internet connectivity and WWW programming. Netware and Windows NT integration.

Address:
213-930-2540, FAX: 213-930-1393
76244.2406@compuserv.com

Ian T. Zimmerman
Linux consulting.

Address:
PO Box 13445, Berkeley, CA 94712
510-528-0800-x19
itz@rahul.net

InfoMagic, Inc.
Technical Support; Installation & Setup; Network Configuration; Remote System Administration; Internet Connectivity.

Address:
PO Box 30370, Flagstaff, AZ 86003-0370

602-526-9852, FAX: 602-526-9573
support@infomagic.com

Insync Design
Software engineering in C/C++, project management, scientific programming, virtual teamwork.

Address:
10131 S East Torch Lake Dr, Alden MI 49612
616-331-6688, FAX: 616-331-6608
insync@ix.netcom.com

Internet Systems and Services, Inc.
Linux/Unix large system integration & design, TCP/IP network management, global routing & Internet information services.

Address:
Washington, DC-NY area,
703-222-4243
bass@silkroad.com
http://www.silkroad.com/

Kimbrell Consulting
Product/Project Manager specializing in Unix/Linux/SunOS/Solaris/AIX/ HPUX installation, management, porting/software development including: graphics adaptor device drivers, web server configuration, web page development.

Address:
321 Regatta Ct, Austin, TX 78734
kimbrell@bga.com

Linux Consulting / Lu & Lu
Linux installation, administration, programming, and networking with IBM RS/6000, HP-UX, SunOS, and Linux.

Address:
Houston, TX and Baltimore, MD
713-466-3696, FAX: 713-466-3654
fanlu@informix.com
plu@condor.cs.jhu.edu

Linux Consulting / Scott Barker
Linux installation, system administration, network administration, internet connectivity and technical support.

Address:
Calgary, AB, Canada
403-285-0696, 403-285-1399
sbarker@galileo.cuug.ab.ca

LOD Communications, Inc
Linux, SunOS, Solaris technical support/troubleshooting. System installation, configuration. Internet consulting: installation, configuration for networking hardware/software. WWW server, virtual domain configuration. Unix Security consulting.

Address:
1095 Ocala Road, Tallahassee, FL 32304
800-446-7420
support@lod.com
http://www.lod.com/

Media Consultores
Linux Intranet and Internet solutions, including Web page design and database integration.

Address:
Rua Jose Regio 176-Mindelo, 4480 Cila do Conde, Portugal
351-52-671-591, FAX: 351-52-672-431
http://www.clubenet.com/media/index.html/

Perlin & Associates
General Unix consulting, Internet connectivity, Linux installation, support, porting.

Address:
1902 N 44th St, Seattle, WA 98103
206-634-0186
davep@nanosoft.com

R.J. Matter & Associates
Barcode printing solutions for Linux/UNIX. Royalty-free C source code and binaries for Epson and HP Series II compatible printers.

Address:
PO Box 9042, Highland, IN 46322-9042
219-845-5247
71021.2654@compuserve.com

RTX Services/William Wallace
Tcl/Tk GUI development, real-time, C/C++ software development.

Address:
101 Longmeadow Dr, Coppell, TX 75109
214-462-7237
rtxserv@metronet.com
http://www.metronet.com/~rtserv/

Spano Net Solutions
Network solutions including configuration, WWW, security, remote

system administration, upkeep, planning and general Unix consulting. Reasonable rates, high quality customer service. Free estimates.

Address:
846 E Walnut #268, Grapevine, TX 76051
817-421-4649
jeff@dfw.net

Systems Enhancements Consulting
Free technical support on most Operating Systems; Linux installation; system administration, network administration, remote system administration, internet connectivity, web server configuration and integration solutions.

Address:
PO Box 298, 3128 Walton Blvd, Rochester Hills, MI 48309
810-373-7518, FAX: 818-617-9818
mlhendri@oakland.edu

tummy.com, ltd.
Linux consulting and software development.

Address:
Suite 807, 300 South 16th Street, Omaha NE 68102
402-344-4426, FAX: 402-341-7119
xvscan@tummy.com
http://www.tummy.com/

VirtuMall, Inc.
Full-service interactive and WWW Programming, Consulting, and Development firm. Develops high-end CGI Scripting, Graphic Design, and Interactive features for WWW sites of all needs.

Address:
930 Massachusetts Ave, Cambridge, MA 02139
800-862-5596, 617-497-8006, FAX: 617-492-0486
comments@virtumall.com

William F. Rousseau
Unix/Linux and TCP/IP network consulting, C/C++ programming, web pages, and CGI scripts.

Address:
San Francisco Bay Area
510-455-8008, FAX: 510-455-8008
rousseau@aimnet.com

Zei Software
Experienced senior project managers. Linux/Unix/Critical business software development; C, C++, Motif, Sybase, Internet connectivity.

Address:
2713 Route 23, Newfoundland, NJ 07435
201-208-8800, FAX: 201-208-1888
art@zei.com

Archive Index Issue Table of Contents

Advanced search