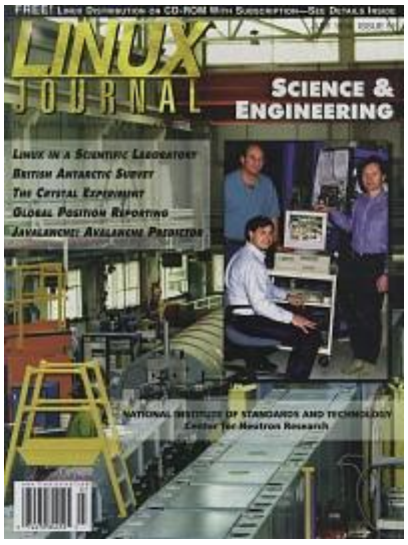
Advanced search

Linux Journal Issue #51/July 1998



Features

Archive Index

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

The Crystal Experiment

Emanuele Leonardi

Giovanni Organtini

Issue #51, July 1998

Linux is now being used in high-energy nuclear studies in Geneva by CERN.

Imagine this: in a 27 kilometer long circular pipe running along a tunnel drilled over 100 meters underground, two beams of a few billion protons accelerate to an energy in excess of 14,000 times their own mass and collide head-on, generating a small *big bang* where hundreds and hundreds of newly-created particles are violently projected in all directions. Searching these, thousands of physicists from all around the world will try to find a few new particles whose existence, according to modern theories, would give new insights into the deepest symmetries of the universe and possibly explain the origin of mass itself.

This almost science fiction scenario is more or less what will happen near Geneva, Switzerland, at CERN (see Resources [1]), the European Center for Nuclear Research, when the Large Hadron Collider (LHC) starts its operations in the year 2005. The instruments the scientists will use to observe these very high-energy interactions are two huge and extremely complex particle detectors, code-named ATLAS and CMS, each weighing over 10,000 tons, positioned around the point where the protons will collide.

Figure 1. This photo shows one of the about 100,000 lead tungstate scintillating crystals which will be used in the electromagnetic calorimeter of the CMS experiment.

Our experimental physics group is now involved in a multi-disciplinary R&D project (see Resources [2]) related to the construction of one of the two detectors, CMS (Compact Muon Solenoid). In particular, we are studying the characteristics of a new crystal, the lead tungstate or PWO, which, when hit by a particle, emits visible light. About 100,000 small PWO bars (Figure 1) will compose the part of the CMS detector called the “electromagnetic calorimeter”, which will measure the energy of all the electrons and photons created in the collisions.

<u>**Figure 2. The dark chamber of our experimental bench: crystals to be measured are inserted here. The rail on the top moves a small radioactive source along the crystal (here wrapped in aluminum foil) and the produced light is collected by the phototube on the left.**</u>

In our laboratory, located in the Physics Department of the University “La Sapienza” in Rome, Italy, we spent the past two years setting up a full experimental bench to measure all the interesting properties of this crystal. The PWO crystals are inserted into a dark chamber (Figure 2) and a small radioactive source is used to excite them so that we can measure the small quantities of light produced. Instruments used on the bench include light detectors, temperature probes, analog-to-digital converters (ADC), high-voltage power supplies, and step motors (Figure 3). To interconnect and control most of these instruments and to allow a digital readout of the data, we used the somewhat old (but perfectly apt to our needs) CAMAC standard.

<u>**Figure 3. The electric signal coming from the phototube is fed into a CAMAC-based DAQ chain which amplifies and digitizes it before sending it to our computer. The photo shows all the instruments involved in the operation.**</u>

One of the problems we had to face when the project began at the end of 1995 was how to connect the data acquisition (DAQ) chain to a computer system for data collection without exceeding the limited resources of our budget. A possibility was the use of an old ISA-bus-based CAMAC controller board available from past experiments. This was a CAEN A151 board released in 1990, a low-level device which nonetheless guaranteed the speed we needed. We then bought an off-the-shelf 100 MHz Pentium PC to handle all the communications. The problem was how to use it. CAEN only provided a very old MS-DOS software driver which, of course, hardly suited our needs as the mono-user, mono-task operating system could not easily fit into our UNIX-based environment.

Enter Linux

One of us (E.L.) was using Linux at the time on his PC at home where he could appreciate Linux's stability and the possibilities offered by the complete availability of the source code. The idea of using such a system in our lab presented several appealing features. First, using Linux would give us a very reliable and efficient operating system. The CPU time fraction spent in user programs is quite large with respect to the time used by the kernel, and there is complete control of priorities and resource sharing among processes. This feature is of great importance when the time performances of the DAQ program are strict (but not so strict to require a true real-time system): data acquisition can be given maximum priority over any other task that may be running on the same computer, such as monitor programs or user shells.

Moreover, we had access to a large UNIX cluster composed of HP workstations which we could use for data analysis. Using Linux, with all the facilities typical of a UNIX OS and the GNU compilers, the data acquisition system could be smoothly integrated with this cluster. Porting of scripts and programs would be straightforward and the use of the TCP/IP-based tools (NFS, FTP) would permit an automatic data transfer among the systems. Also, the use of X-based graphical interfaces would permit remote monitoring of ongoing DAQ sessions, not only from our offices, located a few hundred meters from the lab, but also from remote locations such as CERN.

The multi-user environment would allow a personalized access to data and programs, e.g., granting some users the permissions to start a DAQ session but not to modify the software or allow user interference in ongoing DAQ sessions.

Last but not least, the entire system would be completely free under the GNU license, including compilers, development tools, GUIs and all the public domain goodies that come with Linux.

All these advantages were quite clear in our minds but exploiting Linux was still dependent on being able to use our old CAMAC controller board. It is here that Linux proved all of its great potential as the operating system of choice in our lab.

The CAMAC Device Driver

Our CAMAC controller consisted of a board which, when inserted on the ISA bus of a computer, could connect to as many as seven different CAMAC crates, each containing up to 22 different specialized devices connected to measuring instruments.

This board was mapped on a known set of ISA bus memory addresses through which a user could send commands to each individual instrument and retrieve the responses. UNIX permits access to physical memory addresses only at the kernel level: it was clear that we needed a set of software routines dedicated to the interaction with the DAQ board, usually called a device driver, to insert into the Linux kernel.

Though aware of the existence of kernel-level device drivers, none of us knew exactly how to write one. (All this happened a few months before the appearance of the good article by Alessandro Rubini. See Resources [3].) We then decided to ask for help on the comp.os.linux.hardware newsgroup, and less than 24 hours later we were contacted by Ole Streicher, a German researcher who sent us the source code of a device driver he wrote for a different CAMAC controller (see Resources [4]). Adapting it to our board was a matter of a couple of days, and then experimental data were happily flowing in and out of our DAQ system: the Linux option was finally open.

The ability to dynamically load and unload modules in the kernel, a feature which had been introduced in Linux only a few months before, was of great help in the driver development phase.

DAQ Control and Monitoring Programs

From the user's point of view, the CAMAC system was now visible via simple files, one for each different crate, which could be opened, closed, written to and read from. We also provided an interface library in order to hide the low-level details of the CAMAC operations and facilitate code writing. The presence of both the **gcc** C compiler and the **f2c** FORTRAN-to-C converter allowed us to provide both a C and a FORTRAN version of this interface library, in order to allow our colleagues to write their own DAQ programs.

Using this library we wrote the main DAQ program which was able to automatically set the run conditions, control the movement of the radioactive source via a serial link to a step motor, send light pulses to calibrate the light sensors, and collect the data coming from the DAQ system and analyze them on the fly. To write the user interface, we used the Tcl/Tk package (see Resources [5]): all the program controls appeared on a graphical window which could be opened on any X display (Figure 4).

Figure 4. This is a snapshot of our PC screen during an actual DAQ run. In the center you can see the Tcl/Tk-based control interface while the window on the left shows the data collected during the run. The histogram, updated in real-time during the run, is created using the HBOOK and HPLOT packages of the CERN libraries.

Parallel to the DAQ program, we wrote a program to monitor the status of the data acquisition and of some important parameters such as the number of events collected, event rate and average values. With a scientific libraries package called CERNLIB (see Resources [6]) developed at CERN, freely available along with its source code and widely used in the high-energy physics community, we interfaced the monitor program to a simple analysis facility. This allowed us to access interesting information and execute some preliminary analysis even while the DAQ was being done (Figure 4).

Figure 5. The authors of this article in their natural environment. On the left, with an arm on the DAQ PC, is Emanuele Leonardi and on the right is Giovanni Organtini.

Performance

An important factor for a DAQ system is the time performance. If the controlling software is too slow, data may be lost and the time required to collect a useful amount of data can grow to an unacceptable level.

We found the only time-limiting factor in our system was the conversion time of the ADC board; the operating system could easily keep pace with the DAQ task, even while running several other user tasks. This is very important, as this year our bench will move from a prototype level with a single active DAQ chain to an industrial-strength production facility where multiple measurements will proceed in parallel in order to quickly handle all of the many thousands of crystals needed for the CMS experiment.

In practice, we measured the time to execute a single CAMAC operation to be on the order of 10 microseconds, large with respect to the 1.5 microseconds minimum CAMAC operation time, but very good for an inexpensive board such as the CAEN A151 and much lower than the ADC response time of 110 microseconds.

Conclusion

Thanks to the introduction of Linux in our lab, we were able to realize a complete data acquisition and monitoring system using an off-the-shelf Pentium PC and a low-cost CAMAC board.

The system has been performing flawlessly since the beginning of 1996, and the data collected have been used to study the properties of PWO crystals, which will be used in the CMS experiment at CERN.

The key points in using Linux were the availability of the kernel code and the enthusiasm and technical knowledge of the Linux community which enabled us to create a personalized device driver for our data acquisition system. The standard UNIX tools and the GNU compilers guaranteed a perfect integration with the existing machines and an immediate acceptance of the system by all the physicists in our group.

As soon as we started to show our work, we were invited to several congresses dedicated to computing for high-energy physics and data acquisition systems all around Europe (the PCaPAC'96 Workshop in Hamburg, Germany, the ESONE'97 Workshop at CERN and the CHEP'97 Conference in Berlin, Germany).

Everywhere we got in touch with many other Linux enthusiasts working on related items; the interest of the high-energy physics community in Linux is very high indeed.

We now plan to use this same system for a larger automatic bench which will be used in the next six years to measure the properties of the tens of thousands of crystals which will be used to build the electromagnetic calorimeter of the CMS experiment.

For those interested in our work, an archive containing the latest version of the device driver code and the interface libraries can be found on our FTP site at ftp://ftpl3.roma1.infn.it/pub/linux/local/.

Resources

Emanuele Leonardi got his Ph.D. in physics in 1997 at the University “La Sapienza” in Rome. He is now working as a technology researcher for the National Institute of Nuclear Physics in Rome.

Both authors worked in the L3 experiment at CERN where they published several physics papers and are now collaborating on the CMS experiment R&D phase.

Giovanni Organtini got his Ph.D. in physics in 1995 at the University “La Sapienza” in Rome. He is now a physics researcher at the University RomaTRE in Rome.

Both authors worked in the L3 experiment at CERN where they published several physics papers and are now collaborating on the CMS experiment R&D phase.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Due South with the British Antarctic Survey

Craig Donlon

James Crawshaw

Issue #51, July 1998

Linux now facilitates scientific research in the Atlantic Ocean and Antarctica.

A cold and windy September afternoon marks the start of the fifth Atlantic Meridional Transect (AMT) experiment aboard the British Antarctic Survey (BAS) research vessel RRS *James Clark Ross*. Each year the ship sails from the UK to the Falkland Islands in September en route to service the UK Antarctic research bases. Jim (our UNIX support engineer) is busy fastening down his trusty Toshiba laptop (Tecra 730XCDT with 48MB) in his cabin on board *James Clark Ross* in preparation for the inevitable bad weather. Ahead of us lie six weeks of precision ocean-atmosphere measurements, near real-time data processing, heated debate, troubleshooting and, hopefully, some scientific discovery. Fortunately, we have both chosen one of the most versatile and reliable operating systems at hand to maximize our output during this experiment—*Linux*. This article discusses the impact of Linux, which is now routinely used at BAS and during the AMT experiments. (See http://www1.npm.ac.uk/amt/ for more information on the AMT project.)

The Atlantic Meridional Transect Experiment

Figure 1. The British Antarctic Survey research vessel RRS *James Clark Ross* used for the AMT experiments (Photo: Tony Bale)

The RRS *James Clark Ross* (shown in Figure 1) was launched in 1990 and is one of the world's most complex marine research vessels, incorporating over 400 square meters of scientific laboratory space. It was named after the scientist and polar explorer Admiral Sir James Clark Ross, RN (1800-1862), who in February 1842 reached 78.9 degrees S—a record which remarkably stood well into the 20th century.

Figure 2 shows the track taken by the *James Clark Ross* during the and AMT-5 cruise superimposed on a monthly composite satellite image of Chlorophyll *a* derived from the Sea-viewing Wide Field-of-view Sensor (SeaWiFS) carried on the SeaStar spacecraft. Different types of phytoplankton have characteristically different concentrations of chlorophyll and are, therefore, different in color. By measuring the color of the ocean with the SeaWiFS instrument, estimates of the amount and general type of phytoplankton in specific regions can be made as shown in Figure 2. An extensive web site provides a wealth of information on the SeaWiFS mission at http://seawifs.gsfc.nasa.gov/SEAWIFS.html.

Figure 2. Ground track of the RRS *James Clark Ross* during the AMT-5 experiment superimposed onto a SeaWiFS satellite composite image of Chlorophyll *a*. White marks regions of cloud and the arrow highlights the direction of the north equatorial current system which is pushing nutrient rich Amazon water to the north west. (Image courtesy of Stanford B. Hooker.)

The ocean-atmosphere measurements taken during the AMT cruises are fundamental for the calibration, validation and interpretation of remotely sensed observations, including sea-surface temperature, wind speed, atmospheric water vapour and ocean colour (which can be related to bio-optical processes)—all of which are vital for ongoing climate research. Figure 3 shows the instrument cluster mounted on the forward mast of the *James Clark Ross* which measures, among other things, sea-surface temperature (using an infrared radiometer), solar radiation, wind speed and direction, air temperature, radar backscatter (a measure of surface roughness) and humidity. All of these measurements are used to investigate the processes, occurring at the air-sea interface, which in many cases define the signal actually measured by the satellite instruments. (See http://www1.npm.ac.uk/amt/ for more information on the AMT project.)

Figure 3. Forward mast installation of the RRS *James Clark Ross* showing some of the radiometers and the radar system.

Why Choose Linux?

From a system engineer's point of view, the main requirement of an operating system is that it integrate with the existing computing infrastructure available on (in this case) BAS ships and also at Antarctic bases as well as at headquarters. Since the majority of instruments are logged to Sun SPARC workstations, it makes sense to run an OS which allows NFS mounts to the data areas the workstations create. Many users require the ability to perform data processing locally without having to place extra strain on the data-logging workstations. As most of this is undertaken using shell scripts or compiled C source code, it made sense to run a form of UNIX locally. This enables shell scripts to run with no modifications and the C code to easily recompile. Other user-driven considerations included access to backup hardware, hard-copy output, real-time data displays and access to a common library of software packages.

Figure 4. The main laboratory space occupied by the Linux workstations aboard the RRS *James Clark Ross* during the AMT experiments.

Upon considering these requirements, our choice was made simple: it had to be Red Hat Linux. For our purpose, Linux provides an extremely versatile operating system with the ability to effortlessly and seamlessly integrate itself efficiently into any existing network system. The immense amount of supported hardware made installation on Jim's laptop and on our desktop machines a painless exercise, and we were delighted with the way that even a default installation gave us exactly what we wanted. It is this kind of hardware support and user-friendly installation interface which caught our eye in the first place when considering various UNIX systems for Intel processors.

After unpacking our brand new PC (Intel DX 120; 32MB RAM) and removing the pre-installed OS (we did ask for Linux), the whole installation took less than an afternoon. We now had a fully working UNIX workstation configured within the *James Clark Ross* NIS domain, auto-mounting file systems whenever required. Jim went even further with his laptop. Using Caldera Wabi 2.0, he had the ability to run the BAS standard word processor and e-mail packages (which are MS Windows-based). Being able to do all this using free or inexpensive software proves what a professional product Linux is. Gone are the days when Linux was considered a “toy UNIX” for hackers—it is now a fully functional UNIX environment which is just as stable (if not more) as the various commercial UNIX systems on the market.

Software Support

Software support for Linux is already immense, and growing rapidly. Of particular importance was our need to use the RSI Inc. Interactive Data Language (IDL) to develop processing tools and visualize our data in near real time. IDL is a powerful data-visualization tool, and RSI Inc. recognized the power of Linux several years ago by choosing to support it. Using Linux IDL, a complete data-processing suite was developed for two new instruments deployed during the AMT which are still in active use today. We also had a need to effectively communicate and work on collective documents during our time at sea. Most users choose MS Word for this purpose, and using Caldera Wabi 2.0, we were easily able to supply this application. This, coupled with Linux's ability to mount Novell Netware volumes, meant that we truly had the best of both worlds: access to all our UNIX file systems plus the Netware volumes and associated applications. In fact, we found certain applications ran significantly more reliably under Wabi than they did under their native operating system.

For example, the Wabi interface allowed us to manipulate (independent of the ship's logging system) a Campbell Scientific Data Logger located on the forward mast via short-haul modem communications connected to the Linux desktop. Using Campbell's own data logger software under Windows 95, we found that significant drifts in the system date-time stamp of 10min/day were confusing the data logger, which is auto-adjusted to keep the data logger time in sync with the PC time. The only solution was a system restart every 6 hours or so in order for Windows 95 to grab a correct time on startup. We found that under Linux Wabi, these problems no longer existed.

Linux in Other Areas

The British Antarctic Survey has written custom software to allow its ships and bases to send and receive electronic mail with the rest of the world using standard Internet e-mail addresses. It was decided to write custom software so that mail could be compressed more efficiently than when using standard protocols, and this in turn reduces costs as it decreases the amount of expensive satellite air time required.

When considering all the requirements, it became clear that a system based around Sendmail running on a UNIX workstation was an almost ideal solution. This solution could be easily implemented on our ships and bases which already had Sun SPARC workstations. As for two smaller bases, it was decided to send PCs running Debian Linux. Again, Linux proved to be an inexpensive but professional solution to a problem. It would have been difficult to justify the expense of Sun SPARCs for the smaller bases, whereas it was relatively inexpensive to install Linux on a couple of older PCs which still performed well. The fact that BAS chose to use Linux to perform Antarctic communications with two of its bases shows its trust in the stability of Linux, as the communications systems are vital to the normal operation of bases.

Conclusions

Without Linux, the computing options for these types of operations are simple: either pay large amounts of money for proprietary systems, or suffer at the hands of a less versatile operating system. Linux changes all that. We are able to function at a professional level at a minimum cost with all of the connectivity, reliability, software choices and versatility that Linux offers. Support for Linux within the British Antarctic Survey and Colorado Center for Astrodynamics Research is increasing. Indeed, it is an officially supported operating system at both institutions and not a toy which the IT hackers play with.

Many more users are requesting Linux for reasons as diverse as wanting to run geophysical processing software on remote islands in the southern Atlantic to simply wanting to run their PC as an intelligent X terminal. Today Linux offers a truly cost-effective off-the-shelf solution for all of our requirements that rivals anything else available in the marketplace. Linux is now being recognized for what it is-a truly outstanding operating system that has grown immensely over the last few years thanks to dedicated individuals and groups working in cooperation with others.

Craig Donlon is a research fellow at the Colorado Center for Astrodynamics Research. In between moving from the mile-high community of Boulder, Craig can be found at the Applied Space Institute, European Joint Research Centre in Ispra, Italy, and hacking satellite and ship data on his Linux machine. He can be contacted at cjdn@colorado.edu.

James Crawshaw works for the British Antarctic Survey as a UNIX support engineer based at their headquarters in Cambridge, although two to three months a year are spent in Antarctica serving on board their research ships and bases providing general computing support. He can be contacted in Cambridge at james.crawshaw@bas.ac.uk.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Linux in a Scientific Laboratory

Przemek Klosowski

Nick Maliszewskyj

Bud Dickerson

Issue #51, July 1998

The authors tell us how they use Linux daily to fulfill the requirements of their lab.

Our laboratory, the NIST Center for Neutron Research (NCNR) at the National Institute of Standards and Technology, uses neutron beams to probe the structure and properties of materials. This technique is in many respects similar to its better-known relative, X-ray scattering, but offers some unique advantages for studies of materials as diverse as semiconductors, superconductors, polymers and concrete.

Our work could not be done without computer technology. Computers help us collect experimental data: they interface with the real world, controlling and recording various physical parameters such as temperature, flux and mechanical position. The collected measurements need to be displayed, analyzed and communicated to others. All these stages require sophisticated and flexible computer tools. In this article we will describe how Linux helps us solve many needs that arise in our everyday work. We believe that our experience might be typical of any scientific or engineering research and development laboratory.

The main advantage we get from using Linux is its amazing flexibility. Because of the open development model and open source code, there are no “black box” subsystems; when something doesn't work correctly, we can usually investigate the problem and fix it to our satisfaction. The significant spirit of cooperation and mutual support found in Linux is important to us—a consequence of the general philosophy of open software as well as the practical result of source code being available for anyone to fix. Also, Linux is rather robust, in the sense that once something is set up, it stays set up; Linux shows none of the brittleness that, unfortunately, we have learned to expect from mainstream computer operating systems.

Unfortunately, sometimes we run into a lack of support for some useful hardware or software. Since few manufacturers actively support Linux, the driver availability on Linux lags behind Windows 95, although it is probably better than any other environment thanks to the excellent work of many people who contribute their hardware drivers. We avoid unsupported hardware by checking the availability of drivers before purchasing, and by staying away from the manufacturers who do not publish engineering specifications for their products.

In the end, we use whichever environment does the job better. Since some tools are available on Windows and not on Linux, we sometimes use the former. For instance, the LabView software, available on Windows, is an integrated graphical tool for rapid prototyping of data acquisition, with an impressive collection of instrumentation hardware drivers. It is sometimes the platform of choice, especially for exploratory work, although it doesn't scale well for more complex tasks.

Overall, we have about 25 computers running Linux. We have been very happy with their operation and have saved taxpayers a bundle of money in the process. We have seen Linux grow from a virtual unknown, perceived as risky and devoid of support, to its current status as a serious contender with brand-name UNIX and NT boxes, and we definitely see Linux in our future.

Figure 1. NIST Center for Neutron Research

Interfacing to the Real World

Real-world data acquisition usually requires endlessly repeated high-precision measurements, and so it is ideally suited for a computer, as long as the data is available in computer-readable form. Unfortunately, data acquisition is not a mass-market application, so the acquisition hardware tends to be expensive and hard to obtain, even for the ubiquitous PC/x86 platform. Consider a sound card: it has high quality analog-to-digital and digital-to-analog converters, timers, wave-table memory, etc., all for around $100 US. Similar hardware with relatively small modifications to make it suitable for data acquisition will probably cost around $1000 US.

The scientific instruments we use at the NCNR are quite diverse and interesting on their own; a lot of mechanical and electronic engineering is involved even before computers get into the picture. Some of our instruments are quite impressive in size and weight—we actually use decommissioned battleship gun turret components to support them. You can get a feel for the scale of our instrumentation by looking at Figures 1 and 2; the experiment hall measures approximately 30 by 60 meters.

For the purpose of this article, let us assume all the hard work of designing and constructing an instrument has been done, including providing the appropriate sensors that measure the interesting physical quantities such as temperature, radiation intensity or position. Our task is to read data from these sensors into the computer. (Because of concerns for cost and availability of hardware, PC/x86 platform is the practical choice for data acquisition tasks.)

RS-232 (Serial) Ports

The easiest and quite common situation is for the instrument to have a built-in serial port. We can then talk to it using regular serial communication, just like talking to a modem. Examples of such instruments in our lab include stepper motor controllers, temperature controllers and various precise time measurement apparatus.

The simplicity of a serial-line (RS-232) interface has a cost: a serial connection is rather slow and unsuitable for situations requiring quick response or large amounts of data. It also has the annoying feature of being a very loosely defined standard. There are many variations: DTE vs. DCE configuration; hardware vs. software handshake; various settings of data, stop and parity bits. With so many possibilities, the probability that two randomly selected devices will talk to each other right after plugging in the cable is vanishingly small. The ubiquitous “break-out box” is helpful here: it is a small enclosure connected in series with our serial cable, showing the status of data lines and allowing us to reroute individual signal lines with jumper wires.

Compared to the difficulty of figuring out the proper cabling and communication parameters, the actual programming of serial-port communications is trivial, since Linux already provides good quality serial-port drivers. (Linux, of course, does not rely on serial-port routines in PC BIOS and MS-DOS, since they are so inadequate that a whole cottage industry was created providing so-called “communication libraries”.)

One problem with serial-port communications: RS-232 is inherently a point-to-point link—there is no standard and reliable way of connecting multiple devices to the same serial line. There is a scheme where several devices are daisy-chained, i.e., the computer's transmit line goes to the receive input of device 1, its transmit goes to device 2's receive, and so on, until the transmit line of the last device returns to the computer's receive pin. This requires that all devices cooperate by passing on data not destined for them; it is also not reliable when there can be asynchronous responses from devices in the chain. One of the two standard serial ports usually provided on a PC platform is occupied by the mouse, so we need a multi-port expansion board if we need more than one serial line. Fortunately, Linux has built-in support for several inexpensive multi-port boards. We have used Cyclades and STB boards; they are very easy to configure, and their drivers present themselves to the programmer as a regular serial port.

For initial exploration and testing, we normally use either the Seyon or Kermit terminal emulators. Seyon comes with most Linux distributions, while Kermit has to be obtained from Columbia University's FTP site, as its license terms prohibit third-party distribution. The minicom program is harder to configure, so we do not use it much.

Nick wrote a Tool Command Language (Tcl) serial communication extension for flexible serial-port I/O, with timeouts and terminator characters. Tcl fits well within our environment because it can be conveniently embedded as a scripting tool for heavy-duty FORTRAN and C programs, and it allows for rapid development, while being robust enough to be deployed in a production environment. We will discuss the benefits of scripting in our environment later in the article.

GPIB Bus

Another hardware interface popular in scientific and engineering communities is the GPIB—General Purpose Interface Bus. It was designed and popularized by Hewlett Packard (hence its original name HP-IB), and later became an official industry standard, IEEE-488. It is a medium-speed parallel bus, capable of over 100Kbps bandwidth. Many scientific instruments support it, and there are relatively inexpensive controllers for PCs, made by Hewlett Packard, National Instruments and others. Fortunately, Linux kernel drivers, written by Claus Schroeter, already exist for most common GPIB cards. (See “GPIB: Cool, It Works With Linux” by Timotej Ecimovic, *Linux Journal*, March 1997.)

Versabus Module Europe (VME)

For those applications requiring very fast data transfer, the VME bus is a common choice. VME is popular in the telecommunications industry, as well as for industrial and military test and measurement applications. It is typically housed in large (and expensive) backplane crates, containing 24 card slots. Usually one of these slots is occupied by a controller that controls the I/O modules in the remaining slots. Originally, VME was designed to work with Motorola 68k-series CPUs, and so most crate controllers were 68k-based, but recently PowerPC and even Pentium-based controllers seem to be more popular.

It turns out that there are Linux ports to all of these architectures, but again, it was simplest for us to use an x86-based VME controller. In most respects, it is a standard Pentium/PCI miniature motherboard, with the only unusual feature being an on-board PCI-VME bridge chip. We use a controller made by VMIC with a VIC bridge chip set; Nick wrote a driver for it, based on another VME-bridge driver we found on the Net.

All VME I/O is done via memory mapping. The I/O modules are accessed by reading and writing their specific memory locations; the VME bridge chip is needed to translate CPU native bus cycles onto the VME bus. A program simply needs to map the appropriate memory area (using **mmap**), and it can then execute regular memory load and store operations to access the VME peripherals.

We are currently completing a large data-acquisition system that collects precise timing information from events observed at over 800 detectors. We have designed a front-end processor on a VME card module that handles 32 detectors, and another module which multiplexes data from these front-end modules into the crate controller. As the maximum possible data rate in this application is 300,000 events per second, VME is an appropriate platform.

Programmable Logic Controllers (PLC)

PLC are widely used in the industrial process control environments. (See “Using Linux with Programmable Logic Controllers” by J. P. G. Quintana in *Linux Journal*, January 1997.) They are descendants of relay-based control systems and are built out of modular I/O blocks governed by a dedicated microcontroller. The available modules include digital and analog I/O, temperature control, motion control, etc. They trade off simplicity and low speed for low cost and industrial grade reliability; they are also very nicely integrated mechanically—individual modules pop into a mounting rail and can be installed and removed easily. There are several PLC systems on the market; we currently use the KOYO products.

Typically, a simple control program is developed in a proprietary cross-compiling environment (usually in the form of a relay “ladder diagram”, a method that dates back to days of electromechanical relays), and downloaded via a serial link. Typically such programs run under Windows, but they need be run only for development. After storing the control program in the flash memory, the microcontroller communicates with our Linux boxes, sending data and receiving commands via a built-in serial link.

Figure 2. One of our experimental stations, with the instrument control computer on the left, and two VME crates plus a PLC unit. Linux runs on the PC and in the controller of the lower VME crate.

Other Interfaces

The parallel port provides another popular computer interface. As Alessandro Rubini explained in “Networking with the Printer Port” (*Linux Journal*, March 1997), the parallel port is basically a digital I/O port, with 12 output bits and 4 input bits (actually, the recent enhanced parallel port implementations allow 12 input bits as well). To a dedicated hobbyist that is a precious resource, which can drive all kinds of devices (D/A and A/D converters, frequency synthesizers, etc); unfortunately, there is usually only one such port in a computer, and it tends to be inconveniently occupied by the printer. The serial port can also be used in a non-standard way; its status lines may be independently controlled and therefore provide several bits of digital I/O.

Such digital I/O can be used to “bit-bang” information to serial bus devices such as I2C microcontrollers. (I2C is a two-wire serial protocol used by some embedded microcontrollers, sometimes even available in consumer products.)

USB is another type of interface, appearing in terms of both available hardware and Linux support. Although designed for peripherals such as keyboards, mice or speakers, it is fast enough to be useful for some data-acquisition purposes, and some manufacturers have already announced future products in this area. One nice feature of USB is that a limited amount of power is available from the USB connector, thereby eliminating the need for extra power cables for external devices.

Network-Distributed Data Acquisition

With the decreasing cost of hardware and the flexibility afforded by Linux, we have been planning to use distributed hardware control. Instead of having one workstation linked to all the peripherals, we can deploy several stripped-down computers (hardware servers), equipped with a network card but no keyboard or monitor. Each of these would talk to a subset of hardware, executing commands sent by the main control workstation (controlling client) over the network. For the servers we can use either older, recycled 486-class machines, or even the industry standard PC-104 modules (a miniature PC format composed of stackable modules around 10 cm by 10 cm in size). In this case, the Ethernet becomes our real-world interface.

Scientific Visualization and Computations

Of course, we have also used Linux in the more traditional role as a general-purpose graphics workstation. Here, we are no longer limited to x86 architecture. Since we don't have to contend with hardware issues, we have a choice of several platforms, including Digital's Alpha-based computers. Currently (February 1998), it is possible to buy a 533MHz Alpha workstation for just over $2000 US; the prices seem to be going down, while the clock speed is going up into the reputed range of 800MHz. Alpha Linux is ready for serious computations at a very low price. Alpha is an excellent performer, especially for floating-point calculations—the SPEC benchmark shows an integer computation speed (SPECint95) of 15.7, and floating-point computation speed (SPECfp95) of 19.5 for a slightly slower 500MHz Alpha. By comparison, Pentium II at 233MHz exhibits SPECint95 of 9.49 and SPECfp95 of 6.43, one third the floating-point performance of the Alpha chip.

We have been using Linux-based PC stations since 1995. Often they simply serve as capable remote clients for our departmental computer servers, providing better X terminal functionality at prices lower than some commercial X terminals and for light local office computing. More and more, however, we have used them to perform local computations. An intriguing project we are currently considering is installing networked server processes for a certain kind of calculation often performed here (non-linear fitting) and distributing parallel calculations to servers which are not currently used by other clients. Since an average personal computer spends most of its cycles waiting for keystrokes from the user, we are planning to profitably use those free cycles.

This approach is, of course, inspired by the Beowulf cluster project, where ensembles of Linux boxes are interconnected by dedicated fast networks, to run massively parallel code. (See “I'm Not Going to Pay a Lot for This Supercomputer!” by Jim Hill, Michael Warren and Pat Goda, *Linux Journal*, January 1997.) There are several Beowulf installations in the Washington DC area including Donald Becker's original site at NASA, and the LOBOS cluster at National Institute of Health. In contrast, we plan to use non-dedicated hardware, connected by a general purpose network. We can get away with this because our situation does not require much inter-process communication.

Unfortunately, we don't have much space to discuss the scientific visualization; it is a fascinating field both scientifically and aesthetically; it involves modern 3-D graphics technologies, and some of the images are very pretty. Computer visualization is a new field, and consequently the development tools are as important as end-user applications. In our judgment, the best environment for graphics is provided by the OpenGL environment. OpenGL is a programming API designed by Silicon Graphics for their high-performance graphics systems, which is beginning to appear on Windows; on Linux it is supported by some commercial X servers. Also, Mesa is a free implementation of OpenGL that runs on top of X and makes OpenGL available on Linux. This involves a tradeoff—3-D graphics are significantly slower without the hardware assist provided by high-end graphics hardware and matching OpenGL implementation, but the X Window System's advantage of not being tied to any particular terminal is very significant.

Building on Mesa/OpenGL, there are many excellent visualization programs and toolkits, some referenced in the Mesa page. In particular, the VTK visualization widgets and their accompanying book are worth mentioning. Another application is GeomView, a generic engine for displaying geometric objects, written at the University of Minnesota's Geometry Center.

Scripting and Very High-Level Languages

We have found the scripting-software methodology is very useful for both scientific computing and data acquisition. Scripting is a style of writing software where, instead of constructing a monolithic program with a hardwired control flow, we restructure the code by dividing it into modules which perform parts of the work. To glue these modules together, we compile them with a command-language interpreter.

High-Level Language (HLL) interpreters have been around for a long time (Scheme, Basic, Perl, Tcl, Python), but only recently has there been an emphasis on embedding them within users' programs. Even without such embedding, interpreters are still useful for prototyping, but they tend to run out of steam for larger projects. The key is to put together the flexibility of an interpreted system and the speed and functionality of the compiled HLL code.

For example, let's imagine a program that opens and processes a configuration file, asks the user for input and calculates some results. Traditionally, the control flow of such a program is hardwired in its main routine; each I/O phase is programmed separately with a separate syntax for each phase's data. (The configuration file might be a table of numbers, and the user input might have a form of simple ASCII strings representing commands.)

To rewrite this program in a scripting style, we would recast configuration and calculation phases as separate modules invoked by a scripting interpreter. The data for work modules would be kept in interpreter's variables, while the I/O would be handled by interpreter's native facilities. In order to complete the program, we have to write a short interpreter script that reads the configuration file, stores and processes the values, obtains user input, launches the calculation and outputs the result. The important point, and the one that takes a little while to get used to, is that there is no longer a hardwired control flow in the program: when it is started, the interpreter takes over and awaits the script (either from the command line or from a script file) to set the modules in motion.

Of the several modern scripting languages, we have chosen to use Tcl/Tk. Others, such as Python and Perl, are equally good and have similar capabilities. We have written a significant number of Tcl extensions, dealing with abstractions for platform-independent self-describing data files, binary data matrices and image processing, arithmetic expressions and others.

There are several benefits to the scripting approach. First, it provides for more flexibility: it would be trivial to change the interpreted script to perform two rounds of computations instead of one. Also, it is much easier to decouple the user-interface code from the computational code—all that is needed to add GUI data input is to rewrite the user-interface portions of the script so that it uses the interpreter's GUI widgets.

Second, the interpreter usually provides general-purpose linguistic constructs, such as macros/procedures, and looping and conditional statements. This makes it possible to write sophisticated and flexible batch processing scripts.

Note that a properly designed scriptable application reconciles an artificial and unnecessary distinction between command-line and GUI-based programs. The premise behind graphical user interfaces is to provide visual cues for all operations; however, the tradeoff is often that other operations, for which no GUI element was included, are impossible. In other words, a GUI promises a “What You See is What You Get” operation, but it often delivers “What You Get is What You Get”.

With scripting, the GUI is set up to invoke predefined command lists; at the same time, the interpreter can be directed to accept user-typed commands or file input, allowing for arbitrary command sequences. It is nice to be able to select a file using a file selector dialog, but anyone who has had to negotiate such file selection for a hundred files must appreciate the utility of typing **process *.dat** on the command line.

The final benefit of an extensible scripting language is that it is well-suited to create abstractions for complex objects or actions. Such abstractions are good for two reasons: they make complex manipulations easier to understand and perform, while at the same time they enable high performance since they are compiled extensions. A good example might be BLT, a Tcl graphing extension we often use. It is a sophisticated graphing tool with dozens of options. Its complex internal structure is simply encapsulated: the advanced options are available, but don't have to be used. All that is needed for a simple plot is to provide values for the X and Y coordinates of the plot. At the same time, because it is a compiled extension to Tcl, BLT enjoys quite good performance, even on large plots, comparable to visualization tools written entirely in C.

Thanks to the dynamic loading of shared libraries and extensions, an existing program can be enhanced with graphing capability by simply loading the BLT package. This creates the new **graph** command in the Tcl interpreter, which can then be used in the script that constructs the GUI.

Another example of a useful software abstraction that pops up in several places in our work is the numerical array. Such arrays are extremely important in science: they may contain vectors of data, geometrical coordinates, matrices, etc. The standard HLLs usually have a concept of such an array, but it is usually a second-class object. Arrays provide space for storage of data, but it is not possible to perform infix arithmetic operations on them in the same way as on simple, scalar variables. The array processing in such languages is done one element at a time, which is prohibitively slow for large matrices (see example below). (Of course, FORTRAN90 and as C++ with appropriate matrix algebra libraries allow writing computations like A*B for the matrices as well as scalar variables, but these environments aren't common yet, either on Linux or on commercial platforms.)

Typical C (HLL) code for doing a matrix multiply is as follows:

```
for( i=0 ; i<N ; i++)
    for( j=0 ; j<N ; j++)
      for( k=0 ; k<N ; k++)
        C[i][j] = A[j][k] * B[k][i]
      }
    }
  }
```

For Matlab/Octave (VHLL), the code looks like this:

```
C = A * B
```

The VHLL code is obviously easier to write. Also, in an interpretive language, the loop iterations are interpreted one by one; in VHLL, the whole operation is executed in machine code at full speed.

The term “Very High-Level Languages” refers to such problem domain-specific languages. For numerical computation, there is a commercial VHLL called Matlab. It provides a sophisticated environment for calculation and display of numerical data, with array variables as first-class objects. It is a very nice toolkit and is supported on Linux. Interestingly, there is a free clone of Matlab, called Octave, that provides a large part of its functionality; Matlab code typically runs unchanged in Octave. (See “Octave: A Free, High-Level Language for

Mathematics" by Malcolm Murphy, *Linux Journal*, July 1997.) Those systems are addictive; once you use them for a while it is hard to go back to FORTRAN.

The above remarks are equally relevant on any OS platform, whether it is different flavors of UNIX or even on Windows or Macintosh. However, Linux provides the most complete software development environment. Various native scripting systems exist on individual platforms: Visual Basic on Windows, Hypercard and Metacard on a Macintosh; however, the commercial offerings are never complete. For instance, Visual Basic requires a separate C compiler to create binary extensions. On the other hand, Linux provides all the tools (Tcl/Tk libraries and header files, GCC compiler, etc.) out of the box.

Resources

Przemek Klosowski is a physicist working at National Institute of Standards and Technology. Since he stumbled onto the Internet 13 years ago, Linux 6 years ago and founded Washington DC Linux User Group 4 years ago, he is beginning to feel like an old geezer. This feeling is reinforced by his failure to get excited by Java. Still, his youthful enthusiasm is maintained by the success of Linux and other Open Software initiatives that he supports and sometimes contributes to. He can be reached via e-mail at przemek@nist.gov.

Nick Maliszewskyj is a physicist at the NIST Center for Neutron Research in Gaithersburg, MD, where he loves to play with the big toys to be found there. His current mission is to write software that will let hordes of other people play with them too. Activities in the non-binary world include Aikido, home repair, watching his 3-year-old son with amazement and preparing for the arrival of his second child. Nick can be reached by e-mail at nickm@nist.gov.

Bud Dickerson has always worked for physicists because they let him play with cool toys. He sleeps too well at night, however to be any better with Linux than he is. He can be reached at bud.dickerson@nist.gov.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Global Position Reporting

Richard Parry

Issue #51, July 1998

Although the GPS was originally intended for use by the military, in peace time it has given rise to applications that were heretofore limited to science fiction.

The Global Positioning System (GPS) has given rise to many unique applications and is destined to make its mark among the technological wonders of the world. The Automatic Position Reporting System (APRS) is an application that uses the GPS to allow amateur radio operators to broadcast latitude, longitude, heading, velocity and weather to remote receivers. Linux plays an important role in this application by providing the gateway between wireless APRS LANs and the Internet. This article provides an introduction to the GPS and APRS, and describes how Linux is being used to develop a nationwide APRS backbone. Also included is a list of hosts and web sites to which Linux users can connect to obtain real-time position reports. I will also discuss the Linux applications **aprsmon**, **aprsd** and PerlAPRS which take advantage of the power of Linux and the Internet to extend the usefulness of the GPS.

When historians look back upon the engineering accomplishments of the twentieth century, the Global Positioning System (GPS) is certain to be among the top engineering wonders. It represents major accomplishments in computer hardware and software, reliability, satellite technology, physics, communication and electronic engineering. By any standard, it is a marvel and a testament to the belief that mankind can accomplish anything the imagination can think of.

As Arthur C. Clarke, science fiction author and “father” of the geosynchronous satellite, once said, “Any sufficiently advanced technology is indistinguishable from magic.” In many ways, that phrase describes the GPS perfectly—it is magic. Although virtually everyone has heard of the GPS today, it wasn't always this widely known. I remember being handed a small GPS receiver a few years ago and being told that this little device would tell me where I was located anywhere on earth. I could not believe it and was not prepared to be sucked into this canard. How could this device, barely the size of a cellular phone, tell me where I was located within a few hundred feet? It just couldn't be; this had to be a hoax. Upon further discussion and a demonstration, I was hooked; I knew I had to have one, but wasn't sure why. When the Automatic Position Reporting System (APRS) was being developed, I knew I had found my excuse.

The Automatic Position Reporting System is one of the peacetime applications; it unites the GPS with amateur radio. APRS is one of the most popular facets of amateur radio today, and Linux supports APRS with several unique applications. For example, if you wish to know the location of a float in the Rose Bowl parade or the location of the Olympic Torch, APRS can provide that information.

APRS

The Automatic Position Reporting System allows amateur radio operators to send and receive position reports obtained from either a GPS receiver or a known fixed position. (See APRS Formats.) In fixed-position applications, radio frequency packet transmissions are broadcast from a stationary location such as a building or home. Since the station is fixed, there is no need for a GPS receiver to continually update its position. More interesting are mobile applications in which vehicles are tracked.

APRS Formats

Most GPS receivers have a graphic liquid crystal display (LCD) which is normally attached to the dash of the car for easy viewing. A cable connects the internal GPS receiver to an external antenna. The external antenna is important since it provides better reception. Although a GPS receiver provides visual information to the occupants of the vehicle, virtually all GPS units provide an RS-232/4800 baud connection to allow the receiver to connect to an external device such as a laptop computer. However, for APRS applications, we are interested in broadcasting our position to the wireless APRS network. Therefore, the serial output of the GPS receiver is connected to a terminal node controller (TNC), which acts like a modem and changes the digital data stream to analog tones. The tones are then fed into a transmitter which broadcasts packets containing GPS position information. This configuration is shown in Figure 1.

Figure 1. GPS Configuration

Figure 2. GPS Satellite Antenna and Mr. Parry

Figure 2 shows an example of a tracker. Here, an ordinary automobile is shown with some not-so-ordinary equipment attached to the trunk. The object of interest, located in the center of the trunk, is a GPS satellite antenna. Also shown is a vertical whip antenna, used to broadcast APRS packets from a transmitter located within the vehicle. A second vertical whip antenna is used for voice communication. Pay no attention to the man behind the curtain (i.e., leaning on the car).

APRS Servers

There is currently a nationwide effort to provide the information received by local APRS LANs to the Internet. This is done using APRS servers which provide live APRS traffic to the Internet. By using a simple TELNET client, one can connect to a server and see the information that is being collected throughout the United States. Several APRS servers are available for different operating systems. For Linux users, there are presently two APRS servers available: **aprsmon** and **aprsd**.

The aprsmon server can be found at http://www.cloud9.net/~alan/ham/aprs/; aprsd can be found at http://www.wa4dsy.radio.org/Files/aprsd.beta101.tar.gz. APRS servers allow users to connect and examine remote APRS networks located in several metropolitan areas. The nationwide network of servers is expanding with the ultimate goal of allowing one to locate mobile trackers anywhere in the U.S. To better understand the information provided by these servers, try a TELNET session to any of the addresses listed below. The numeric value after the host name is the port number and is required.

- kb2ear.aprs.net 14579 (Northern NJ)
- kb2ear.aprs.net 6261 (USA)
- kb2ear.aprs.net 14580 (Composite of above)
- www.wa4dsy.radio.org 14579 (Atlanta, GA)
- socal.aprs.net 14579 (Southern CA)
- www.aprs.net 10151 (USA Composite)
- www.aprs.net 14579 (Southeast FL)
- sboyle.slip.netcom.com 14579 (San Francisco, CA)

The information returned from these TELNET sessions is real-time raw data that is broadcast by amateur radio operators at intervals from 1 to 30 minutes. The short duration broadcasts (e.g., one minute) are intended for mobile (tracker) applications where there is movement and therefore a need for frequent updates. The longer duration broadcasts (e.g., 30 minutes) are intended for fixed stations (homes) broadcasting their locations. These servers provide packets which include the position of the transmitting station's latitude, longitude and often a brief message about the station. Listing 1 is the output from a typical APRS TELNET session.

Each line of text in Listing 1 is a packet that contains the amateur radio call sign of the source station, the destination address and any repeaters used in the path. For example, in the first packet shown after the login message, W4DUF is broadcasting to all stations in the APRS network. Due to distance limitations (typically a few miles), other local stations along the route, called digipeaters (digital repeaters), are used to extend the distance by repeating the packet. In the example, stations N4TKT-2, WIDE and N4NEQ-2 are being used to repeat the packet. In this way, distances can be extended to a large metropolitan area (i.e., a 20 mile radius), as well as across the nation by using special high frequency (HF) digipeaters called GATEs. Due to limited RF bandwidth, broadcasting positions nationally is typically limited to special events such as tracking the Olympic torch as it traveled across the U.S.

Figure 3. APRS WWW Page

With the development of a nationwide APRS backbone using the Internet, transmitting local APRS traffic can bypass the constraints of limited HF bandwidth. An APRS TELNET session as shown in Listing 1 is interesting, but difficult to understand due to the raw format of the information. To better understand the data being presented, graphically formatted web pages are used. These web sites take the raw information and overlay the location of the stations on a map. Figure 3 is an example of a typical APRS web page. The list of web sites below shows real-time or near real-time (delayed 15 minutes) APRS traffic and requires a Java-enabled Net browser.

- http://www.wa4dsy.radio.org/aprs/usa.html (Entire US)
- http://www.wa4dsy.radio.org/aprs/soeast.html (Southeastern US)
- http://www.wa4dsy.radio.org/aprs/ga-atl.html (Atlanta, GA)
- http://www.aprs.net/sfl.html (Southern Florida)
- http://sboyle.slip.netcom.com/LIDSAPRS.html (San Francisco, CA)

PerlAPRS

As we have seen, APRS servers provide raw data, and web browsers can show the information in a graphically interesting and informative manner. However, both are passive applications that require a user.

For many applications, it would be nice to automate the system to perform a specified task automatically. For example, you might wish to be informed by e-mail that the lead float in the Rose Bowl Parade has reached a specific point in the route. Or perhaps you have a mobile tracker and wish to sound an alarm when the tracker reaches a specific location. For this application, you need a program that will examine the raw APRS data and execute a command based on user-specified criteria. This is exactly what PerlAPRS does, and Linux is the perfect platform for this type of application since it supports multitasking so well.

PerlAPRS examines incoming packets and executes a command when a call sign and location match the criteria specified by the user. Location criteria is specified using grid squares, a rectangular area measuring approximately 2.5 by 5 miles.

The best way to understand how PerlAPRS works is to look at an example. The line numbers shown in the left-hand column below are provided for illustrative purposes and are not part of the normal output. Line 1 shows an example packet. PerlAPRS parses the packet and extracts the call sign, latitude and longitude. Line 2 displays the call sign as KD6AZU, the latitude as 3243.700 (32 degrees, 43.700 minutes North) and the longitude as 11707.700 (117 degrees, 7.700 minutes West). Next, PerlAPRS searches a call sign file, previously customized by the user, looking for a match. The first two attempts at a match shown on lines 3 and 4 fail. The third comparison shown on lines 5 and 6 is successful. This match causes the command, **cmd3.sh**, to be executed. The command may be any UNIX-style command; however, simple shell scripts are used for most applications.

```
1. Packet= KD6AZU>APRS,KD4DLT-7,N4NEQ-2,WIDE*:
@042327/3243.70N/11707.70W/0
2.  KD6AZU 3243.700 11707.700
3. - KI6MP-10 DM12JV
4. - KC6VVT-9 DM12IT
5. * KD6AZU DM12KR Sun Aug 10 15:56:13 1997 3
6.     Sun Aug 10 15:57:13 1997 1 cmd3.sh
```

This brief discussion of PerlAPRS is intended to provide a simple overview. The program provides several additional features intended for real time applications. PerlAPRS is distributed under the GNU licensing agreement. The source code and further information on the program can be found at http://people.qualcomm.com/rparry/perlAPRS/.

GPS Primer

Linux, amateur radio and the APRS protocol are only part of the system we have discussed so far. The GPS is truly what makes the system practical. Although the details of the GPS are extremely complex, the basic idea is relatively simple. Triangulation is used to pinpoint a receiver.

For example, assume you and your friend both have accurate synchronized clocks. At some unknown distance from you, she yells, “It is now 6:00 and 0.000 seconds.” When you hear her, your clock shows the time as 6:00 and 0.333 seconds. You can now compute your distance from her as 100 meters by multiplying the speed of sound (300 meters per second) by the elapsed time (0.333 seconds). With this single test point, you are able to compute your distance from the source. Specifically, you are located in any direction 100 meters from your friend. This scenario is shown in Figure 4A by the multiple dots located on the circumference of the circle.

Figure 4. How GPS Works

With a second friend, we can further clarify our position. In Figure 4B, a friend at point Y, also with an accurate clock, takes another measurement. Again you make the computation and find the distance from this friend. You now have your position narrowed to two points shown by the two dots where the circles intersect. Using a third friend at point Z and another measurement, you are able to pinpoint your exact location.

The GPS works on a similar principle; however, the speed of light replaces the speed of sound in the experiment, and your friends are replaced by satellites. In the above explanation we have conveniently assumed that the world is flat to provide a clearer understanding of the concept. When the concept is extended to three dimensions, a single measurement does not produce a circle as shown in Figure 4A, but a sphere. A second measurement does not limit our position to two unique locations as shown in Figure 4B, but a circle that is the intersection of two spheres. Last, a third measurement does not yield a unique location as shown in 4C, but two points which are the intersection of three spheres. Thus, with three measurements, we have two possible locations. The good news is that one of the points can be eliminated since it corresponds to a position above the Earth's atmosphere. So unless you are an astronaut, your unique location on earth has been found with only three measurements.

We made a second convenient assumption, specifically that all parties in the experiment had accurate clocks. Although the GPS satellites have accurate atomic clocks, the receiver on the ground does not have such a luxury, nor would it be practical. Fortunately, by adding a fourth satellite, the person on the ground does not require an atomic clock. This is a simple algebraic problem in four unknowns: latitude, longitude, altitude and time. With four satellites we can solve for all four unknowns and provide an accurate and unique position for the listener on Earth. The experienced GPS user may argue that he has obtained accurate positions with only three satellites. This is true; it is done by letting the GPS receiver assume the altitude is 0 (sea level). Therefore, if we are willing to give up knowing our altitude, which is valid in many applications, the GPS can indeed provide an accurate position using only three satellites, since we have three unknowns and three equations.

GPS Accuracy

The above explanation is in many ways an oversimplification. In real life, numerous variables affect the accuracy of the system. For example, radio frequency transmissions are affected by objects such as buildings and trees. These structures cause reflections, referred to as multi-path. Signals from the satellites are reflected off nearby structures, causing delays which ultimately affect the accuracy of the measurement. Radio frequencies are also affected by rain, sleet, snow, humidity and even the temperature of the air, since the speed of the transmission is affected as well as the attenuation of the signal. All of these variables result in loss of accuracy. However, these inaccuracies are small compared with the deliberate error called Selective Availability (SA).

To understand SA, we must understand that GPS applications fall into two service categories: the Standard Positioning Service (SPS) for civilians, and the Precise Positioning Service (PPS) for military and authorized personnel. PPS GPS receivers remove the adverse affects of SA and are therefore far more accurate. SPS GPS receivers provide less accuracy than the GPS is capable of, and each is generally limited to an accuracy of 100 meters. However, there are ways of overcoming the limitations of SPS receivers by using Differential GPS (DGPS). For those interested in DGPS, the web is an excellent source of further information.

Conclusion

The world has not been the same since the invention of the telephone, radio, television and the computer. The GPS is also destined to make its mark in the technological evolution of mankind; it has given rise to many unique applications. The APRS was developed to allow amateur radio operators to broadcast positions using packet radio. APRS servers and Linux further extend the GPS to uses that were science fiction not too long ago.

Resources

Richard Parry works as a software engineer at Qualcomm, Inc., known by most as the home of the e-mail program Eudora. He attends the University of California at San Diego and studies computer science. When not sitting in front of a monitor, he plays racquetball, but does entirely too much of the former and not enough of the latter. He can be reached at rparry@qualcomm.com or visit his home page at http://people.qualcomm.com/rparry/.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Javalanche: An Avalanche Predictor

Richard Sevenich

Rick Price

Issue #51, July 1998

This article introduces a prototypical avalanche-predicting software package implemented with a Fuzzy Logic algorithm.

Javalanche is prototypical in the sense that the current model is too sparse and naive for practical avalanche prediction. Nevertheless, it suggests that Fuzzy Logic may be an appropriate tool for such an application, upon significant enhancement of the model presented here. The software was developed using Java in a Debian/GNU Linux environment. Graphs were created using **gnuplot**.

Variables for Avalanche Prediction

Evaluating avalanche hazard relies on gathering meaningful data from a large number of variables including slope aspect and angle, wind load and direction, terrain roughness, snow crystal forms present in the snowpack, snowpack layer resistances, the layering effect of strong over weak zones, current temperature and temperature history, and recent snowfall depth and water content. It is noteworthy that both long-term and current variables belong in any usable model, that some factors are interrelated and that a factor may or may not play a predominant role at some particular time.

To be practical, the values of the input variables should be relatively straightforward to measure in environments ranging from tamed ski areas to untamed wilderness. Many of the typical assessment tools are qualitative but have proved their worth. Snow layers can be assessed by digging a snow pit and examining the pit walls for snow crystal forms, temperatures and layer resistances. A common method for assessing snow layer resistance is a hand test which measures the level of resistance the snow layer presents to penetration. These levels are categorized as fist, four finger, one finger, pencil and knife in order of increasing resistance. This aids in determining the existence of a buried instability. A technique for assessing the amount of surface snow that can be transported by wind is the foot penetration test. The tester steps on the snow with one foot and measures the penetration, with 30cm being considered enough to suggest a potential hazard. A refinement would attempt to factor in the weight and foot area of the tester. There are other such tests. Slope aspect is the compass direction the slope faces. Its hazard effect will be influenced by wind direction and exposure to the sun. The latter influence varies with the time of year. A good web site related to these issues with links to other sites is the Cyberspace Snow and Avalanche Center at http://www.csac.org/.

The bottom line is that a reasonably useful model will employ many variables, need extensive testing and refinement and require significant input from experienced avalanche personnel. It is clearly easier to apply the model in a developed ski area rather than in the back country. The computer models of which we are aware are mechanistic in nature, e.g., there is European work using finite element analysis. We feel that Fuzzy Logic is an appropriate tool and advance this article to explain the approach. We stress at the outset that this paper is expository and the model presented is not yet usable in a practical setting. However, we would approach a mature model by including new variables one at a time and testing the resulting software. Further, we have not even chosen the most important variables, but rather a handful that are easily understood.

Essential Elements of Fuzzy Logic

Articles and books describing Fuzzy Logic are widely available, as a cursory web search will quickly confirm. We recommend Earl Cox's book as a first, practical exposure (*The Fuzzy Systems Handbook*, AP Professional, 1994). First devised by Lotfi Zadeh (“Fuzzy Sets”, *Information and Control*, Volume 8, 338-353, 1965), Fuzzy Logic is best known for its applications in industrial control. However, it is also quite successfully used in decision-making applications, which is the basis of our project.

Fuzzy Logic is particularly appropriate in situations where a mathematical model is either unavailable or too unwieldy and where human expertise gleaned from experience and supported by intuition is available. In particular, it emulates the human reasoning process and employs linguistic forms in its modeling process. For this article the first author is the Fuzzy Logic programmer, and the second author provides the avalanche expertise.

In this article, we will introduce Fuzzy Logic via our problem space. This approach will give you insight into the concepts via a somewhat detailed example application. However, the scope of this article does not allow us to present Fuzzy Logic formally, nor in its full richness.

The minimal ingredients of a Fuzzy Logic model include these elements:

- One or more input variables
- A family of fuzzy sets for each input variable
- One or more output variables
- A family of fuzzy sets for each output variable
- A group of rules connecting input and output variables

There are also algorithms which are applied to the model:

- Fuzzification of crisp input variables
- Application of the rules
- Defuzzification of rule results to achieve crisp outputs

The terminology embedded in the preceding two lists will become familiar as we work through the Avalanche Predictor example.

The Fuzzy Sets for the Javalanche Model

The model is to be applied when there has been snowfall during the last 24-hour period. There are three input variables:

- Slope_Pitch, the average slope angle (degrees) in the region of the suspected avalanche danger
- Water_Equiv, the snowfall's water content (centimeters of equivalent water)
- Current_Temp, the current temperature (Celsius)

To introduce fuzzy sets, we'll start with the input variable, Slope_Pitch. Wild slopes do not, of course, have constant pitch and even a measurement of average pitch is approximate. Nor is it clear that the distinction between a number like 15.2 degrees and 17.3 degrees is all that useful. Fuzzy sets provide a way to incorporate that inherent fuzziness into a model. We somewhat arbitrarily classify the Slope_Pitch variable into four categories, based loosely on the corresponding skiing ability needed to competently negotiate the terrain. These categories are Novice, Intermediate, Advanced and Expert.

Figure 1. Fuzzy Set for Novice Slope_Pitch

Figure 2. The Four Fuzzy Sets for Slope_Pitch

There is no widely accepted ski industry standard for these terms, but there is an approximate agreement on what they imply. For example, most skiers would consider the pitch range of 0 to 10 degrees as Novice, but there would be less agreement on the angle at which the slope would be considered no longer Novice, but Intermediate. Fuzzy Logic would accommodate this uncertainty by defining a fuzzy set for novice slope pitch as shown in Figure 1, where the vertical axis is called the degree of membership (dom). In Figure 2, the fuzzy sets for Intermediate, Advanced and Expert are incorporated as well. Looking at Figure 2, an input Slope_Pitch of 17.5 degrees would have a degree of membership of 0.25 in the Novice category and of 0.75 in the Intermediate category, reflecting the fuzzy transition from Novice to Intermediate Slope_Pitch. Ascertaining the doms of the various input values is called the fuzzification process.

Figure 3. The Three Fuzzy Sets for Water_Equiv

Figure 4. The Three Fuzzy Sets for Current_Temp

Figures 3 and 4 show fuzzy set choices for the other two input variables, Water_Equiv and Current_Temp. The choices of fuzzy set ranges and shapes are somewhat arbitrary, but should be guided by the knowledge of the expert. From Figures 2, 3, and 4 we see that the model has the following sets:

- Four fuzzy sets for Slope_Pitch
- Three fuzzy sets for Water_Equiv
- Three fuzzy sets for Current_Temp

There is only one output variable, Avalanche_Danger. It is scaled from 0 to 100. It is tempting to interpret this as the probability of avalanche, but at this current stage of development it is an arbitrary scale. If the model were significantly enhanced and then used both extensively and successfully, this parameter could be calibrated and perhaps be rather like a probability. Figure 5 depicts the four fuzzy set categories for Avalanche_Danger.

Figure 5. The Four Fuzzy Sets for Avalanche_Danger

Note that the expert snow scientist must be consulted by the programmer to construct the fuzzy sets. It can be expected that these would be modified and additional inputs incorporated as experience with the model is gained.

The Rules for the Javalanche Model

Rules come in both conditional and unconditional varieties. For Javalanche, only conditional rules are currently implemented. A typical rule might be “If Water_Equiv is Small AND Slope_Pitch is Novice AND Current_Temp is

Below_Freezing, then Avalanche_Danger is Low." The **if** clause (antecedent) of the rule contains input fuzzy sets, while the **then** clause (consequent) contains output fuzzy sets. Each of the rules here links three fuzzy sets in the antecedent with the "AND" conjunction. Each consequent involves a single output fuzzy set.

<u>**Figure 6. Rules for Current_Temp = Below_Freezing**</u>

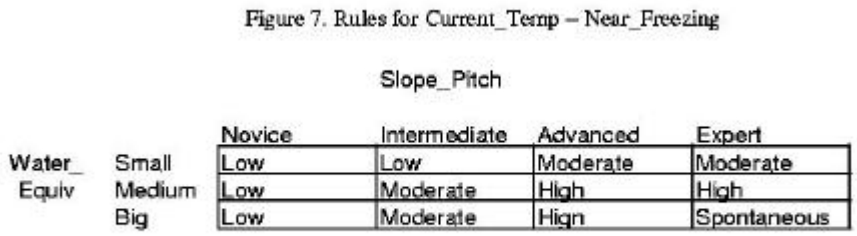


Figure 7. Rules for Current_Temp – Near_Freezing

Slope_Pitch

		Novice	Intermediate	Advanced	Expert
Water_ Equiv	Small	Low	Low	Moderate	Moderate
	Medium	Low	Moderate	High	High
	Big	Low	Moderate	Hign	Spontaneous

Figure 7. Rules for Current_Temp = Near_Freezing

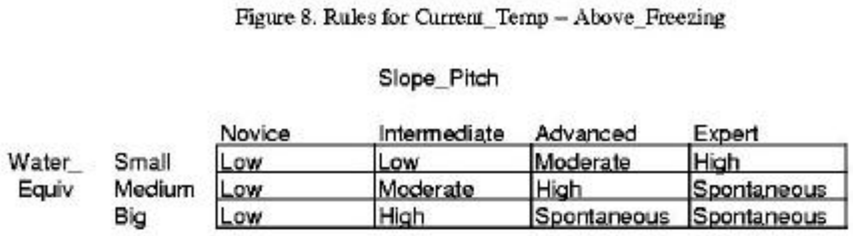


Figure 8. Rules for Current_Temp – Above_Freezing

Slope_Pitch

		Novice	Intermediate	Advanced	Expert
Water_ Equiv	Small	Low	Low	Moderate	High
	Medium	Low	Moderate	High	Spontaneous
	Big	Low	High	Spontaneous	Spontaneous

Figure 8. Rules for Current_Temp = Above_Freezing

Recall that the multiplicity of fuzzy sets for the three input variables is 4, 3 and 3, so that the total number of rules is the product, 36. Rather than quote each of the 36 rules, we represent them with the three tables shown in Figures 6, 7 and 8. Extracting a rule from a table is straightforward. The table entries show Avalanche_Danger for two inputs, Water_Equiv (row) and Slope_Pitch (column) while the third input is contained in the figure label. For example, in Figure 6, the upper left corner entry is "Low" and the corresponding inputs are:

- Water_Equiv = Small (row)

- Slope_Pitch = Novice (column)
- Current_Temp = Below_Freezing (Figure 6's label)

Hence the related rule is, “If Water_Equiv is Small AND Slope_Pitch is Novice AND Current_Temp is Below_Freezing, then Avalanche_Danger is Low”; the same rule quoted earlier.

Just as for the fuzzy sets, the expert snow scientist must be consulted by the programmer in order to compose adequate rules. As with the fuzzy sets, experience with applying the model in the real world will most likely result in adjustments to the rules.

A Sample Calculation

To see how a Fuzzy Logic algorithm works, we'll make an example calculation. Of course, such calculations are done by the program, but hand calculations are essential for understanding and for debugging the program. The steps we'll go through are:

1. Start with three crisp input values.
2. Fuzzify those three values.
3. Evaluate the appropriate rules from the 36 available, obtaining fuzzy outputs.
4. Defuzzify the outputs to obtain a crisp output.

Let's say we have measured/estimated the three input variables to be Slope_Pitch = 17 degrees, Water_Equiv = 5 centimeters, and Current_Temp = 3 Celsius. These are the crisp values.

To fuzzify an input variable means finding its doms in all its fuzzy sets. Using Figure 2, we find that Slope_Pitch has these doms in its fuzzy sets:

- Novice dom = 0.3
- Intermediate dom = 0.7
- Advanced dom = 0.0
- Expert dom = 0.0

Similarly, from Figure 3, the Water_Equiv values are

- Small dom = 0.0
- Medium dom = 1.0
- Big dom = 0.0

Last, from Figure 4, the Current_Temp values are:

- Below_Freezing dom = 0.0
- Near_Freezing dom = 0.5
- Above_Freezing dom = 0.5

This completes the fuzzification process.

After fuzzification, the rules are evaluated. Not all the rules will apply in each instance. In particular, if any of the three inputs has a dom = 0.0, then that rule does not apply. From the preceding dom calculation we see that two fuzzy sets for Slope_Pitch, one fuzzy set for Water_Equiv, and two fuzzy sets for Current_Temp have nonzero dom values. Consequently, four (= 2x1x2) rules apply; namely, the first two in the middle row of Figure 7 and the first two in the middle row of Figure 8.

We'll continue our sample calculation by evaluating only one of the four rules. Let's consider the rule that has a consequence of Moderate Avalanche_Danger, from Figure 7: “If Water_Equiv is Medium AND Slope_Pitch is Intermediate AND Current_Temp is Near_Freezing, then Avalanche_Danger is Moderate.”

To evaluate this rule, we combine the doms of the antecedent fuzzy sets by forming their product:

- Slope_Pitch has Intermediate dom = 0.7
- Water_Equiv has Medium dom = 1.0
- Current_Temp has Near_Freezing dom = 0.5

The product = 0.35 is then assigned to the output, i.e., the Avalanche_Danger value has a dom of 0.35 in the Moderate fuzzy output set. Using the product of the doms to combine the fuzzy sets joined by the AND conjunction is called the “product AND”. Fuzzy Logic allows other choices (see Cox's book).

The other three rules which apply in our case must also be evaluated. We won't do those calculations here—they are quite similar to the evaluation of the first. Note that of the four rules that apply, two have a consequence of Moderate and two have a consequence of Low. We choose to combine the dom values for the Low fuzzy set by adding them together, thus allowing each rule that fires to have an effect. We do the same thing for the Moderate doms. This is often done in decision-making problems, but is not the only option possible (again, see Cox's book). Hence, we now have these dom values for Avalanche_Danger:

- Low = 0.3
- Moderate = 0.7

- High = 0.0
- Spontaneous = 0.0

These dom values are then “defuzzified”, as in Figure 5. After looking at the figure with these dom values, it seems reasonable to conclude that the resultant number will be between 10.0 and 20.0, and because the Moderate dom is stronger, it ought to be closer to 20.0 than to 10.0. In practice, we use a weighted average known as the “center of gravity”, and it yields 19.0 for this case. We won't do the detailed calculation here.

Thus, for our sample calculation, the input values of Slope_Pitch = 17 degrees, Water_Equiv = 5 centimeters, and Current_Temp = 3 have led to an output value of Avalanche_Danger = 19.0, a value mostly in the Moderate region, but with some membership in the Low region.

An Overview of the Software

The software is available via anonymous FTP from ftp://turing.sirti.org/pub/ras/fuz3.tar.gz. When unzipped and unarchived, it will produce a directory tree with fuz3 as the top node. The top node contains a README file, enabling a user to both use and modify the package. To execute the software, it is assumed that the user's machine has Java properly installed. We used JDK1.1.1.

In the lowest subdirectory, io_n_sets, three files contain the fundamental classes chosen for the model, as follows:

- ioput.java contains a class for input and output variables.
- fz_set.java contains a class for the fuzzy sets.
- cond_rule contains a class for the conditional rules.

These classes contain no information specific to the avalanche prediction model.

The parent directory of io_n_sets is init_n_run which contains two source files of interest: make_init_file.java and run_eng.java. The first of these creates an initialization file, fz_init.dat, which is read by run_eng.java to initialize its Fuzzy Logic “engine”. Only make_init_file.java contains the model for the avalanche predictor. Hence, it may be modified to apply the software to other decision-making problems. As expected, after initializing itself, run_eng.java requests the input variable selection from the user, then runs the Fuzzy Logic engine and produces an output result.

The software can be executed from a terminal window in the X Window System environment by entering the command:

```
appletviewer run_eng.html
```

Possible Future Work

Here we discuss two topics as possible improvements:

- Refining and extending the Javalanche application
- Replacing make_init_file.java with a user language and translator

To refine and extend the Javalanche application would require field testing and model refinement/enhancement by an active avalanche control group. The earlier portion of this paper identified various other important input parameters which we will investigate. Even if this does not prove feasible, we believe we have made a case for the use of Fuzzy Logic in avalanche prediction.

The approach using make_init_file.java serves to isolate/modularize the specific application, but is not user-friendly. A preferable approach is to allow a user to employ a simple editor to create a text file containing the application-specific details. This is to be written in a language designed specifically for this purpose (a user-specific language). This is then run through the translator whose output is an initialization file, functionally similar to fz_init.dat. The translator can provide a very important feature not provided by make_init_file.dat. In particular, the translator will check the text file written by the user for any errors which are not intrinsically run-time errors. This could then be used by an avalanche control group whose personnel need not be programmers and must merely learn a descriptive text modeling system based on terms familiar to them.

The translator could also produce a second set of files appropriate for producing graphical views (e.g., using gnuplot) of the fuzzy sets for the user. The designing, implementation, and testing of the translator will most likely be assigned as a homework project for students in the compiler design course at Eastern Washington University. This task could be accomplished in a straightforward manner using **flex** and **bison**, compiler construction tools available within Linux. There are also Java versions of these tools for Linux which may be mature by now.

Richard Sevenich (rsevenich@ewu.edu) is a Professor of Computer Science at Eastern Washington University in Cheney, WA. He is also a part-time ski patroller at Schweitzer Mountain near Sandpoint, Idaho. His computer science interests include Fuzzy Logic, Application-Specific Languages and Parallel, Distributed, Real-time Industrial Control. He is an enthusiastic user of Debian/GNU Linux.

Rick Price has avalanche control and prediction experience from his many years of work as a full-time ski patroller at Schweitzer Mountain. He typically keeps an active log of the snowpack conditions and history, supported by field data such as snowpack and avalanche records. Over the years he has attended various avalanche courses and clinics. More recently, Rick has become a middle school teacher in the Bonner County School District in Idaho, retaining a part-time involvement with the Schweitzer Ski Patrol. He can be reached at debbyprice@hotmail.com.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

ROOT: An Object-Oriented Data Analysis Framework

Fons Rademakers

Rene Brun

Issue #51, July 1998

A report on a data analysis tool currently being developed at CERN.

ROOT is a system for large scale data analysis and data mining. It is being developed for the analysis of Particle Physics data, but can be equally well used in other fields where large amounts of data need to be processed.

After many years of experience in developing interactive data analysis systems like PAW and PIAF (see Resources), we realized that the growth and maintainability of these products, written in FORTRAN and using 20-year-old libraries, had reached its limits. Although still popular in the physics community, these systems do not scale up to the challenges offered by the next generation particle accelerator, the Large Hadron Collider (LHC), currently under construction at CERN, in Geneva, Switzerland. The expected amount of data produced by the LHC will be on the order of several petabytes (1PB = 1,000,000GB) per year. This is two to three orders of magnitude more than what is being produced by the current generation of accelerators.

Therefore, in early 1995, Rene Brun and I started developing a system, intending to overcome the deficiencies of these previous programs. One of the first decisions we made was to follow the object-oriented analysis and design methodology and to use C++ as our implementation language. Although all of our previous programming experience was in FORTRAN, we soon realized the power of OO and C++, and after some initial “throw-away” prototyping, the ROOT system began to take shape.

In November 1995, we gave the first public presentation of ROOT at CERN and, at the same time, version 0.5 was released via the Web. By then, Nenad Buncic and Valery Fine had joined our team.

Since the initial release, there has been a constantly increasing number of users. In response to comments and feedback, we've been regularly releasing new versions containing bug fixes and new features. In January 1997, version 1.0 was released and in March 1998 version 2.0. Since the release of version 1.0, more than 9,300 copies of the ROOT binaries have been downloaded from our web site, about 500 people have registered as ROOT users, and the web site gets up to 100,000 hits per month.

ROOT is currently being used in many different fields such as physics, astronomy, biology, genetics, finance, insurance, pharmaceuticals, etc.

The source and binaries for many different platforms can be downloaded from the ROOT web site (http://root.cern.ch/). The current version can be used and distributed freely as long as proper credit is given and copyright notices are maintained. For commercial use, the authors would like to be notified.

Main Features of ROOT

The main components of the ROOT system are:

- A hierarchical object-oriented database (machine independent, highly compressed, supporting schema evolution and object versioning)
- A C++ interpreter
- Advanced statistical analysis tools (classes for multi-dimensional histogramming, fitting and minimization)
- Visualization tools (classes for 2D and 3D graphics including an OpenGL interface)
- A rich set of container classes that are fully I/O aware (list, sorted list, map, btree, hashtable, object array, etc.)
- An extensive set of GUI classes (windows, buttons, combo-box, tabs, menus, item lists, icon box, tool bar, status bar and many others)
- An automatic HTML documentation generation facility
- Run-time object inspection capabilities
- Client/server networking classes
- Shared memory support
- Remote database access, either via a special daemon or via the Apache web server
- Ported to all known UNIX and Linux systems and also to Windows 95 and NT

The complete system consists of about 450,000 lines of C++ and 80,000 lines of C code. There are about 310 classes grouped in 24 different frameworks, each class represented by its own shared library.

The CINT C/C++ Interpreter

One of the key components of the ROOT system is the CINT C/C++ interpreter. CINT, written by Masaharu Goto of Hewlett Packard Japan, covers 95% of ANSI C and about 85% of C++. Template support is being worked on, and exceptions are still missing. CINT is complete enough to be able to interpret its own 70,000 lines of C and to let the interpreted interpreter interpret a small program.

The advantage of a C/C++ interpreter is that it allows for fast prototyping, since it eliminates the typical time consuming edit/compile/link cycle. Once a script or program is finished, you can compile it with a standard C/C++ compiler (**gcc**) to machine code and enjoy full machine performance. Since CINT is very efficient (for example, for/while loops are byte-code compiled on the fly), it is quite possible to run small programs in the interpreter. In most cases, CINT outperforms other interpreters like Perl and Python.

Existing C and C++ libraries can easily be interfaced to the interpreter. This is done by generating a dictionary from the function and class definitions. The dictionary provides CINT with all necessary information to be able to call functions, create objects and call member functions. A dictionary is easily generated by the program **rootcint** that uses the library header files as input and produces a C++ file containing the dictionary as output. You compile the dictionary and link it with the library code into a single shared library. At run-time, you dynamically link the shared library, and then you can call the library code via the interpreter. This can be a very convenient way to quickly test some specific library functions. Instead of having to write a small test program, you just call the functions directly from the interpreter prompt.

The CINT interpreter is fully embedded into the ROOT system. It allows the ROOT command line, scripting and programming languages to be identical. The embedded interpreter dictionaries provide the necessary information to automatically create GUI elements like context pop-up menus unique for each class and for the generation of fully hyperized HTML class documentation. Furthermore, the dictionary information provides complete run-time type information (RTTI) and run-time object introspection capabilities.

Installation

The binaries and sources of ROOT can be downloaded from http://root.cern.ch/root/Version200.html. After downloading, uncompress and unarchive (using **tar**) the file root_v2.00.Linux.2.0.33.tar.gz in your home directory (or in a system-wide location such as /opt). This procedure will produce the directory /root. This directory contains the following files and subdirectories:

- AA_README: read this file before starting

- bin: directory containing executables
- include: directory containing the ROOT header files
- lib: directory containing the ROOT libraries (in shared library format)
- macros: directory containing system macros (e.g., GL.C to load OpenGL libs)
- icons: directory containing xpm icons
- test: some ROOT test programs
- tutorials: example macros that can be executed by the bin/root module

Before using the system, you must set the environment variable ROOTSYS to the root directory, e.g., export ROOTSYS=/home/rdm/root, and you must add $ROOTSYS/bin to your path. Once done, you are all set to start rooting.

First Interactive Session

In this first session, start the ROOT interactive program **root**. This program gives access via a command-line prompt to all available ROOT classes. By typing C++ statements at the prompt, you can create objects, call functions, execute scripts, etc. Go to the directory $ROOTSYS/tutorials and type:

```
bash$ root
root [0] 1+sqrt(9)
(double)4.000000000000e+00
root [1] for (int i = 0; i < 5; i++)<\n>
printf("Hello %d\n", i)
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
root [2] .q
```

As you can see, if you know C or C++, you can use ROOT. No new command-line or scripting language to learn. To exit, use **.q**, which is one of the few “raw” interpreter commands. The dot is the interpreter escape symbol. There are also some dot commands to debug scripts (step, step over, set breakpoint, etc.) or to load and execute scripts.

Let's now try something more interesting. Again, start root:

```
bash$ root
root [0] TF1 f1("func1", "sin(x)/x", 0, 10)
root [1] f1.Draw()
root [2] f1.Dump()
root [3] f1.Inspect()
 // Select File/Close Canvas
root [4] .q
```

Figure 1. Output of f1.Draw()

Here you create an object of class TF1, a one-dimensional function. In the constructor, you specify a name for the object (which is used if the object is stored in a database), the function and the upper and lower value of x. After having created the function object you can, for example, draw the object by executing the **TF1::Draw** member function. Figure 1 shows how this function looks. Now, move the mouse over the picture and see how the shape of the cursor changes whenever you cross an object. At any point, you can press the right mouse button to pop-up a context menu showing the available member functions for the current object. For example, move the cursor over the function so that it becomes a pointing finger, and then press the right button. The context menu shows the class and name of the object. Select item **SetRange** and put **-10, 10** in the dialog box fields. (This is equivalent to executing the member function **f1.SetRange(-10,10)** from the command-line prompt, followed by **f1.Draw()**.) Using the **Dump** member function (that each ROOT class inherits from the basic ROOT class TObject), you can see the complete state of the current object in memory. The **Inspect** function shows the same information in a graphics window.

Histogramming and Fitting

Let's start root again and run the following two macros:

```
bash$ root
root [0] .x hsimple.C
root [1] .x ntuple1.C
 // interact with the pictures in the canvas
root [2] .q
```

Note: if the above doesn't work, make sure you are in the tutorials directory.

Figure 2. Output of ntuple1.C

Macro hsimple.C (see $ROOTSYS/tutorials/hsimple.C) creates some 1D and 2D histograms and an Ntuple object. (An Ntuple is a collection of tuples; a tuple is a set of numbers.) The histograms and Ntuple are filled with random numbers by executing a loop 25,000 times. During the filling, the 1D histogram is drawn in a canvas and updated each 1,000 fills. At the end of the macro, the histogram and Ntuple objects are stored in a ROOT database.

The ntuple1.C macro uses the database created in the previous macro. It creates a canvas object and four graphics pads. In each of the four pads, a distribution of different Ntuple quantities is drawn. Typically, data analysis is done by drawing in a histogram with one of the tuple quantities when some of the other quantities pass a certain condition. For example, our Ntuple contains the quantities px, py, pz, random and i. The command:

```
ntuple->Draw("px", "pz < 1")
```

will fill a histogram containing the distribution of the px values for all tuples for which **pz < 1**. Substitute for the abstract quantities used in this example quantities such as name, sex, age, length, etc., and you can easily understand that Ntuples can be used in many different ways. An Ntuple of 25,000 tuples is quite small. In typical physics analysis situations, Ntuples can contain many millions of tuples. Besides the simple Ntuple, the ROOT system also provides a Tree. A Tree is an Ntuple generalized to complete objects. That is, instead of sets of tuples, a Tree can store sets of objects. The object attributes can be analyzed in the same way as the tuple quantities. For more information on Trees, see the ROOT HOWTOs at http://root.cern.ch/root/Howto.html.

During data analysis, you often need to test the data with a hypothesis. A hypothesis is a theoretical/empirical function that describes a model. To see if the data matches the model, you use minimization techniques to tune the model parameters so that the function best matches the data; this is called fitting. ROOT allows you to fit standard functions like polynomials, Gaussian exponentials or custom defined functions to your data. In the top right pad in Figure 2, the data has been fit with a polynomial of degree two (red curve). This was done by calling the **Fit** member function of the histogram object:

```
hprofs->Fit("pol2")
```

Moving the cursor over the canvas allows you to interact with the different objects. For example, the 3D plot in the lower-right corner can be rotated by clicking the left mouse button and moving the cursor.

The GUI Classes and Object Browser

Embedded in the ROOT system is an extensive set of GUI classes. The GUI classes provide a full OO-GUI framework as opposed to a simple wrapper around a GUI such as Motif. All GUI elements do their drawing via the TGXW low-level graphics abstract base class. Depending on the platform on which you run ROOT, the concrete graphics class (inheriting from TGXW) is either TGX11 or TGWin32. All GUI widgets are created from “first principles”, i.e., they use only routines like **DrawLine**, **FillRectangle**, **CopyPixmap**, etc., and therefore, the TGX11 implementation needs only the X11 and Xpm libraries. The advantage of the abstract base class approach is that porting the GUI classes to a new, non X11/Win32, platform requires only the implementation of an appropriate version of TGXW (and of TSystem for the OS interface).

All GUI classes are fully scriptable and accessible via the interpreter. This allows for fast prototyping of widget layouts.

The GUI classes are based on the XClass'95 library written by David Barth and Hector Peraza. The widgets have the well-known Windows 95 look and feel. For more information on XClass'95, see ftp://mitac11.uia.ac.be/html-test/xclass.html.

Figure 3. ROOT Object Browser

Using the ROOT Object Browser, all objects in the ROOT system can be browsed and inspected. To create a browser object, type:

```
root [0] TBrowser *b = new TBrowser
```

The browser, as shown in Figure 3, displays in the left pane the browse-able ROOT collections and in the right pane the objects in the selected collection. Double clicking on an object will execute a default action associated with the class of the object. Double clicking on a histogram object will draw the histogram. Double clicking on an Ntuple quantity will produce a histogram showing the distribution of the quantity by looping over all tuples in the Ntuple. Right clicking on an object will bring up a context menu (just as in a canvas).

Integrating Your Own Classes into ROOT

In this section, I'll give a step-by-step method for integrating your own classes into ROOT. Once integrated, you can save instances of your class in a ROOT database, inspect objects at run-time, create and manipulate objects via the interpreter, generate HTML documentation, etc. A very simple class describing some person attributes is shown in Listing 1. The Person implementation file Person.cxx is shown in Listing 2.

The macros **ClassDef** and **ClassImp** provide some member functions that allow a class to access its interpreter dictionary information. Inheritance from the ROOT basic object, TObject, provides the interface to the database and inspection services.

Now run the **rootcint** program to create a dictionary, including the special I/O streamer and inspection methods for class Person:

```
bash$ rootcint -f dict.cxx -c Person.h
```

Next, compile and link the source of the class and the dictionary into a single shared library:

```
bash$ g++ -fPIC -I$ROOTSYS/include -c dict.cxx
bash$ g++ -fPIC -I$ROOTSYS/include -c Person.cxx
bash$ g++ -shared -o Person.so Person.o dict.o
```

Now start the ROOT interactive program and see how we can create and manipulate objects of class Person using the CINT C++ interpreter:

```
bash$ root
root [0] gSystem->Load("Person.so")
root [1] Person rdm(37, 181.0)
root [2] rdm.get_age()
(int)37
root [3] rdm.get_height()
(float)1.810000000000e+02
root [4] TFile db("test.root","new")
root [5] rdm.Write("rdm") // Write is inherited from the
TObject class
root [6] db.ls()
TFile** test.root
 TFile* test.root
 KEY: Person rdm;1
root [7] .q
```

Here, the key statement was the command to dynamically load the shared library containing the code of your class and the class dictionary.

In the next session, we access the **rdm** object we just stored on the database test.root:

```
bash$ root
root [0] gSystem->Load("Person.so")
root [1] TFile db("test.root")
root [2] rdm->get_age()
(int)37
root [3] rdm->Dump() // Dump is inherited from the TObject
class"
age     37      age of person
height  181     height of person
fUniqueID       0       object unique identifier
fBits   50331648        bit field status word
root [4] .class Person
[follows listing of full dictionary of class Person]
root [5] .q
```

A C++ macro that creates and stores 1000 persons in a database is shown in Listing 3. To execute this macro, do the following:

```
bash$ root
root [0] .x fill.C
root [1] .q
```

This method of storing objects would be used only for several thousands of objects. The special Tree object containers should be used to store many millions of objects of the same class.

Listing 4 is a C++ macro that queries the database and prints all persons in a certain age bracket. To execute this macro, do the following:

```
bash$ root
root [0] .x find.C(77,80)
age = 77, height = 10077.000000
age = 78, height = 10078.000000
age = 79, height = 10079.000000
age = 80, height = 10080.000000
NULL
root [1] find(888,895)
age = 888, height = 10888.000000
age = 889, height = 10889.000000
age = 890, height = 10890.000000
age = 891, height = 10891.000000
age = 892, height = 10892.000000
age = 893, height = 10893.000000
age = 894, height = 10894.000000
age = 895, height = 10895.000000
root [2] .q
```

With Person objects stored in a Tree, this kind of analysis can be done in a single command.

Finally, a small C++ macro that prints all methods defined in class Person using the information stored in the dictionary is shown in Listing 5. To execute this macro, type:

```
bash$ root
root [0] .x method.C
class Person Person(int a = 0, float h = 0)
int get_age()
float get_height()
void set_age(int a)
void set_height(float h)
const char* DeclFileName()
int DeclFileLine()
const char* ImplFileName()
int ImplFileLine()
Version_t Class_Version()
class TClass* Class()
void Dictionary()
class TClass* IsA()
void ShowMembers(class TMemberInspector& insp, char* parent)
void Streamer(class TBuffer& b)
class Person Person(class Person&)
void ~Person()
root [1] .q
```

The above examples prove the functionality that can be obtained when you integrate, with a few simple steps, your classes into the ROOT framework.

Linux an Increasing Force in Scientific Computing

Analyzing the FTP logs of the more than 9,300 downloads of the ROOT binaries reveals the popularity of the different computing platforms in the mainly scientific community. Figure 4 shows the number of ROOT binaries downloaded per platform.

Figure 4. ROOT Download Statistics

Linux is the clear leader, followed by the Microsoft platforms (Windows 95 and NT together equal Linux). The results for the other UNIX machines should probably be corrected a bit, since many machines are multi-user machines where a single download by a system manager will cover more than one user. Linux and Windows are typical single-user environments.

Summary

In this article I've given an overview of some of the main features of the ROOT data-handling system. However, many aspects and features of the system remain uncovered, such as the client/server classes (the TSocket, TServerSocket, TMonitor and TMessage classes), how to automatically generate HTML documentation (using the THtml class), remote database access (via the rootd daemon), advanced 3D graphics, etc. More on these topics can be found on the ROOT web site.

Resources

Acknowledgements

Fons Rademakers received a Ph.D. in particle physics from the University of Amsterdam. Since 1988 he has been working at CERN developing database, data analysis and graphics software. Fons started using Linux in 1993 and has been advocating it ever since. Besides developing ROOT, he is building several Linux PC farms for physics data processing (a joint project with Hewlett Packard). When not programming, he races go-carts and rides his trail bike. He can be reached via e-mail at Fons.Rademakers@cern.ch.

Rene Brun received a Ph.D. from the University of Clermont-Ferrand, France. He joined CERN in 1973. Rene made major contributions to the CERN Program Library, creating and coordinating the development of major software projects such as GEANT and PAW. In 1989, he received the IEEE/CANPS award for his contribution to a general detector simulation framework for nuclear and particle physics. He can be reached via e-mail at Rene.Brun@cern.ch.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

A Glimpse of Icon

Clinton Jeffery

Shamim Mohamed

Issue #51, July 1998

This article gives a quick introduction to the programming language Icon, developed at the University of Arizona.

Linux users are early adopters of new technology, so it's not surprising that many in the Linux community wish to use the best programming language for a given application, rather than being limited to just one language. The purpose of this article is to tell you about one of the simplest and most powerful programming languages available. It's called Icon, and it is a language for people who love programming. This tutorial is a “teaser” meant to pique your curiosity; the April 1998 issue of *Linux Gazette* has a longer tutorial which goes into more detail about the features described here.

My Programming Language Can Beat Up Your Programming Language

Languages are the subject of religious wars; very little is gained by arguments “proving” one language is better than another. Icon is not perfect, nor is it the “best” language—but it is a very nice language to use. Icon is for people who don't want to deal with memory management in C or C++; for people who want the power of Perl and beyond, but prefer a cleaner expression syntax and fewer special cases; and for people who have a use for rich data structures and algorithms, but take for granted all the programming building blocks they learned in school. Icon is used for children's games, scripture analysis, CGI scripts, compiler research, literate programming, system administration and visualization. It is in many ways what BASIC should be and what Perl and Java could have been. (If you know a language that allows simpler and more direct solutions to the three short, complete program examples given in this article, please tell us about it.)

Icon: Listing the Basics

Icon's basic philosophy is to make programming easy. Its syntax is similar to C or Pascal; programs are composed of procedures, starting from **main**. Icon's built-in list and table data types beat out most languages: other languages have similar types but just don't seem to do the operators and semantics as nicely. Both types use familiar subscript notation, hold values of any type and grow or shrink as needed. Lists take the place of arrays, stacks and queues. Tables associate keys of any type with corresponding values. These types are ingeniously implemented; for example, lists are like arrays when you use them like arrays, and like linked lists when you use them like linked lists.

Although Icon has some exotic concepts compared with C or FORTRAN, in several ways Icon programs are *more readable*, not just shorter. For example, when they are “true”, the relational operators return the value of the right operand, and associate left to right, so **(12 < x < 20)** tests whether x is between 12 and 20.

Here is a silly sample program that counts the number of occurrences of each word given on its command line and writes the words out in alphabetical order, along with their corresponding counts. A table is created with all keys mapping to a default value of 0. Then, each argument on the command line is used as a key in the table to increment a counter. The table is sorted, producing a list of two-element lists containing the keys and their values. These pairs are removed from the list one at a time, and the keys and values are written out.

```
procedure main(argv)
 T := table(0)
 every T[ !argv ] +:= 1
 L := sort(T)
 while pair := pop(L) do
 write(pair[1], ": ", pair[2])
end
```

The Joy of Generators

Generators are Icon's unique feature; they are its computer science research contribution. They give the language simpler, more intuitive notation, so they are worth making a mental leap. *Generators* can produce more than one value, and expression evaluation tries each value from a generator until it finds one that makes the enclosing expression succeed and produce a value. For example, **(2|3|5|7)** is a simple expression that produces the values 2, 3, 5 and 7; so the expression **(x = (2|3|5|7))** tests if the value of **x** is one of those four values.

In the previous program example, the expression **!argv** generated the elements from the list **argv**. Expression evaluation tries to obtain a value; the **every** control structure causes *all* the values to be produced. This code

```
every i := (1 to 10) | (20 to 30) do
 write(L[i])
```

prints the first ten values from the list, followed by elements 20 through 30.

Generators are a very natural way to write procedures that compute a sequence of values. In a language like C, the procedure has to maintain its state between calls using static data; in Icon, this is done automatically. Here's one way you might write a web-link checker:

```
every url := get_url(document) do
 test_url(url)
```

The procedure **get_url** scans the document for hyperlinks:

```
procedure get_url(filename)
 f := open(filename) |
 stop("Couldn't open ", filename)
 while line := read(f) do {
 ...
 url := ...
 suspend url
 }
end
```

In the above example, **get_url** is called only once. Each time a **suspend** occurs, a result is produced for the surrounding expression, and if the surrounding expression fails, the call is resumed where it left off, at the **suspend**. Generators are the basis for additional powerful language features (see *Linux Gazette* article for details).

Graphics and User Interfaces

Icon's built-in graphics have about 40 functions and introduce only one new type, the window, which is a special extension of the file type. This contrasts with graphics APIs in other languages where learning graphics means learning 400 or more functions that manipulate several dozen new types of values. Passing strings and integers into a few functions is all you need to write amazing graphics without excessive code.

One demonstration of Icon graphics is Brad Myers' “rectangle-follows-mouse” test, a program that opens up a window in which a rectangle follows a mouse around on the screen. A window is opened (file mode “g”) with an XOR raster drawing operation that causes graphics to erase themselves when redrawn. In the loop, for each user event, the ten-pixel square is erased and redrawn at the new mouse location. **&x** and **&y** are Icon keywords which hold the current mouse location and are saved in variables **x** and **y**. The variables **x** and **y** start out as null. The expression **\x** fails if **x** is null, causing the first call to DrawRectangle to be skipped the first time through the loop, since at this point, there is no rectangle to draw.

```
procedure main()
 w := open("win","g", "drawop=reverse")
 repeat {
 # get mouse/keyboard event
 Event(w)
 # erase old rectangle
 DrawRectangle(w, \x, y, 10, 10)
 # draw new rectangle
 DrawRectangle(w, x := &x, y := &y, 10, 10)
 }
end
```

Simple graphics programming is easy, but complex graphics are also possible. The Icon Program Library (IPL), a collection of Icon utilities and libraries, offers a more extensive Motif-style user interface toolkit as well as a WYSIWYG (what you see is what you get) interface builder that lets you build interfaces by drawing them. The IPL contains several other examples of graphical games and applications.

POSIX Made Simple

The Unicon flavor of Icon adds an elegant set of UNIX system-level facilities. An ultra-simple version of the **ls** utility illustrates some of these features. This version takes a directory name on the command line and produces a listing of file information including file size and modified time, sorted by name. (A more interesting version is included in *Linux Gazette* article.)

ls reads the directory and performs a **stat** call on each name it finds. In Icon, opening a directory is exactly the same as opening a file for reading; every *read* returns one file name.

```
$include "posix.icn"
procedure main(argv)
 f := open(argv[1]) |
 stop("ls: ", sys_errstr(&errno))
 names := list()
 while name := read(f) do
 push(names, name)
 every name := !sort(names) do {
 p := lstat(name)
 write(p.size, "   ", ctime(p.mtime)[5:17],
 " ", name)
 }
end
```

The **lstat** function returns a record with all the information that **lstat(2)** returns. In the Icon version, the **mode** field is given as a human readable string—not an integer to which you must apply bitwise magic. Also, in Icon, string manipulation is very natural.

Give Icon a try; whether you're a programmer or not, you'll love it.

Resources

Clint Jeffery is an assistant professor in the Division of Computer Science at the University of Texas at San Antonio. He writes and teaches about program monitoring and visualization, programming languages and software engineering. Contact him at jeffery@cs.utsa.edu or read about his research at www.cs.utsa.edu/faculty/jeffery.html. He received his Ph.D. from the University of Arizona.

Shamim Mohamed met UNIX in 1983 and was introduced to Linux at version 0.99 pl12. These days he is a Silicon Valley polymath and factotum, and an instrument-rated pilot flying taildraggers. He can be reached at spm@drones.com or www.drones.com/. He received his Ph.D. from the University of Arizona.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Advanced search

Having Fun on ViewSurf

Pierre Ficheux

Issue #51, July 1998

This article explains how Linux is used in the ViewSurf “Beach Report”, a fun WebCAM-based service.

The raison d'être of ViewSurf is to give surfers access to up-to-date on-line weather reports. Classic weather report information is provided, such as temperature and wave status, but the bonus is an up-to-date video that gives the surfer a current picture of what is happening at his favourite beach.

I met the creator of ViewSurf (Nicolas Saubade) during the summer of 1996. Nicolas works for COM1 in Cestas near Bordeaux, France. COM1 is a very famous company in France because it's the foremost modem manufacturer in Europe even though most COM1 modems are not distributed under its own label. Additionally, COM1 develops and distributes the ViewCOM, a high performance video compressing system used in many security applications (see Figure 1).

Figure 1. ViewCOM VM3

The ViewCOM uses a standard video input, such as a video camera or any PAL/SECAM/NTSC source, and converts this source to a proprietary format based on the JPEG compression algorithm. This format is called VCR, and the conversion can be achieved in real-time. The ViewCOM includes a V34 PC-Card modem, so it is typically installed on a foreign site and called by specialized software running under Microsoft Windows (ViewCOM Manager) via a simple phone line.

ViewCOM firmware includes a recording function to create a compressed video sequence and send it to the caller via modem. The size of each sequence is 100KB to 400KB and running time is 1 or 2 minutes of video.

Figure 2. ViewSurf basic configuration

The basic configuration of the ViewSurf service is quite simple (see Figure 2). Each site has a video camera connected to a ViewCOM. The ViewCOM is directly accessible via modem. The caller records a short film segment and uploads it to a web server. For the Beach Report, this operation occurs about three times a day. The browser on the client side must download a plug-in from the COM1 web site in order to display VCR sequences. This plug-in originally existed only for Microsoft Windows and Macintosh, so I wrote a UNIX/Linux version which is now on the COM1 site (available for Linux ELF, Solaris and SunOS).

Nicolas wanted to install several sites, but it was quite difficult to manage because the ViewCOM Manager, a nice graphical program, is not really programmable—the problem with most Windows applications. He had to manually call four sites, three times a day, seven days a week—not an acceptable situation.

I proposed to him that Linux be used to automate the process. I wrote some simple shell scripts to call each site, create and download the film and copy it to the main web server (an SGI Indy) using the **rcp** command. Most of these scripts are based on the **chat** program. The download portion was written in C to keep up with the high speed on the serial line (57,600 or 115,200Kbps).

I know rcp is not the best solution; Linux is a very good web server system in its own right, but the SGI was already in place. Film is integrated in ViewSurf pages with HTML code such as:

```
<EMBED SRC="http://your_linux_server/films/film.vcr"
WIDTH=320 HEIGHT=40>
```

Actually, using rcp requires no HTML modification in the existing pages, which is an advantage, so we opted to stay with it.

The main shell scripts, including dial up to a group of sites, are simply activated by a crontab entry. Additionally, these scripts give some statistics about ViewCOM access in order to detect any problems.

This project was not an official COM1 project, so the software was installed on a very old DX2/66 running Slackware 3.0. We had to buy a new 16550A-based ISA card for the serial line.

The ViewCOM Manager was no longer needed for ViewSurf. Nicolas was surprised by the power of Linux—all I had to do to solve a problem was write some shell scripts using standard Linux commands, which would have been very difficult and costly to implement in Windows. Some months later, Nicolas created a Snow Report, which is a service for skiing information comparable to the Beach Report. Last winter, 3 ViewCOMs were installed in the French Pyrénées mountains.

Nicolas has written some additional HTML pages in order to make the service more attractive, and ViewSurf now includes interesting links to fun sites and tourist WebCAMs all around the world. A specific domain now exists for ViewSurf (viewsurf.com), and the service is available (in French) at http://www.viewsurf.com/. Figure 3 is an example of a ViewSurf page. Don't forget to download the VCR plug-in.

Figure 3. Hossegor Beach (France)

Actually, the Linux PC is very efficient and robust. The last time I had to reboot it was to install a new kernel version.

The Future of ViewSurf

Beach Report and Snow Report are free services for the end user, but Nicolas created ViewSurf in the hope of making some money with it. He's currently trying to sell the service to French Tourism Offices, but it's quite hard; basically, France is lagging in communications and Internet services. Additionally, many French people consider computers and the Internet as American Trojan horses such as McDonald's or Disneyland Paris.

Most French on-line services are available for a low performance Videotex-compatible terminal called Minitel, which was distributed free of charge by France Telecom at the beginning of the 1980s. This technology is obsolete, but France Telecom is currently the only French operator for communications. The Minitel allows them to charge up to several dollars per minute for some on-line services. This could be the reason why most French people don't have a PC at home, and as a result, Internet-based services are not seriously considered.

Nicolas has gotten a contract with the government organization which deals with traffic regulation in Paris. Some French highways have been on the Net since September 1997. If you compare it with other WebCAM systems, ViewSurf gives very good quality for a small data size.

This software would be more easily configurable without editing crontab or shell scripts each time you wished to change the call time or add a new site. To that end, I wrote a set of CGI (Common Gateway Interface) scripts which present a simple and portable interface for the Linux server configuration. The advantage of using CGI instead of standard Linux programs is the capability to configure the server from any forms-capable browser running on any operating system.

Another crucial option is to have the ability for several users to look at a “live” video (not recorded files) at the same time. For this, the Linux PC could be used as a server to distribute the live image from ViewCOM to several users connected from the Net. To reach this goal, I wrote a multi-threaded Linux daemon, based on the POSIX 1003.1c LinuxThreads library by Xavier Leroy (http://pauillac.inria.fr/~xleroy/linuxthreads). Actually, this daemon handles only the “video/x-vcr” MIME type and uses two specific TCP ports. The live video can be inserted in an HTML page with a line such as:

```
<EMBED SRC="http://your_linux_server:daemon_port"
WIDTH=320 HEIGHT=240>
```

The second port is reserved for ViewCOM administration, such as setting brightness or contrast. Additionally, the daemon can control a weather station in order to get real-time information about external temperature, wind speed and other weather information. A VISCA (a standard for video camera remote control) functionality is about to be added to control zoom, pan-and-tilt and other camera parameters directly from the Internet browser. Figure 4 is a snapshot of the Bordeaux/Bayonne motorway on the private COM1 web server.

Figure 4. Snapshot of Bordeaux/Bayonne Motorway

The ViewCOM is often connected to the PC via a serial line, but one of the most important advantages of the system could be the ability to control a remote ViewCOM. So, it's not necessary to install a PC on the site you want to look at, you just have to set up a ViewCOM connected to a simple phone line or a leased line. In the phone line case, it's possible for the daemon to call the ViewCOM at starting time or only when an HTTP request occurs. In this last case, the daemon hangs up the line when the last client is disconnected.

References and Contact

Pierre Ficheux is in charge of system development at Lectra-Systèmes, Cestas, France. When not doing something with Linux, he loves picking tunes on his guitar on the nice beach at Arcachon. He can be reached by e-mail at pierre@rd.lectra.fr.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Encrypted File Systems

Bear Giles

Issue #51, July 1998

Here's a good way to protect your files. Mr. Giles explains how to encrypt your entire file system rather than individual files.

In one episode of “Miami Vice” Crockett and Tubbs have managed to gain access to a drug runner's computer, only to be stymied by its insistence on a password before presenting incriminating evidence. Not to worry—after only three unsuccessful guesses, the helpful computer offered to reveal the secret password to our heroes. It's easy to laugh at this plot development, but many otherwise intelligent people continue to do equally dumb things.

Consider the law office where legal papers are always kept in locked cabinets behind locked doors. Every computer on the LAN also has access to the “password-protected” word processing documents, but the encryption can be broken in seconds with readily available software. The name of this program, and the files it can crack, are in the sci.crypt FAQ. These files could be retrieved by a hostile agent “working” for a cleaning contractor.

Or consider the company with sales offices spread nationwide. Highly sensitive pricing and contact information is distributed on CD-ROM discs, which are discarded as soon as each new disc arrives. Alternately, a salesman may have his laptop stolen while on the road. (See *Practical UNIX and Internet Security*, Garfinkel and Spafford, O'Reilly and Associates, 1996.)

Or consider the individual computer owner who leaves his system in a shop for free installation of an upgrade. One of the employees quietly copies a few files, and by the time the victim learns of the extent of the identity theft it's too late—he's already recommended the same shop to several of his friends for the unusually good service.

Solution: File Encryption

> For every complex problem there is an answer that is clear, simple and wrong. --H. L. Mencken

The simple solution to these problems is file encryption. But this solution is flawed for several reasons:

- Encryption *within* programs is generally weak to the point of uselessness due to U.S. export regulations.
- Encryption *outside* programs requires explicit actions to decrypt and to re-encrypt. This problem may be manageable if a file needs to be accessed only by a single user, but it's a much more difficult problem if several people need to share access.
- Explicit encryption requires sharing the password, and the more people who have the password, the more likely it becomes that someone will jot it down in an obvious location.
- Explicit encryption may enable a disgruntled employee to encrypt the files with a different password.
- Decrypting a file increases the risk that unencrypted versions will remain on the disk or on backup media.

Even with its flaws, file encryption may still be better than the alternatives. Fortunately there is a better solution.

Solution: File System Encryption

Our solution is to encrypt the entire file system. User programs see a regular file system—perhaps even a file system that natively supports encryption. An attacker who can only see the physical disk sees garble.

This approach is not perfect. Most notably, some implementations could leave decrypted data visible in the disk cache. That is a minor problem with the cache in core (if an attacker has compromised root, you have more serious problems), but a major problem if these pages get written to swap.

On the other hand, the kernel ensures that disk sectors are decrypted during reads and re-encrypted during writes. The impact on users is minimal. In one practical scenario, a "responsible individual" will mount the encrypted file system in the morning. (This requires the encryption key.) In the evening, the last person to leave could unmount the file system, or it could be automatically unmounted by a `cron` job.

Encryption Algorithms

> Better the devil we know... --Anonymous

We've agreed on the desirability of encrypting file system. But which encryption algorithm should we use? The wrong choice will leave us with a false sense of security.

Writing our own encryption routines is one possibility. The downside is encryption algorithms are notoriously difficult to properly design and implement. The problem is that the designer does not know what others will find difficult. He only knows what *he* finds difficult. Mathematics is littered with the bodies of “difficult” problems which became trivial after one person had a flash of insight.

As a practical matter, we should limit our search to well-known encryption algorithms. This has the additional benefit of allowing us to share encrypted file systems with others with a minimum amount of hassle.

XOR

The first encryption algorithm learned by most programmers is the lowly **xor** algorithm. To encrypt the data, we XOR it with the key (modulo the length of the key, if we use multi-byte encryption). To decrypt the data, we XOR it with the key again.

- Benefits: fast and exportable
- Drawback: trivial to break
- Synopsis: stops casual snooper

DES

DES has a controversial past. It was a government-endorsed algorithm for non-classified use, but some people believe that the government deliberately introduced weaknesses. On the other hand, decades of research have revealed only relatively modest weaknesses. It is economically feasible for a large company to build a DES-cracking machine.

- Benefits: strong, well-tested, 56-bit keys (The variant known as TRIPLE-DES uses 112-bit keys.)
- Drawback: not exportable
- Synopsis: a reasonable choice

IDEA

DES was designed for hardware implementations—and is difficult to implement efficiently in software. IDEA was designed around the low-level operations common on small processors. It is not a U.S. federal standard and wasn't weakened by the dreaded TLAs (three letter acronyms, such as DEC and FBI). On the other hand, while the TLAs have undoubtedly analyzed it, they aren't talking.

- Benefits: strong, tested, 64-bit keys (used internally by PGP)
- Drawback: not exportable
- Synopsis: a reasonable choice

RSA

RSA encryption is a relatively ineffective algorithm. Many people feel that the primary weakness with PGP lies in the 1024-bit RSA encryption of the IDEA key, not the IDEA encryption of the actual data.

- Benefits: solution to public key encryption problem, 128-bit keys
- Drawbacks: requires at least 1024 bits for security comparable to IDEA, very slow
- Synopsis: not appropriate

Obtaining the Source: Cypherpunks

> Fools rush in where angels fear to tread. --Alexander Pope

Undoubtedly, some people now feel the urge to run out and write an encrypting file system. The rest of us turn to the *Cypherpunks*. They have published a set of patches to the 2.0.11 kernel which implement DES and IDEA encryption in “loopback” devices. The primary source for these patches is at: ftp://ftp.csua.berkeley.edu/pub/cypherpunks/filesystems/linux.

There are four patches:

1. loopfix-2.0.11.patch: modifications to loopback device
2. export-2.0.11.patch: more patches, mostly to documentation and the makefile
3. crypto-2.0.11.patch: export-restricted patches: DES and IDEA
4. mount-2.5k.patch: modification to **mount** to pass encryption keys.

The U.S. government continues to interpret the International Traffic in Arms Regulation (ITAR) in a manner that prohibits the export of meaningful cryptographic software via electronic means. There are no restrictions on the export of the same material in printed form or its subsequent distribution from sites outside North America.

The source code in crypto-2.0.11.patch implements DES and IDEA encryption and cannot be legally exported, even though this source is readily available worldwide. Violating export restrictions will not aid the effort to promote the free use of strong encryption, since the government could use this as proof of the need for stronger restrictions on *domestic* distribution.

Building the Kernel

Building the new kernel is no different than applying any other set of patches. The latest stable kernel release for which this works is 2.0.30. For convenience, I will assume it is stored in /usr/src/linux-2.0.30.tar.gz. Next, build a reference version of the kernel. Then, follow these steps:

1. Get the latest encrypted file system patches. For convenience, I will assume that they are the 2.0.11 patches and stored in /usr/src/cryptfs.
2. Apply the patches to the kernel, retaining the reference copy. On my system, this involved making a working directory, and applying the patches and fixing problems. I made the working directory by issuing the following commands:

   ```
   cd /usr/src
   rm linux
   tar xzpf linux-2.0.30.tar.gz
   mv linux linux-2.0.30.efs
   ln -s linux-2.0.30.efs linux
   ```

   I applied the patches using these commands:

   ```
   cd linux
   patch < ./cryptfs/export-2.0.11.patch
   patch < ./cryptfs/loopfix-2.0.11.patch
   patch < ./cryptfs/crypto-2.0.11.patch
   ```

   I fixed problems using these commands:

   ```
   mv *.h linux/include/linux
   mv des.c linux/kernel
   mv idea.c linux/drivers/block
   mv loopfix.txt linux/Documentation
   ```
3. Configure and build the new kernel. Remember to enable the loopback device and file system encryption.
4. Get the source for **mount** and apply the required patch. Build it.
5. Reboot the system with your new kernel.

At this point everything should be ready to go, but I've encountered problems after builds. I believe my problem was caused by improper application of the patches, perhaps due to order-based instabilities caused by changes between the 2.0.11 and 2.0.30 and above kernels. One recurrent problem occurred with the **urandom** command:

```
od -x /dev/urandom | more
```

Giving this command produced kernel warning messages. If this happens to you, reinstall the kernel source and patches and check your warnings carefully.

Encrypted File Systems: Ready, Set, Go!

Find a couple of blank floppies on which to test an encrypted file system. Then, create an encrypted file system using DES encryption:

```
# dd if=/dev/urandom of=/dev/fd0 bs=1k count=1440
# losetup -e des /dev/loop0 /dev/fd0
Pass phrase: des test
# mke2fs /dev/loop0
# losetup -d /dev/loop0
```

A couple of notes about this example:

- The first command initializes the floppy disk with *random* data. Initializing the disk to zeroed data reduces a blank disk to a “known plaintext” cryptology problem—not a good idea.
- The second command specifies that we want a loopback device to cover the floppy device driver with a DES encryption layer. We could replace /dev/fd0 with the name of a file. The pass phrase is not echoed. Also, the pass phrase can be 120 characters long—and should definitely be more than 8 characters!
- The third command is the normal **mkfs(1)** utility.
- The fourth command releases the loopback device.

We also want to create an encrypted file system using IDEA; the same idea, only replace **des** with **idea**.

Finally, create one more pair of disks which use different passwords. (If you want to be unusually perverse, use your previous IDEA test pass phrase on your second DES test disk and vice versa.)

Now we're ready to mount these disks. First, try to mount the floppies using a standard mount command:

```
# mount /dev/fd0 /mnt -text2
```

These commands should fail with “can't find an EXT2 file system.” Now try mounting each floppy again:

```
   # mount /dev/fd0 /mnt -text2,loop,encryption=idea
   # mount /dev/fd0 /mnt -text2,loop,encryption=des
```

In each case you should be prompted for a pass phrase. Needless to say, you should not be able to mount the DES encrypted disk when specifying IDEA encryption, and vice versa. Likewise, you should not be able to mount the DES encrypted disk 1 with the second password or vice versa, and you should be able to mount the file system when you specify the correct encryption format and password.

This is another area where gremlins have appeared on my system. Once IDEA encryption worked fine but /dev/urandom failed; in another case, /dev/urandom worked but IDEA encryption produced kernel warnings on every even sector.

Now a few more tests. Edit the /etc/fstab file to add these entries:

```
   /dev/fd0 /mnt/des  ext2
           defaults,noauto,loop,encryption=des  0 0
   /dev/fd0 /mnt/idea ext2
           defaults,noauto,loop,encryption=idea 0 0
```

Try to mount your test disks on /mnt/des and /mnt/idea. Once again you should be prompted for a pass phrase and will be successful only when encryption algorithm and pass phrases match.

Finally, reboot your system and repeat these tests. If possible, install the modified kernel on a second system and verify that you can exchange media between the systems. Such is life on the bleeding edge of technology.

Applications

Now that we have encrypted file systems, what can we do?

- We can add strong encryption to programs which don't support them natively, and we keep their files on an encrypted file system.
- We can add strong encryption to distributed media. Some people already build ISO-9660 images in a file via a loopback device; producing an encrypted image would be trivial.
- CD-ROM-based back-up protocols become more attractive. Outdated back-up discs can be discarded without fear of a dumpster diver gaining access to crucial information.
- We can improve system security. Programs such as **Tripwire**, which record cryptographic signatures of key files, traditionally require read-only media to prevent attackers from modifying the reference information. It is still conceivable that an inside attacker could replace this critical disk. Now, we

can easily keep this crucial information in an encrypted form, making a spoofed disk much harder to produce.
- We can add a measure of strong encryption to entire systems which don't support them natively. Encrypted file systems should be exportable via NFS or SMB—packet sniffers remain a problem but the disk would be protected.

Long-Term Applications

Even taking a cursory glance at the trends of security software, one notices recurring themes. Encrypted file systems protect the data on disks. SSH (Secure Shell) encrypts and authenticates communications. Secure-RPC (remote procedure call) encrypts interprocess communications. RPM authenticates software upgrades.

Is there any question that encryption and authentication routines belong in the kernel? Encryption keys could be stored with each device and process, and with negotiations for unique session keys automatically mediated between any process within and without the system that desired it. There would not be needless duplication of identical routines, or worries about export restrictions since these issues would have already been addressed. If necessary the encryption routines could be localized to a loadable module, although that raises certain security issues.

The downside is anyone with root access can grab the encryption keys from the system tables; however, once root is compromised all bets are off anyway. On the other hand, supplying strong encryption and authentication services in the kernel should reduce the risk of root becoming compromised. Also, DH key negotiation means that my keys aren't compromised even if I'm talking to someone who is compromised.

Bear Giles bear@coyotesong.com, of Coyote Song LLC, is a UNIX Consultant with almost 15 years of experience. He has used Linux at home since pre-0.99 days.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Graphical Desktop Korn Shell

George Kraft IV

Issue #51, July 1998

The Graphical Desktop Korn Shell (DtKsh) is a featured part of the Common Desktop Environment (CDE). DtKsh provides a consistent and reliable graphical Motif shell language that is supported on all CDE-compliant systems.

Portability and pervasiveness are two important characteristics to consider when you are developing code. Using a programming language with a well-defined and stable application programming interface (API) answers the need for portability. A programming language with a large, established installed base provides pervasiveness. Although Perl, Tcl/Tk, Common Gateway Interface (CGI) and Java have large installed bases, they are not suited for some projects. The reason for this is their inconsistent installation base due to the lack of a well-defined or rapidly changing API.

The Desktop Korn Shell (DtKsh) that comes with the Common Desktop Environment (CDE) is built on the ksh93 standard with X, Xt, Motif, ToolTalk and CDE built-in APIs. Unlike Perl and Tcl/Tk, major vendors have built and supported DtKsh through the CDE initiative. Using DtKsh, desktop programmers can develop and/or prototype plug-and-play Graphical User Interface (GUI) applications that are compatible on all CDE-compliant systems without compilation. Although DtKsh applications are interpreted for portability, they can easily be migrated to Motif in C for performance.

Tcl/Tk can be ported to C with the aid of special Tcl/Tk libraries; however, programmers are as disadvantaged with the C Tcl/Tk libraries as they are with the Tcl/Tk shell, because of a not-so-standard application programming interface. DtKsh, unlike Tcl/Tk, provides a well-established API set where the programmer's knowledge transcends from C to shell programming.

DtKsh Benefits

In AIX, /bin/ksh is an XPG4-compliant version of ksh88. CDE's */usr/dt/bin/dtksh* on AIX is based on the newer ksh93 standard. ksh93 now includes floating-point mathematics, associative arrays, new string operations, hierarchical variables, reference variables, developer-extendable APIs using attached shared libraries and character class patterns.

Floating-point mathematics: Korn Shell variables can be cast, or defined, to various aggregate data types. Floating-point mathematics is a new feature in the Korn Shell that enables the assignment and operation of decimal values. The following example defines the floating-point variable **PI**, then assigns to it the decimal value of 3.14159.

```
typeset -F PI # define "PI" as a float
PI = 3.14159
```

Associative arrays: Instead of using positive integer indices, associative arrays allow elements of an array to be addressed using alphanumeric strings. The following example shows **SYSINFO** as an array containing information about an operating system. The associative **SYSINFO** array can be indexed with the alphanumeric string of **"os"** to find the string value of AIX.

```
typeset -A SYSINFO # define "SYSINFO" as an
# associative array
SYSINFO["os"]=AIX
```

New string operations: Six new string operations were introduced in ksh93. These new operations provide substringing and substitution of a string pattern with an alternate. Substringing permits extraction of a smaller string, given an offset indicating where to begin and possibly its length.

- A substring of a larger string can be extracted by length at a given starting point, or a substring can be taken by starting at the offset within the larger string and stopping at the end of the string. The following shows a substring of a given length:

  ```
  ${variable:offset:length}
  ```

- A substring of no particular length can be taken by just providing the offset.

  ```
  ${variable:offset}
  ```

  String substitution of a character pattern can be performed for the first occurrence, a repeated occurrence, at the beginning of the string (prefix) or at the end of the string (suffix).

- Substitute the first occurrence of a pattern with an alternate string:

  ```
  ${variable/pattern/string}
  ```

- Substitute all occurrences of a pattern with an alternate string:

  ```
  ${variable//pattern/string}
  ```

- Substitute the pattern prefix with the alternate string:

  ```
  ${variable/#pattern/string}
  ```

- Substitute the pattern suffix with the alternate string:

  ```
  ${variable/%pattern/string}
  ```

Hierarchical variables: Hierarchical variables, or compound names, enable C structure-like aggregate data types. This allows Korn Shell to store information in variables in an associative fashion. For example, if we had a box with a width of 80 and a height of 24, then we could store all that information in one hierarchical variable instead of separate and disjointed variables of storage. Each element of the compound name must be used before setting sub-members.

```
BOX= # declare before assigning sub-members
BOX.WIDTH = 80
BOX.HEIGHT = 24
```

Reference variables: Referencing allows a variable to point to the same value as another variable; both variables reference the same value as shown below:

```
# name reference
typeset -n FOO=BAR
FOO="Hello World"
# print "Hello World"
print ${BAR}
```

Desktop built-in commands: Korn Shell provides some standard X, Xt, Motif, POSIX internationalization and CDE C language APIs directly built into the shell. Direct access to these APIs from the shell provides a significant runtime performance improvement for DtKsh shell applications. Using the standard X and Motif APIs, with some semantic changes to the source, makes it possible for DtKsh shell scripts to be migrated to C and compiled.

POSIX internationalization: Korn Shell provides the shell equivalent of the C language POSIX internationalization APIs **catopen** and **catgets**. The internationalization APIs allow the shell program to change its message catalog depending on its language. Internationalized shell scripts enable multilingual support.

Character class patterns: Regular expressions in the shell are enhanced by predefining a set of character class patterns. Now we can easily match certain classes of characters by using the [[:*class*:]] notation where *class* can be specified as alnu, alpha, cntrl, digit, graph, lower, upper, print, punct, space and xdigit.

```
# only print files that
# begin in upper case
print  [[:upper:]]*
```

```
# old way
print [A-Z]*
```

DtKsh “Hello World” Source

The familiar “Hello World” Motif application, shown in Listing 1, is actually written in DtKsh instead of C. Similar to C, we initialize the top-level shell widget, then start building the GUI application. Listing 1 shows a standard Motif message dialog using the familiar XmCreateMessageDialog API. In DtKsh, handles to widgets can be retrieved, widgets can be managed and unmanaged, and callbacks can be created. Afterwards, the program enters into the Xt Intrinsic's main loop via XtMainLoop where it processes X protocol events. In this case, clicking on the OK button would be an event processed by the event loop.

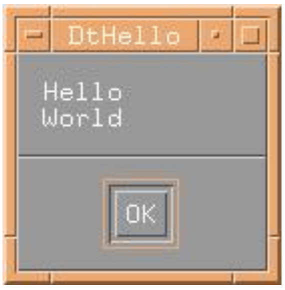


Figure 1. DtKsh and Motif “Hello World”

The Motif “Hello World” DtKsh application in Listing 1 can be easily ported to C with a few minor changes, shown in Listing 2. By adding some include files, defining some variables, adding some commas and semi-colons, and sprucing up some arguments, we have a C program. The result is that DtKsh shell scripts make the same API calls as the C Motif application.

AIX provides some extra DtKsh help through a GUI builder. Developers can drag and drop widgets onto a canvas, then add logic code to enable the application to do some work. Like any GUI builder, the code is somewhat verbose; however, it is consistent and portable. AIX is the only version of UNIX that offers this feature.

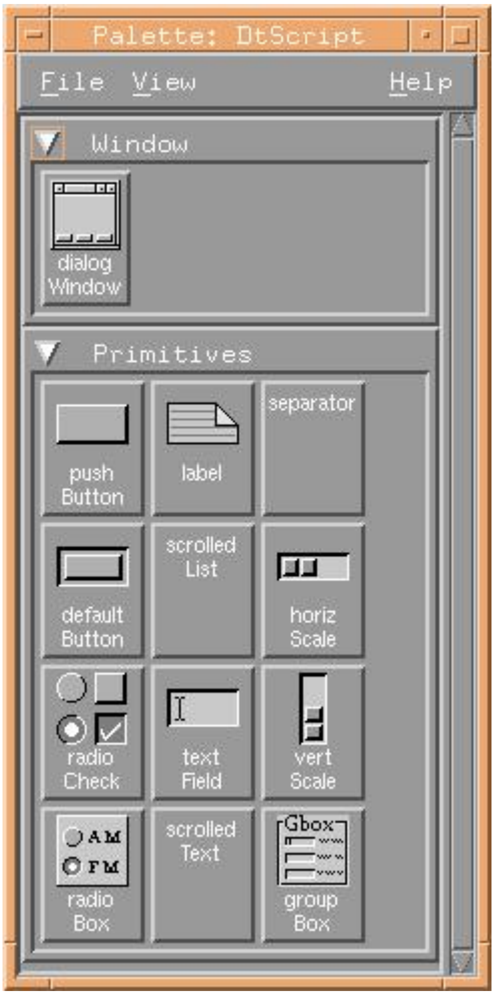


Figure 2. AIX QtScript GUI Builder

User Extendable

Developers can create their own new APIs for DtKsh by creating glue-layer libraries. Glue-layer libraries enable DtKsh to be extended with built-ins for functions such as system management and networking. The performance advantage of using built-in functions rather than calling to an external command is that built-ins execute within the process of the shell script. Commands that are called externally must create new resources in the operating system and run as separate processes. DtKsh glue layer libraries pass arguments between a normal UNIX C library and the DtKsh shell, and they return a success or failure status. The following list provides a few rules for creating a glue layer:

- Name the function with a **b_** prefix.
- Function returns a 0 integer for success, between 1 and 255 for failure.
- Function should take *argc* and *argv* as input.
- Link your glue-layer libraries shared.

Listing 3 shows a DtKsh shell script that dynamically loads the “example” shared glue-layer library. Once the glue layer library is loaded and the new built-in APIs are defined, the script can make direct calls with arguments to the new built-in functions. In Listing 3, the example built-in is called with the “Hello World” arguments.

By providing in-line built-in functions, we can run scripts much faster because we are not relying on outside programs running as separate system erocesses. Listing 4 shows the C glue-layer for the example built-in shared library. Following the rules outlined above, we prefix the example function with a **b_**, and we pass in an argument vector and its size. After the function has done its work, we return 0 for success and a positive integer for failure. DtKsh built-in functions can also act as procedures that pass environment variables in and out through its argument list. See *Desktop KornShell Graphical Programming* by J. Stephen Pendergrast, Jr. [Addison-Wesley, 1995] for more details on how to pass and retrieve environment variables from built-in procedures.

Conclusion

The Desktop Graphical Korn Shell provides programmers with the standard ksh93 baseline APIs with the addition of the X Window System, Motif and the Common Desktop Environment. Shell programmers can write portable shell scripts, prototype GUI shell scripts and migrate GUI shell scripts to faster running C programs. DtKsh also provides programmers with the ability to extend the shell language with built-in shared libraries so that scripts can benefit from feature-rich libraries, such as those for configuration management.

The Advantages of DtKsh

Acknowledgements

George Kraft is an Advisory Software Engineer for IBM's Network Computer Division. He has previously worked on CDE V2.1 and V1.0 for IBM's RS/6000 Division and on X and Motif for Texas Instruments' Computer Systems Division. He has a BS in Computer Science and Mathematics from Purdue University. He can be reached via e-mail at gk4@austin.ibm.com.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

A SCSI Test Tool for Linux

Pete Popov

Issue #51, July 1998

Mr. Popov shows us how easy it is to test SCSI devices when our operating system is Linux.

A few months ago my ex-boss and I were discussing the latest product of his company and the systems they were using for testing purposes, both in the office and in the factory. “It's all DOS—what you want is cheap,” he told me. I believe that statement summarizes the reason DOS became the operating system of choice for factories and many laboratories. A fully equipped SCSI DOS system needs little memory, a cheap video card and monitor and a relatively cheap SCSI controller. Of course, these days there are many other reasons for using DOS for testing SCSI peripheral devices. There is a vast knowledge base, availability of ASPI-based tools and, most likely, the company already has a hefty investment in test software or in developing its own tests. (ASPI stands for Advanced SCSI Protocol Interface and was developed by Adaptec. It is the de facto SCSI programming interface on DOS and Windows.) Nevertheless, Linux is also a viable SCSI test system, and there are some very good reasons why you should consider Linux for testing your SCSI devices.

Testing a SCSI sequential access device is not a simple matter. These devices (DDS and AIT tape drives, to be specific) are usually used in a server environment where system downtime is not acceptable, especially when the down-time is due to the data backup device hanging the SCSI bus. (DDS, digital data storage, is Sony's 4mm tape drive technology. AIT, advanced intelligent tape, is Sony's new 8mm tape drive technology.) To put such a product on the market, extensive testing needs to be done in quite a few areas. Currently, the testing is performed on PCs running Sony internal tests or Independent Software Vendor (ISV) backup software or OEM proprietary systems running a flavor of UNIX, NT or some other proprietary OS. The qualification cycle of these devices is much longer than that of hard disks, but so is the life cycle of the product. The more systems you can test on and the more types of tests you can run, the more solid your product will be.

Some portions of the SCSI protocol cannot be tested without using a specialized tool, such as an Itech SCSI emulator. This type of tool allows the engineer to develop powerful low-level SCSI tests to test the SCSI protocol handling. For example, with such a tool you might issue a write command, send one or more bytes to the drive, raise ATN (attention control signal) and then, when the drive goes to output data, send an abort or reset message. These tools, however, are quite expensive (in excess of $5000 US) and certainly not all engineers need them, since not all firmware engineers work on the SCSI protocol. By the time the product makes its way to the test lab, most SCSI protocol problems are solved so the lab needs only one of those specialized testers, mainly for use in firmware regression testing.

Yet, with all of the specialized tests we have, some of our most important testing is done on commercial UNIX systems, surprisingly enough with well-written C shell scripts using standard UNIX utilities such as **dd**, **tar** and **mt**. My guess would be that at least half of the firmware problems in the past have been found running these scripts on UNIX systems. Linux has all those utilities, as well as a selection of shells. The same shell scripts running on commercial UNIX systems can easily be ported to Linux, maintaining the same functionality. This means you can supplement the expensive commercial UNIX systems in your lab with a few low-cost PCs running Linux. This alone makes Linux a serious contender in the SCSI testing field.

Walk into a lab where SCSI devices are being tested, and attached to the PCs on a single SCSI bus, you'll usually see a few devices. It's certainly important to test more than one device at a time; however, it rarely makes sense to have more than three or four devices running under the same test. Even if you fully populate the SCSI bus, if one device hangs overnight it sometimes hangs the SCSI bus, not allowing the rest of the devices to continue. It makes no difference, in this case, that you have six other devices attached. What would be better is to attach a second or even third SCSI controller and run different types of tests on each one. That way you can utilize the system more efficiently and thereby get more done in the same amount of time. Furthermore, if one test finds a firmware bug and the bus hangs, the other tests can continue. DOS, however, is not a multitasking operating system, and if you wish to run different tests on each SCSI controller, you'll have to add all that complexity to your test. Well, it's really no wonder I've never seen that done. It's tough enough debugging complex SCSI tests, not knowing whether the failure was due to the test or the device. Adding additional complexity to the test will most certainly take away from the firmware engineer's time who will have to debug what may turn out to be a test problem, not a drive firmware problem. Linux, on the other hand, does not suffer from this limitation. You can write basic tests, add them to your shell script and let the operating system worry about the multitasking. If one test fails because of a hung device, the other tests can continue running.

The standard UNIX utilities provide a high level of functionality testing, but, to complete a test suite, the engineer needs finer control of the SCSI device. The ability to send any SCSI command, including commands with illegal bits set, as well as illegal commands, is a must. This is one area where standard UNIX utilities cannot do the job and an alternative method is needed. Some time ago, I decided it would be nice to have a library of SCSI commands that made it easy to write tests, as well as to expand the library itself. So I started playing with the Linux generic SCSI driver, which seemed the easiest way to go, and I recently released such a library under the GPL. libdat.a contains just about all the sequential-access SCSI commands and, if there is something else you need, adding new commands is quite trivial. The library is packaged with a tape tool called **stt**, SCSI Tape Utility, which is based on libdat.a. **stt** adds a powerful capability to my Linux workstation at the office. I can now interactively send any command to the tape drives, as well as reprogram drives and make reprogramming firmware tapes. (These last two features are removed from the GPL version.) It is also an example of how easy it is to write SCSI tests using libdat.a and the generic driver in general. Most importantly, I now find it easier to write tests for my Linux workstation than for proprietary tools. Here's an example of a short C program (the #includes are not shown):

```
  _Inquiry();      /* show device information */
  _Space(EOD,0);   /* space to End Of Data */
  _ReadPosition(); /* show current logical
                    * position */
  _Space(FMK, -2); /* space reverse 2 filemarks */
```

While the above program doesn't do much, it does show the ease with which the programmer can write tests. The stt utility provides a longer example of a fully functional and useful program based on libdat.a.

You may be happy with your current test setup, but consider the following questions. Could you do more if your OS was more capable? What if you could write C programs and shell scripts, instead of DOS batch files? What if your test system was fully networked? Could you run the log files through a Perl filter to format them and display them on your internal web site? Could you benefit from the standard UNIX utilities, which you don't have to rewrite? Certainly you could benefit from attaching more than one controller to your system and running more than one type of test at the same time while your OS took care of the multitasking. What if there was a generic, easy-to-use SCSI interface and library that gave you full control of your SCSI devices as well as access to all source code? What if you could do your development on a platform with a rich set of development tools, including compilers, debuggers, version control systems, etc? Next time you are considering a platform for your SCSI testing, look at the answers to these questions and do yourself a favor. Consider how much more you could do, if that platform was running Linux.

Resources

Pete Popov is a firmware engineer at Sony's Advanced Storage Development Division in San Jose, CA, currently working on Sony's AIT-2 tape drive. The director of his division is still skeptical of Linux, but he just made the huge leap from a Macintosh to a PC, so he needs a little more time to come around. Pete can be contacted at ppopov@redcreek.com.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

LINUX JOURNAL

Advanced search

Introducing Samba

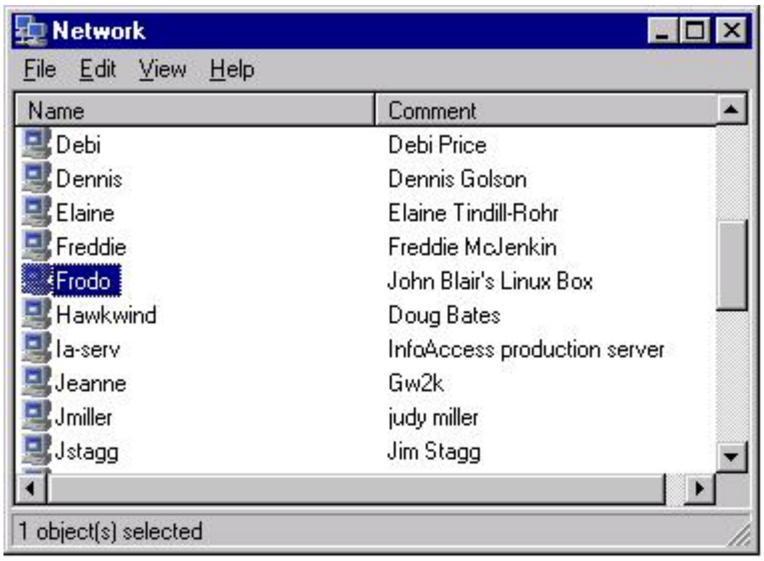
John Blair

Issue #51, July 1998

When you need to network your Linux box with Windows, Samba is the way to do it.

The whole point of networking is to allow computers to easily share information. Sharing information with other Linux boxes, or any UNIX host, is easy—tools such as FTP and NFS are readily available and frequently set up easily “out of the box”. Unfortunately, even the most die-hard Linux fanatic has to admit the operating system most of the PCs in the world are running is one of the various types of Windows. Unless you use your Linux box in a particularly isolated environment, you will almost certainly need to exchange information with machines running Windows. Assuming you're not planning on moving all of your files using floppy disks, the tool you need is Samba.

Samba is a suite of programs that gives your Linux box the ability to speak SMB (Server Message Block). SMB is the protocol used to implement file sharing and printer services between computers running OS/2, Windows NT, Windows 95 and Windows for Workgroups. The protocol is analogous to a combination of NFS (Network File System), lpd (the standard UNIX printer server) and a distributed authentication framework such as NIS or Kerberos. If you are familiar with Netatalk, Samba does for Windows what Netatalk does for the Macintosh. While running the Samba server programs, your Linux box appears in the “Network Neighborhood” as if it were just another Windows machine. Users of Windows machines can “log into” your Linux server and, depending on the rights they are granted, copy files to and from parts of the UNIX file system, submit print jobs and even send you WinPopup messages. If you use your Linux box in an environment that consists almost completely of Windows NT and Windows 95 machines, Samba is an invaluable tool.



Name	Comment
Debi	Debi Price
Dennis	Dennis Golson
Elaine	Elaine Tindill-Rohr
Freddie	Freddie McJenkin
Frodo	John Blair's Linux Box
Hawkwind	Doug Bates
Ia-serv	InfoAccess production server
Jeanne	Gw2k
Jmiller	judy miller
Jstagg	Jim Stagg

1 object(s) selected

Figure 1. The Network Neighborhood, Showing the Samba Server

Samba also has the ability to do things that normally require the Windows NT Server to act as a WINS server and process “network logons” from Windows 95 machines. A PAM module derived from Samba code allows you to authenticate UNIX logins using a Windows NT Server. A current Samba project seeks to reverse engineer the proprietary Windows NT domain-controller protocol and re-implement it as a component of Samba. This code, while still very experimental, can already successfully process a logon request from a Windows NT Workstation computer. It shouldn't be long before it will act as a full-fledged Primary Domain Controller (PDC), storing user account information and establishing trust relationships with other NT domains. Best of all, Samba is freely available under the GNU public license, just as Linux is. In many environments the Windows NT Server is required only to provide file services, printer spools and access control to a collection of Windows 95 machines. The combination of Linux and Samba provides a powerful low-cost alternative to the typical Microsoft solution.

Windows Networking

Understanding how Samba does its job is easier if you know a little about how Windows networking works. Windows clients use file and printer resources on a server by transmitting “Server Message Block” over a NetBIOS session. NetBIOS was originally developed by IBM to define a networking interface for software running on MS-DOS or PC-DOS. It defines a set of networking services and the software interface for accessing those services, but does not specify the actual protocol used to move bits on the network.

Three major flavors of NetBIOS have emerged since it was first implemented, each differing in the transport protocol used. The original implementation was referred to as NetBEUI (NetBIOS Extended User Interface), which is a low-overhead transport protocol designed for single segment networks. NetBIOS over IPX, the protocol used by Novell, is also popular. Samba uses NetBIOS over TCP/IP, which has multiple advantages.

TCP/IP is already implemented on every operating system worth its salt, so it has been relatively easy to port Samba to virtually every flavor of UNIX, as well as OS/2, VMS, AmigaOS, Apple's Rhapsody (which is really NextSTEP) and (amazingly) mainframe operating systems like CMS. Samba is also used in embedded systems, such as stand-alone printer servers and Whistle's InterJet Internet appliance. Using TCP/IP also means that Samba fits in nicely on large TCP/IP networks, such as the Internet. Recognizing these advantages, Microsoft has renamed the combination of SMB and NetBIOS over TCP/IP the Common Internet Filesystem (CIFS). Microsoft is currently working to have CIFS accepted as an Internet standard for file transfer.

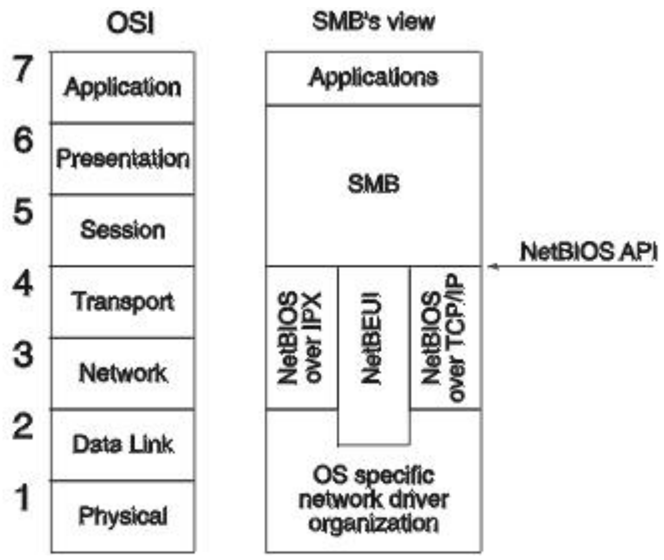


Figure 2. SMB's Network View compared to OSI Networking Reference Model

Samba's Components

A Samba server actually consists of two server programs: **smbd** and **nmbd**. **smbd** is the core of Samba. It establishes sessions, authenticates clients and provides access to the file system and printers. **nmbd** implements the “network browser”. Its role is to advertise the services that the Samba server has to offer. **nmbd** causes the Samba server to appear in the “Network Neighborhood” of Windows NT and Windows 95 machines and allows users to browse the list of available resources. It would be possible to run a Samba server without nmbd, but users would need to know ahead of time the NetBIOS name of the server and the resource on it they wish to access. **nmbd** implements the Microsoft network browser protocol, which means it participates in browser elections (sometimes called “browser wars”), and can act as a master or back-up browser. **nmbd** can also function as a WINS (Windows Internet Name Service) server, which is necessary if your network spans more than one TCP/IP subnet.

Samba also includes a collection of other tools. **smbclient** is an SMB client with a shell-based user interface, similar to FTP, that allows you to copy files to and from other SMB servers, as well as allowing you to access SMB printer resources and send WinPopup messages. For users of Linux, there is also an SMB file system that allows you to attach a directory shared from a Windows machine into your Linux file system. **smbtar** is a shell script that uses smbclient to store a remote Windows file share to, or restore a Windows file share from a standard UNIX tar file.

The **testparm** command, which parses and describes the contents of your smb.conf file, is particularly useful since it provides an easy way to detect configuration mistakes. Other commands are used to administer Samba's encrypted password file, configure alternate character sets for international use and diagnose problems.

Configuring Samba

As usual, the best way to explain what a program can do is to show some examples. For two reasons, these examples assume that you already have Samba installed. First, explaining how to build and install Samba would be enough material for an article of its own. Second, since Samba is available as Red Hat and Debian packages shortly after each new stable release is announced, installation under Linux is a snap. Further, most “base” installations of popular distributions already automatically install Samba.

Before Samba version 1.9.18 it was necessary to compile Samba yourself if you wished to use encrypted password authentication. This was true because Samba used a DES library to implement encryption, making it technically classified as a munition by the U.S. government. Binary versions of Samba with encrypted password support could not be legally exported from the United States, which led mirror sites to avoid distributing pre-compiled copies of Samba with encryption enabled. Starting with version 1.9.18, Samba uses a modified DES algorithm not subject to export restrictions. Now the only reason to build Samba yourself is if you like to test the latest alpha releases or you wish to build Samba with non-standard features.

Since SMB is a large and complex protocol, configuring Samba can be daunting. Over 170 different configuration options can appear in the smb.conf file, Samba's configuration file. In spite of this, have no fear. Like nearly all aspects of UNIX, it is pretty easy to get a simple configuration up and running. You can then refine this configuration over time as you learn the function of each parameter. Last, the latest version of Samba, when this article was written in late January, was 1.9.18p1. It is possible that the behavior of some of these options will have changed by the time this is printed. As usual, the documentation included with the Samba distribution (especially the README file) is the definitive source of information.

The smb.conf file is stored by the Red Hat and Debian distributions in the /etc directory. If you have built Samba yourself and haven't modified any of the installation paths, it is probably stored in /usr/local/samba/lib/smb.conf. All of the programs in the Samba suite read this one file, which is structured like a Windows *.INI file, for configuration information. Each section in the file begins with a name surrounded by square brackets and either the name of a service or one of the special sections: **[global]**, **[homes]** or **[printers]**.

Each configuration parameter is either a global parameter, which means it controls something that affects the entire server, or a service parameter, which means it controls something specific to each service. The **[global]** section is used to set all the global configuration options, as well as the default service settings. The **[homes]** section is a special service section dynamically mapped to each user's home directory. The **[printers]** section provides an easy way to share every printer defined in the system's **printcap** file.

A Simple Configuration

The following smb.conf file describes a simple and useful Samba configuration that makes every user's home directory on my Linux box available over the network.

```
  [global]
          netbios name = FRODO
          workgroup = UAB-TUCC
          server string = John Blair's Linux Box
          security = user
          printing = lprng
  [homes]
          comment = Home Directory
          browseable = no
          read only = no
```

The settings in the **[global]** section set the name of the host, the workgroup of the host and the string that appears next to the host in the browse list. The security parameter tells Samba to use “user level” security. SMB has two modes of security: share, which associates passwords with specific resources, and user, which assigns access rights to specific users. There isn't enough space here to describe the subtleties of the two modes, but in nearly every case you will want to use user-level security.

The printing command describes the local printing system type, which tells Samba exactly how to submit print jobs, display the print queue, delete print jobs and other operations. If your printing system is one that Samba doesn't already know how to use, you can specify the commands to invoke for each print operation.

Since no encryption mode is specified, Samba will default to using plaintext password authentication to verify every connection using the standard UNIX password utilities. Remember, if your Linux distributions uses PAM, the PAM configuration must be modified to allow Samba to authenticate against the password database. The Red Hat package handles this automatically. Obviously, in many situations, using plaintext authentication is foolish. Configuring Samba to support encrypted passwords is outside the scope of this article, but is not difficult. See the file ENCRYPTION.txt in the /docs directory of the Samba distribution for details.

The settings in the **[homes]** section control the behavior of each user's home directory share. The comment parameter sets the string that appears next to the resource in the browse list. The **browseable** parameter controls whether or not a service will appear in the browse list. Something non-intuitive about the **[homes]** section is that setting **browseable = no** still means that a user's home directory will appear as a directory with its name set to the authenticated user's username. For example, with **browseable = no**, when I browse this Samba server I will see a share called **jdblair**. If **browseable = yes**, both a share called **homes** and **jdblair** would appear in the browse list. Setting **read only = no** means that users should be able to write to their home directory if they are properly authenticated. They would not, however, be able to write to their home directory if the UNIX access rights on their home directory prevented them from doing so. Setting **read only = yes** would mean that the user would not be able to write to their home directory regardless of the actual UNIX permissions.

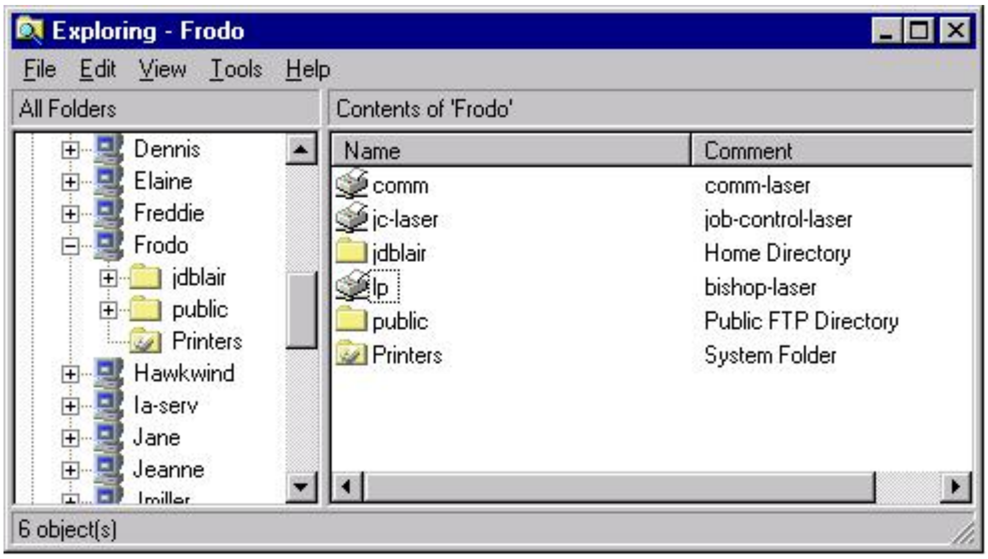
The following configuration section would grant access to every printer that appears in the printcap file to any user that can log into the Samba server. Note that the **guest ok = yes** normally doesn't grant access to every user when the server is using user-level security. Every print service must define **printable = yes**.

```
[printers]
        browseable = no
        guest ok = yes
        printable = yes
```

This last configuration snippet adds a server share called public that grants read-only access to the anonymous ftp directory. You will have to set up the printer driver on the client machine. You can use the **printer name** and **printer driver** commands to automate the process of setting up the printer client on Windows 95 and Windows NT clients.

```
[public]
        comment = Public FTP Directory
        path = /home/ftp/pub
        browseable = yes
```

```
          read only = yes
          guest ok = yes
```



Name	Comment
comm	comm-laser
jc-laser	job-control-laser
jdblair	Home Directory
lp	bishop-laser
public	Public FTP Directory
Printers	System Folder

Figure 3. Appearance of Samba Configuration in Windows Explorer

Be aware that this description doesn't explain some subtle issues, such as the difference between user and share level security and other authentication issues. It also barely scratches the surface of what Samba can do. On the other hand, it's a good example of how easy it can be to create a simple but working smb.conf file.

Conclusions

Samba is the tool of choice for bridging the gap between UNIX and Windows systems. This article discussed using Samba on Linux in particular, but it is also an excellent tool for providing access to more traditional UNIX systems like Sun and RS/6000 servers. Further, Samba exemplifies the best features of free software, especially when compared to commercial offerings. Samba is powerful, well supported and under continuous active improvement by the Samba Team.

Resources

John Blair When not evangelizing Linux, currently works as a UNIX and Windows NT consultant for brainwell.com, inc. Amongst other services, brainwell.com provides commercial Samba support. He can be reached at john.blair@brainwell.com.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Softfocus BTree/ISAM v3.1

Edmund P. Morgan

Issue #51, July 1998

Softfocus offers a low-cost solution (with source code) for your data management needs.

- Manufacturer: Softfocus
- E-mail: jon@tap.net
- URL: http://www.greymatter.co.uk/gmWEB/Items/BND00133.HTM
- Price: single-user $115 US, multi-user $175 US
- Reviewer: Edmund P. Morgan

Developers, have you ever needed a way to store data without the overhead of a RDBMS (relational database management system)? You are now in luck, because Softfocus offers a low-cost solution (with source code) for your data management needs.

Documentation

The documentation is supplied in the form of a 171-page manual, which is brief yet packed with a wealth of knowledge and information. It provides a table of contents, index and appendix. The manual starts with a explanation of the BTree and ISAM (indexed sequential access method) concepts. The next section links the concepts with instructions on implementing applications under this product. Most of the manual is reserved for explaining the API (application programming interface).

The manual provides you with the following information:

- Function parameter list
- Function return code
- Message text of the return code
- What the error implies

- Multi-user (multi-tasking OS) information about this function
- Location of the file pointer after a function call
- Other information and descriptions of the function
- Any changes from the last version

The manual also provides plenty of examples to guide you through the process of building your application.

Installation

Installing this product is a snap—you just copy the files to your hard disk. All of the source code, the Makefile, the configuration file and other files are available to make compilation easy. The product supports a variety of C compilers and environments, and assumes you are familiar with C. I have used this product in various environments (i.e., DOS, Linux, Windows 95, Windows NT, HP-UX, DEC UNIX, SGI Irix, Solaris and Dynix/PTX).

Software

The software distribution is comprised of over 120 files, including over 15,000 lines of source code. This product can support the following:

- Database file size limited by disk space
- Record size up to 65KB
- Virtually unlimited database files open at the same time (depending on OS)
- Keys (string, integer, binary, long integer, floating point, user defined)
- One index or multiple indices
- Duplicate or non duplicate key values
- Ascending and descending index

The software via the API allows easy database application development, and only one include file gives you access to the APIs. The APIs are divided into three sections. The first is the high level ISAM APIs. The database layout is based on C structures that you provide when building your application. These APIs give you the ability to write programs without knowing the low level details of database management. These APIs begin with the prefix “bt3” for the function names. This scheme gives you quick access to information only associated with these high level APIs. These APIs are appropriately named so as not to divert your attention from your application. The API function names include names such as **create**, **open**, **add**, **close** and **delete**. These functions perform the following tasks:

- **create**: create a database.

- **open**: open a database.
- **add**: add a new record to a database.
- **delete**: remove a record from a database.
- **close**: close a database.

Many other ISAM APIs can help you search through the database, allocate memory, lock and unlock a database or record, flush a record to the database, etc. This product includes the variable length and low level BTree APIs. The variable length API manages data with varying lengths. The low level BTree API handles all of the database management details. When you use the ISAM APIs, they call the BTree APIs. Since I have used only the ISAM APIs directly, I cannot comment much on the use of the other APIs. Each API gives you access to the following information:

- Function return code
- Message text of the return code

If the examples in the manual are not enough, the distribution comes with plenty of demos and test programs to investigate. The distribution also includes a utility to fix most problems associated with database indexing and corruption. Typically, I call this utility once before manipulating an existing database.

Conclusion

I have been using this product for over five years. I would recommend it to any developer who needs database management without the overhead of a RDBMS. This product is lean, fast and does not require a lot of disk space. You, as the developer, have complete control of your database management application. This product is easy to port (recompile), and the manual provides complete information. Also, no license is needed to distribute your application. The best thing about Softfocus BTree is that, like Linux, source code is included.

Edmund P. Morgan has been involved in the computer industry as a software developer since 1983. His favorite environment is Linux and C. His most interesting project involves the current after hours work of automating the entire information management infrastructure of his local church. And, of course, Linux is the server OS and development environment of choice. He can be reached at emorgan@cup.net.

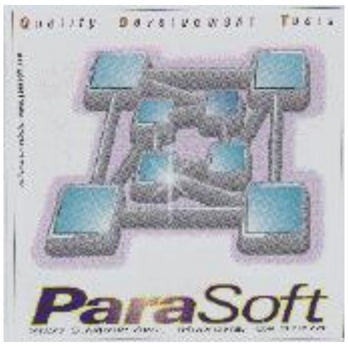
Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Insure++

Jim Nance

Issue #51, July 1998

In order to combat the “creative user” problem, there is a type of program which will take source code or object files and produce a version that analyzes itself for bugs as it runs.



- Manufacturer: Parasoft Corporation
- E-mail: info@parasoft.com
- URL: http://www.parasoft.com/
- Price: $1,995 US
- Reviewer: Jim Nance

For the last four years I have worked as a programmer writing software to find errors in integrated circuit designs. During this time I have learned a lot about chasing bugs. Ideally, you want to find and fix a program's bugs before you ship it to your customers. Remarkably, customers seem to be extremely creative people who can figure out how to use (and break) programs in ways programmers have never foreseen.

In order to combat the “creative user” problem, there is a type of program which will take source code or object files and produce a version that analyzes itself for bugs as it runs. The wonderful thing about this type of program is that it allows you to find bugs that are not causing any visible problems, so that you can fix them before they cause anyone trouble. We have found these programs to be invaluable at work.

Parasoft Corporation produces one of these programs, which they market under the name of Insure++. We recently evaluated the Solaris version of Insure++ at work, and I was excited to learn that they also had a Linux version.

Getting Started

A few weeks later I got e-mail from a sales person at Parasoft. She introduced herself and offered to put me in touch with their programmers if I had any technical problems. She went on to tell me that their product currently only worked with libc5 and not glibc, but that they were working on glibc support. I was impressed with both her helpful attitude, and the fact that she knew about glibc, which had only been available for two months.

A few days later Insure++ arrived at my house. The box contained a CD-ROM, a 10-page booklet with installation instructions and a 500-page user's manual. I had the software installed on the computer within five minutes, even though I had one minor problem with their installation script. I then called Parasoft to get a license key. I was very impressed with the salesperson who answered. After he gave me the key, he helped me create a $HOME/.psrc file, the startup file for Insure++, and he walked me through one of the examples included on the CD-ROM. Then he showed me a few features of the product and gave me his telephone extension and told me to call him if I had any problems.

Learning about Insure++

Insure++ operates by taking your C or C++ source code and creating a new file which contains your code plus some automatically generated statements. The purpose of these statements is to analyze how your program is using memory, function calls and variables, so that potential problems can be found. Insure++'s analysis is extremely detailed. It knows when you use uninitialized variables or memory. It knows when there are no longer any pointers to allocated memory (leaks). It knows when you reference past the end of an array or structure. It knows when you call functions improperly. And it knows even more. Insure++'s analysis is also very robust. It can handle programs that use threads and programs that use memory obtained from files created by **mmap** or SysV shared memory objects.

Insure++ is also easy to use. Instead of compiling your program with **gcc**, you compile and link it with a program called **insure**. The insure compiler takes care of generating the modified source files, compiling them with gcc and then deleting them. It also does compile-time error checking. After the program is compiled you run it in the normal fashion, and it runs as normal, except that it is analyzing itself for errors. Errors found at compile or run time can be logged to a file, to stderr or to stdout, and error messages can be customized in order to be interpreted by programs such as Emacs. The default behavior is to send error messages to an X11-based program called Insra. Insra displays the error messages in an easy-to-understand manner, and it acts as an interface with your editor. Insra can also save the errors, allowing you to reload them and fix the problems later.

Most programs are not completely self-contained. Instead, they use code from system libraries like the C library or the X11 library. In order to fully check your program for errors, these libraries must be compiled with insure. Since most people have no interest in recompiling something like the X11 library, Insure++ comes with precompiled versions of several system libraries including libc, libm, libX11, libXaw, libXt and libdlsym. If you need to use a library that's not included with Insure++, and you can't or don't want to recompile it yourself, you can just link with the standard library. Insure++ will still be able to do some error checking of the library functions, but it will not be as detailed or complete as it would be if the library compiled with insure.

Installation

The first thing I did with Insure++ was to get out a Sunsite archive CD and start compiling programs. This proved to be an excellent way to learn how to use the product. All the programs I tried proved to be easy to compile. In each case, I simply overrode the appropriate variables on the **make** command line to cause the program to be built with the insure compiler and the **-g** debug flag. For example, this command works with many X11 programs:

```
make CC=insure CDEBUGOPTS=-g
```

When make invokes the first insure compiler, the compiler starts up the Insra GUI to display any problems Insure++ detects at compile time. All future invocations of the compiler talk to the existing Insra process so you don't have a new window popping up for each compiler process that runs.

After the program is built, you just run it as you always have. When it starts up, it sends messages about any bugs it finds to Insra. Insra will display the source code in which the error occurred and the stack trace when it occurred. If you click on an element in the stack trace, Insra will start up your editor in the appropriate file so you can fix the problem.

One thing you notice when running programs compiled with insure is that they run *much* slower. I wrote a few test programs to try to quantify this phenomenon. If a program sits in a tight loop, makes no function calls and does not access any arrays, it does not slow down at all. If a program spends most of its time calling functions or accessing arrays, it slows down by about a factor of 80. Most programs do call functions and access arrays, so you can expect to see a significant slowdown. A good rule of thumb is every second of user time the program usually takes is going to translate to a minute of user time with Insure++.

I had a few minor problems with Insure++, such as Insra not being able to find source code when that code was spread out in multiple directories. I was always able to resolve these problems quickly by referring to the manual, which is well-indexed and well written. In fact, it is written well enough that it is fun to read even if you are not trying to solve a problem.

At one point I thought I had found a bug in Insure++. I had written the following test program:

```
  int main()
   {
          char *x = malloc(30);
          char z = x[1];
          char y = x[31];
          int zz;
          x[0] = 0;
          z += 3;
          free(x);
          zz = x[0];
          return z*z;
   }
```

Insure++ detected that I had accessed past the end of the x[]array when I initialized y and that I had used x[] after I had freed its memory. However, it did not detect that I had initialized z with an uninitialized member of x[] (i.e., x[1] had not been initialized). I was excited because this omission gave me a chance to try out Parasoft's technical support. I put the test program on one of my web pages so I could show it to the people at Parasoft. I then called up the salesperson who had provided my license key and told him I had found a bug. He transferred me to one of their programmers. I gave the programmer the URL where the code was, and he took a look at it while I was on the phone with him. He told me that by default Insure++ does not check to see if variables less than 4 bytes long are uninitialized. He then told me what to put in my .psrc file to change this. Then he gave me his e-mail address and telephone extension so I could contact him if I had any other problems.

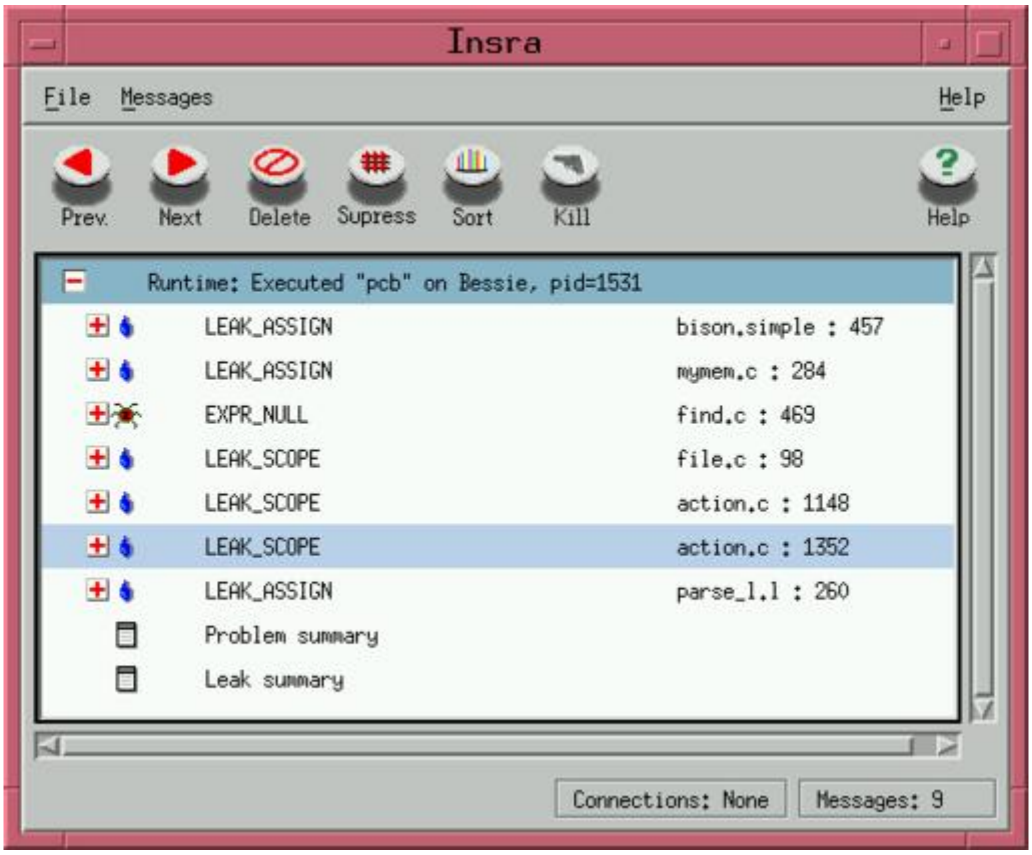
I had three separate interactions with three different people at Parasoft, and each interaction was very positive. I figured that people doing reviews for magazines got VIP treatment, but I wondered how other people would be treated. I found someone at work who had used Insure++ at a prior job, and I asked him what he thought about Parasoft's technical support. He told me that he also thought it had been excellent—so much for my special treatment.

Advanced Debugging

After I had gotten familiar with the basics of using Insure++, I decided to try some of the more advanced features. The first one I investigated was the interaction between **gdb** and programs compiled with insure. Insure does a good job of hiding the modifications it makes to the source code from the debugger. For the most part, you can't tell that anything is different. One thing that is different is that the program will call the function **_Insure_trap_error** whenever it detects a problem in your code. By setting a debugger breakpoint in this function, you can get the program to stop each time Insure++ finds a problem. Then you can use the debugger to examine your program's variables and find out why the problem occurred. I tried this on a few programs and found it to be a very useful feature. There are also other functions you can call from the debugger to get information about which location in the program memory was allocated for a variable and how much memory is currently allocated.

Another feature I classified as advanced has to do with programs that use their own memory managers. Insure++ knows about **malloc**, **calloc**, **free**, **new**, **delete** and other standard memory management functions. This allows it to do in-depth error checking when you use these functions. It is fairly common for programs to have their own memory managers which allocate large blocks of memory and dole it out themselves. In order to get detailed error checking for these programs, it is necessary to teach Insure++ about your memory manager. I did not do this myself, but the people evaluating Insure++ at work did, and they indicated it was a fairly straightforward task. It is even possible to teach Insure++ about functions that have nothing to do with memory management and have it verify arbitrary things about the state of your program each time the function is called.

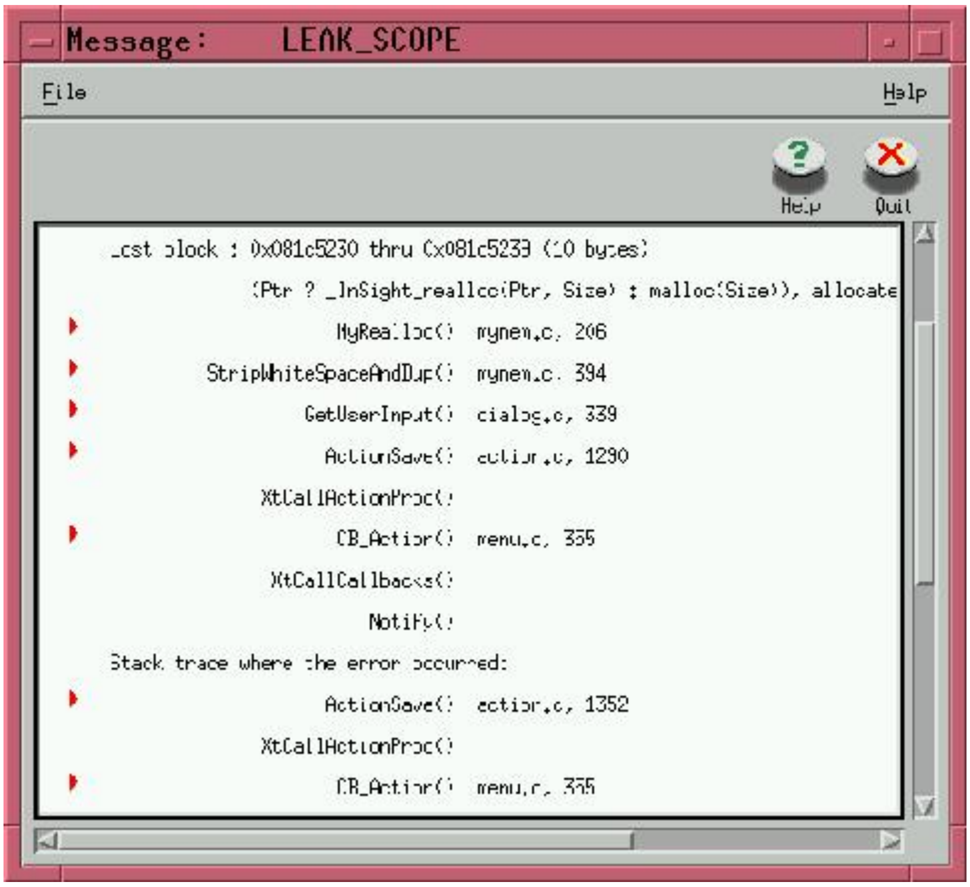
An Example



```
Runtime: Executed "pcb" on Bessie, pid=1531
    LEAK_ASSIGN          bison.simple : 457
    LEAK_ASSIGN          mymem.c : 284
    EXPR_NULL            find.c : 469
    LEAK_SCOPE           file.c : 98
    LEAK_SCOPE           action.c : 1148
    LEAK_SCOPE           action.c : 1352
    LEAK_ASSIGN          parse_1.l : 260
    Problem summary
    Leak summary

Connections: None    Messages: 9
```

Figure 1. Insra Window Showing Errors



```
  Lost block : 0x081c5230 thru 0x081c5239 (10 bytes)
                (Ptr ? _InSight_realloc(Ptr, Size) : malloc(Size)), allocate
                     MyRealloc()  mymem.c, 206
       StripWhiteSpaceAndDup()  mymem.c, 394
                  GetUserInput()  dialog.c, 339
                    ActionSave()  action.c, 1290
              XtCallActionProc()
                      CB_Action()  menu.c, 335
               XtCallCallbacks()
                          Notify()
  Stack trace where the error occurred:
                    ActionSave()  action.c, 1352
              XtCallActionProc()
                      CB_Action()  menu.c, 335
```

Figure 2. Stack Trace at Memory Leak Location

By now, I felt I knew enough about Insure++ to do an example for this review. I selected a printed circuit layout program called pcb-1.3 in the apps/circuits directory of sunsite.unc.edu. (There is a newer version of pcb available.) This program consists of slightly over 600KB of source code, so it is a reasonably complex program, even though it is by no means huge. When compiling the program, Insure++ warned about several possible problems, but none of them looked too serious. I then started up **pcb** using the example data that came with it. I had no idea how to use the pcb software, so I just started clicking on things. (This is usually an excellent way to find bugs.) After a few minutes of playing with the program, I had six memory leaks and one NULL pointer evaluation listed in the Insra window (see Figure 1). At this point, I clicked on the error for line 1352 of the program action.c, and the screen shown in Figure 2 popped up. This shows the stack trace when the memory was allocated and when it was leaked. If you click on the red arrows, the indicated source code will be opened in your favorite editor. A quick look at the source code shows why Insure++ is complaining. In the function **ActionSave** on line 1290 of action.c, the program gets a file name from the user:

```
1289 case F_LayoutAs:
1290  name = GetUserInput("enter filename:", "");
1291  if (name)
1292  SavePCB(name);
1293  break;
```

This file name is contained in memory set by malloc, pointed to by the variable **name**, a local variable in ActionSave. Later, at line 1352, we leave ActionSave:

```
1349  break;
1350  }
1351 }
1352 }
```

There is no code in ActionSave to free the memory pointed to by **name**. When we leave the function ActionSave and the local variable **name** ceases to exist, there is no way to access this memory anymore—it has been leaked. Insure++ is warning us about this situation.

In addition to several memory leaks, Insure++ found a bug in the program related to using NULL pointers. It complained about this line:

```
469 return(&LineSortedByHighX[Layer][index2]);
```

I couldn't see anything wrong with this line by looking at it, so I restarted the program in the debugger, using the _Insure_trap_error breakpoint I mentioned earlier. When the program stopped, I examined the expression and it turns out that LineSortedByHighX[Layer] is NULL. The reason the program does not crash on this line is that we are taking the address of LineSortedByHighX[Layer][index2] rather than trying to dereference it. Presumably, at some later time, some function will evaluate this address and crash. Insure++ makes it possible to fix the problem and prevent a crash.

Conclusion

As you might be able to tell, I think Insure++ is a great product. It finds programming errors better than any other product I have used, it runs under my favorite operating system, and Parasoft's technical support is excellent. There is one problem—it's expensive. The cheapest configuration you can buy is a 3-user-node locked license, which costs $1,995 per node. Nothing I do on my home computer is worth that much money, so I won't be buying a copy of Insure++ for myself. I suspect that most people who program for fun will not be buying a copy either. Who should buy Insure++ then? People, or more likely companies, who do professional software development. When you are paying programmers several thousand dollars a month and bugs cost big money to deal with, then the price begins to look more attractive. When you consider that Insure++ might enable you to ship your product earlier, it begins to look very attractive. If you develop software for a living, you need this product. Insure++ runs under several flavors of UNIX and Windows too. In fact, anyone who is in an environment where programming time is money should consider evaluating Insure++. It is an excellent product.

Jim Nance (jlnance@avanticorp.com) works as a software engineer for the Verification Products Division of Avant! Corporation in Research Triangle Park, North Carolina. He has been using Linux since kernel version 0.12. In addition to hacking Linux, he enjoys reading murder mysteries, getting bossed around by his 3-year-old daughter, and spending time with his Sunday school class.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Combining Apache and Perl

Reuven M. Lerner

Issue #51, July 1998

This month Mr. Lerner gives us a look at mod-perl, a module for the Apache web server.

The CGI (Common Gateway Interface) standard has been around for several years and is beginning to show its age. CGI is great because all web servers support it, programmers can write in any language, and programs can be portable across a large number of platforms. Netscape's NSAPI and Microsoft's ISAPI bind more tightly to their respective web servers, but programmers interested in using these APIs are much more restricted than with CGI.

A particularly big problem with CGI is its inefficiency. Each invocation of a CGI program creates a new process on the server. If you write CGI programs in Perl, you are starting a new copy of Perl each time a CGI program runs, using additional memory and processor time. Wouldn't it be nice if we could have the flexibility of CGI programs without having to use all of those system resources? Better yet, wouldn't it be great if we could use our existing CGI programs in such a framework with little or no modification? The answer, of course, is “yes”; even as hardware continues to get cheaper and more powerful, it seems silly to be wasting memory and CPU time unnecessarily.

This month, we look at **mod_perl**--one of the proposed solutions to this problem. mod_perl is a module for the popular and powerful Apache web server, which runs on many operating systems including Linux. At the most basic level, mod_perl makes it possible to run server-side Perl programs more efficiently than when using the CGI protocol. However, mod_perl offers much more than efficiency, as we will see. It also provides a full interface to the Apache internals, giving Perl programmers a chance to modify the web server itself.

Retrieving and Installing mod_perl

Apache modules are configured and installed at compile time. If you are interested in installing mod_perl, you have to download and recompile the source code in Apache. Luckily, this is rather easy to do. Note that while anyone can download, configure and compile Apache, only someone with root access can install Apache to its default position. If you don't have root access, you will still be able to run, but only on an unrestricted port number, namely, one above 1024.

The latest version of mod_perl is always available from CPAN (Comprehensive Perl Archive Network). At this time, the latest version of mod_perl is 1.10, which means that you can retrieve it from http://www.perl.com/CPAN/modules/by-module/Apache/mod_perl-1.10.tar.gz. Later versions will have the same URL, with a different version number. In addition, try to use a CPAN mirror close to you, rather than loading down www.perl.com; go to http://www.perl.com/CPAN/ for help in finding one.

Once you have downloaded mod_perl, you will also have to download the latest version of Apache, 1.2.6, from http://www.apache.org/ or one of its mirrors. Unpack the Apache and mod_perl distributions in the same directory. On my system, I did the following:

```
cd /downloads
tar -zxvf apache_1.2.6.tar.gz
tar -zxvf mod_perl-1.10.tar.gz
```

If you want to modify the default Apache module set, now is the time to modify /src/Configuration. If you are not familiar with Apache configuration, don't worry—things will work just fine without customizing the module set.

The rest of the Apache configuration and compilation is done within the mod_perl directory. Move into the mod_perl directory (probably called something like mod_perl-1.10) and type:

```
perl Makefile.PL
```

On my system, mod_perl asks me two questions:

```
Configure mod_perl with ../apache_1.2.6/src ? [y]
```

to which I press **return**, and

```
Shall I build httpd in ../apache_1.2.6/src for you? [y]
```

to which I press **return** again. This configures all of the files necessary for building mod_perl and Apache. When the UNIX shell prompt returns, simply type **make** and press **return**. The resulting Apache binary (**httpd**) will be in the src subdirectory under the Apache directory. On my system, httpd resides in /usr/sbin/httpd, so copying the resulting binary will replace the old Apache with the new one.

Restart Apache by logging in as root and typing:

```
killall -1 -v httpd
```

Now, you're in business with your new version of Apache. If you're not sure whether the new version has been installed, connect to the web server and ask for its version information:

```
telnet localhost 80
```

After connecting, type:

```
HEAD / HTTP/1.0
```

On my system, I get the following response:

```
HTTP/1.1 200 OK
Date: Sun, 12 Apr 1998 19:02:41 GMT
Server: Apache/1.2.6 mod_perl/1.10
Connection: close
Content-Type: text/html
```

In other words, the web server running on port 80 (the default port for HTTP traffic) is running Apache 1.2.6, with mod_perl 1.10 compiled in.

Configuring Apache for mod_perl

One of the most popular uses for mod_perl is as a fast replacement for CGI. In order to use it this way, we need to modify Apache's configuration files, so it knows how to handle programs that use mod_perl.

Why must Apache know how to treat these programs? Thinking about CGI programs should make it clear. Browsers request CGI programs in exactly the way they request static documents. The browser does not know whether a given URL points to a program or a static document; that determination is made by the server. If the request is for a static document, the server returns the document verbatim to the user's browser. If the request is for a program, the server executes it and returns any output to the user's browser.

In both of these cases, the browser's behavior is the same: it sends the request to the server and displays the contents of any received response. This places the onus on the server to recognize which files are to be transmitted verbatim, and which are programs whose output will be sent as a response. Apache lets us choose between allowing CGI programs to be located anywhere on the system (as long as they end with an agreed-upon suffix, such as .pl or .cgi) and requiring that they be located in one or more designated directories. This is done using directives in the Apache configuration files.

Now that we have added mod_perl to our server, we must tell Apache how to handle three types of URLs: static documents, CGI programs and mod_perl programs. Adding mod_perl to the mix does not have to change the existing configuration on your system. I created a directory named perl-bin under my web root directory (/home/httpd/perl-bin) and decided all mod_perl programs would reside there, just as all CGI programs reside in cgi-bin. I then added the following lines to my server's srm.conf file:

```
<Location /perl-bin>
SetHandler perl-script
PerlHandler Apache::Registry
Options ExecCGI
</Location>
```

The **<Location>** and **</Location>** tags indicate that we want our settings to take effect for a particular directory, rather than the entire Apache server. Then, we tell Apache to treat documents in the perl-bin directory as Perl scripts, rather than static documents or something else. If you are curious, the Apache manual has an entire section describing handlers, including the **AddHandler** and **SetHandler** directives that allow us to configure file types according to location or file extension. Other handlers, for instance, include cgi-script (for CGI programs), server-info (for information about the server) and imap-file (for image maps).

Now that Apache knows which files in /perl-bin should be considered mod_perl programs, we must tell mod_perl how to handle these Perl documents. We will use the **Apache::Registry** module, which allows us to run CGI programs. Finally, we will use the **Options** directive to allow CGI programs to be run within this directory.

Finally, we make one last modification to srm.conf, telling mod_perl to produce HTTP headers. We do that outside of the **<Location>** directive, since we always want mod_perl to return complete headers. The line to add is:

```
PerlSendHeader On
```

Adding the PerlSendHeader directive does not relieve us from the responsibility of indicating the type of content we are returning. In other words, we still must add the “Content-type” header to the top of our output, just as we do when writing CGI programs.

Basic Programs with mod_perl

All the pieces are now in place to use mod_perl instead of CGI programs. Let's try a simple program that prints out the current state of the environment. Copy the following into a file called test.pl in the perl-bin directory:

```
use strict;
print "Content-type: text/html\n\n";
foreach my $key (sort keys %ENV)
{
print "\"$key\" =
\"$ENV{$key}\"<BR>\n";
}
```

Set permissions so that the file is executable, and ask your browser to retrieve /perl-bin/test.pl. If all goes well, you will see a list of environment variables in your browser.

If you have been writing CGI programs (or using Perl for any length of time), then the above might seem strange. For example, where is the initial line indicating the location of the Perl interpreter, as well as its switches? The initial hash-bang (#!) syntax which we are so accustomed to is missing because it's unnecessary. That two-character code tells the UNIX shell that it shouldn't try to interpret a program (i.e., as a shell script), but rather that it should give the responsibility to another program. That's why Perl programs usually begin with the line:

```
#!/usr/bin/perl
```

while Tcl programs begin with:

```
#!/usr/bin/tclsh
```

and so forth. Because our program is run by mod_perl and mod_perl understands Perl programs, we don't need the hash-bang syntax at the top of our program.

Command-line switches raise a more subtle issue, one that cuts to the heart of mod_perl's advantages over standard CGI programs. Programs run much faster under mod_perl for several reasons, but the two primary ones are that Perl is embedded in Apache (saving the overhead of starting Perl with each invocation), and programs are compiled once, then cached (saving the overhead of compilation with each invocation). The combination of embedding Perl within Apache and caching compiled programs can mean a tremendous boost in execution speed, often ranging from 400 percent to 2000 percent.

There are tradeoffs for these increases in speed, and one of them is that command-line switches no longer work as expected. Switches are handled at compilation time, so if you expect switches to work each time your program is run, you will be disappointed. However, all is not lost. Programmers interested in turning on Perl's warnings (the **-w** flag) and security checks (the **-T** flag, for tainting) from within mod_perl programs can do so with a directive inside of the srm.conf file. To turn on warnings, you simply add the line:

```
PerlWarn On
```

This has the effect of turning on warnings from within your programs. As usual, warning messages are sent to the Apache error log.

By the same token, you can activate Perl's security checks (commonly known as “tainting”) by adding the **PerlTaintCheck** directive inside of srm.conf:

```
PerlTaintCheck On
```

When you write CGI programs (or any other programs, for that matter) in Perl, it is usually a good idea to include the **use strict** directive, as we saw in the above example. When programming with mod_perl, however, it is extremely important to **use strict**. Otherwise, variable definitions may remain in memory after your program exits, creating problems for future invocations of this or other programs.

By the same token, do not use the **exit** function to leave your program prematurely. Normally, calling exit from within a CGI program will end the program—not a bad thing, if it has already produced all of its output. If you call exit from within a mod_perl program, the program takes Perl along with it; and since Perl is embedded within the copy of Apache, killing Perl effectively kills that particular server process as well. If you absolutely must call exit from within your program, use **Apache::exit** instead; it will do what you want without unexpected side effects.

CGI Programs with mod_perl

Now that we have gone through a basic introduction to mod_perl and writing CGI-style programs with Apache::Registry, let's look at an example of CGI programs under mod_perl—a simple guest book program that takes form parameters and appends their contents to a file on the system. The form is shown in Listing 1. Note that the form looks just like the forms we have seen in the past. The sole difference is our form's action, which sits in perl-bin rather than the usual cgi-bin.

The program is shown in Listing 2. If we were to add a “hash-bang” first line to this program, it would operate equally well under CGI or mod_perl environments. We use CGI.pm to retrieve information about the submitted form. While this works just fine for recent versions of CGI.pm, earlier versions are not completely compatible with Apache::Registry.

The main difference between the program in Listing 2 and its CGI counterpart is speed. While I cannot give exact numbers, my subjective tests showed the response from mod_perl to be almost instantaneous, with the CGI version taking noticeably longer—perhaps up to one second. This might not seem like much, but the combination of a cached CGI program with an already-running version of Perl is impressive, even with a short program that compiles quickly. As you can see, it does not require many changes to your original program.

A non-CGI use of mod_perl

So far, I have mentioned mod_perl only as a replacement for CGI. However, mod_perl is much more than that; it gives you a Perl interface to the guts of Apache. If you have configured your server correctly, you can modify every aspect of Apache using a Perl program. Better yet, some enterprising souls have already spent time writing modules which do just that. For example, the **Apache::Status** module allows you to take a look at the current state of mod_perl running on your server. Apache::Status comes with mod_perl and is a good example of what can be done with this package.

As was the case with Apache::Registry, we are going to have to set the handlers for a particular directory. In this case, the directory does not have to physically exist on the disk, since the URL is interpreted before a file is ever opened. You must add these lines to your srm.conf file in order to get the Perl status:

```
<Location /perl-status>
SetHandler perl-script
PerlHandler Apache::Status
</Location>
```

As was the case with Apache::Registry, we set the Apache handler to be **perl-script**. Since we want Apache::Status to be handling the perl-status directory, we point to it as our **PerlHandler**.

If you put the above lines in your server's srm.conf file and restart the server, anyone requesting /perl-status from your server is going to have access to information about your server. If you would prefer to keep such information private, you must use access controls, as shown in the following example:

```
<Location /perl-status>
SetHandler perl-script
PerlHandler Apache::Status
order deny,allow
deny from all
allow from 127.
</Location>
```

This allows you to retrieve status information from the server computer itself; attempts to retrieve /perl-status from another computer will be greeted with an “unauthorized access” message.

What Next?

I have been surprised and impressed by mod_perl's speed and flexibility, and I expect to use it more and more as time goes on. The fact that it can run most existing CGI programs without modification is a great boon to those of us who already have a stockpile of such programs.

mod_perl is not a panacea, of course. Its speed comes at a price; namely, greater demands for system memory. The inclusion of Perl (a known memory hog) inside of Apache means that the httpd processes on your server will start off larger than otherwise. Over time, each server process will grow, as compiled Perl programs are cached in memory. Before you use mod_perl on your system, you should make some calculations regarding the amount of memory that Apache is using; this may affect the number of server processes you want to run on your system.

Nevertheless, mod_perl is a tremendous advance for both Apache and Perl and promises to get much better with time. Next month, we will look at some of the ways in which mod_perl can speed up our database connections, making Apache an even better server for dynamic sites dependent on relational databases.

Resources

Reuven M. Lerner is an Internet and Web consultant living in Haifa, Israel, who has been using the Web since early 1993. In his spare time, he cooks, reads and volunteers with educational projects in his community. You can reach him at reuven@netvision.net.il.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Letters to the Editor

Various

Issue #51, July 1998

Readers sound off.

Satellite Remote Sensing

I have just received the April 1998 issue of *Linux Journal*. My highest regards for such an informative article as “Satellite Remote Sensing of the Oceans” by S. Keogh, E. Oikonomou, D. Ballestero and I. Robinson. My attention was held very closely by the details and explanations written in this article. I'm well aware of the problems in remote sensing systems, and the potentially enormous amounts of data which must be manipulated in order to make sense of it. This was *great*! Find more authors like them.

—Bill Staehle staehle@netvalue.net

Kernel Korner

First let me congratulate all of *LJ*'s editors. I am staying up hours to digest and learn all I can possibly read. It is really wonderful to have a chance to learn so much.

Here I am referring to “Writing a Linux Driver” by Fernando Matía. I find it to be a good foundation for device drivers. An added reference for anyone who would like to look into drivers further is *Writing A UNIX Device Driver* by Janet Egan and Thomas Teixeira (John Wiley and Sons, 1988). Even though it is for UNIX, it proves very useful for Linux. Thanking you and Fernando Matía.

—Silvio agola@grumpy.igpp.ucla.edu

About Issue 47 (GUI)

I have subscribed to your wonderful magazine since 1995 and I have found it to be useful, informative and fair. But I must tell you my disappointment about your graphical user interfaces issue, March 1998.a) XView is old and passing out of favor, even though it is still useful.b) CDE is proprietary.c) TkDesk is nothing more than an elaborate file manager.d) GTK and GNOME are by no means ready for production machines.

You failed to mention the wonderful, useful, highly advanced KDE project (see http://www.kde.org/), despite the fact that it is in the beta three stage. I have installed it on production machines in radiological clinics under both Linux (x86 architecture) and SunOS. I wonder why you did not dedicate a few lines to a project like KDE—in my opinion, the only runner capable of stopping the wave of Microsoft's products. If libQT licenses are the problem, it is inconsistent with the presence of articles on all-commercial products like CDE and X-Designer/ Motif, since libQT is free for free Linux development. Please reconsider this position and take a look at KDE.

Thank you very much for your consideration and keep up the excellent *LJ*--I love it.

—Daniel Benenstein dbenenst@cs.com.uy

There was no conscious decision on my part to snub KDE. In fact, I had a KDE article scheduled, but it came in after the deadline and the issue was already full. I plan to run this article in the near future. The May issue does have a short comparison review of both KDE and GNOME in the *Linux Gazette* column —Editor

Dealing with Cookies

The most elegant method for dealing with cookies is simply to symbolically link the cookie file to /dev/null, like so:

```
ln -s ~/.netscape/cookies /dev/null
```

This has the effect of accepting all cookies, so you are not denied access to any web sites, but immediately funnels them into a black hole. Having /dev/null “world readable” is infinitely preferable to disclosing your browsing habits to webbots.

—M. Leo Cooper thegrendel@theriver.com

About Ghostscript

I have been reading *LJ* (which I find very nice) for three months. I have learned different things (as I do from any journal), but I have also found some bugs. In the article “Ghostscript” by Robert Kiesling (March 1998), there are two points I do not agree with.

The author says that it is not possible to see .eps files included in LaTeX files with **xdvi**. I say: wrong. I use LaTeX2e and xdvi almost every day, and I can see the .eps files through xdvi. (I cannot zoom, but I can see what is displayed, which is usually enough.) The .eps files I use are generated by **xfig** and **transfig**.

The author says that **gs** used with device X11 will create a window. I have tried this, and the X11 device is not a default one for gs 4.xx. I can see my .ps files with gs, but I must use Ghostview as the GUI. I do not think that the programmers have taken the X11 device away from version 3 to version 4. Has the author made a mistake in his script files?

I am using Red Hat 4.0 (with a lot of patches and a Slackware-like installation) but my LaTeX2e, xdvi and Ghost(script/view) are original and standard ones, so I should be able to do what the author mentioned, but I cannot.

—Raphael Marvie marvierx@cs.man.ac.uk

I have to blame my use or installation of xdvi for the inability to print \special commands. The distribution on which the article is based has been upgraded several times now. I've received plenty of mail from TeXperts saying they have no problem with reproducing EPS graphics on screen.

The X11 driver is standard in every recent version of Ghostscript, but it is not necessarily the default. It can, however, be specified on the command line with the parameter **-sDEVICE=X11**. During the process of building Ghostscript, it and all of the other supplied driver code can be specified in the Makefile.

—Robert Kiesling kiesling@terracom.net

PhotoShop

I noticed in your April edition, you used PhotoShop to create the cover image. I understand that at SSC you use non-Linux systems for some of your graphics and page layout. Wouldn't it be nice if you could do it all in Linux? I think the GIMP is great, but nothing on Linux compares with PhotoShop.

The only thing to do is to port PhotoShop (and possibly Framemaker) to Linux. Adobe is aware of the desire for such products, and even confronts it on the following web pages: http://www.adobe.com/supportservice/custsupport/QANDA/259e.htm and http://www.adobe.com/supportservice/custsupport/QANDA/2bf6.htm I believe if enough people show an interest, Adobe will eventually come around. What a great day that will be for Linux! Adobe asks that we send all suggestions of this nature to:

Adobe Systems, Inc.PhotoShop Product Management345 Park AvenueSan Jose, CA 95110-2704

I hope that many of you will send requests to Adobe to let them know how you feel.

—Jason F.jasonf@usi.net

APO Mailing

I'd like to ask in an open forum that computer product resellers, and Linux folks specifically, allow for U.S. forces overseas to have better web commerce access.

Although we can often order products, many web-based address databases are unable to accept an APO/FPO address. There also seems to be a small lack of understanding of how mail from the U.S. is delivered to troops stationed overseas.

When a package is sent to an APO/FPO address, it is carried by U.S. Postal Service mail to one of three military postal offices. New York serves European troops, Miami serves troops in the southern hemisphere, and San Francisco serves the Pacific installations. There is no additional cost to the shipper—the military picks up the tab as soon as it gets to the APO/FPO. If the city is added to the address, it can have terminal effects on packages, many of which never arrive or are bounced around several places. We cannot receive mail from either FedEx or UPS.

There are web commerce servers that provide for this type of addressing, but more need to be available. I appreciate the efforts of the web maintainers in keeping good commerce available and ask for this request to be considered.

—Leam Hallgers hom@spidernet.it

Lost & Found

Write us at info@linuxjournal.com or send snail mail to *Linux Journal*, P.O. Box 980985, Houston, TX 77098. All published letters are subject to editing.

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

COMDEX/Spring 1998

Jon “maddog” Hall

Issue #51, July 1998

Vendors around us were astonished by the attention and business we drew to our booths.

Photo Album

COMDEX in Chicago (April 20-23) was a titanic Linux hit. Vendors around us were astonished by the attention and business we drew to our booths.

The Linux Pavilion had a huge sign overhead (thanks to Carlie Fairchild of *Linux Journal* and Andy Wahtera, our new ZD/COMDEX representative), and multiple large floor signs guided people entering COMDEX to the Linux International Pavilion. We had a page on the COMDEX web site, mention in the Show Daily and other marketing “aids”.

Linux International vendors with booths in the Pavilion were Caldera, S.u.S.E., InfoMagic Inc., *Linux Journal* and Red Hat Software, Inc., a small number of vendors, but big in heart.

While smaller in attendance than its Las Vegas cousin, COMDEX in Chicago seemed to have a lot more end-user customers than the Las Vegas show—not really surprising when you consider Chicago is a cultural, economic and manufacturing center. While Mr. Bill was still trying to boot Windows 98 and have it stay up, the Linux International Pavilion was singing a sweet song. Some people thought we had set a new world's record for “longest line at COMDEX”--the line where people waited to pick up a free Linux CD-ROM.

I accompanied Red Hat's “booth gang”, Anna, Terry and Mike, to visit the Argonne National Laboratory and Western Suburban Chicago Linux Users Group (which thankfully is abbreviated AALUG and has its web site at http://hydra.pns.anl.gov/lug/lug-main.html). The meeting was actually held at the Fermi National Lab, which recently announced that Linux will be officially supported at their laboratory and with their applications. Donnie Barnes flew in from Durham, North Carolina to give a talk on Red Hat 5.0, and to help give out Red Hat “souvenirs”. I gave a brief talk at the end of Donnie's epic speech.

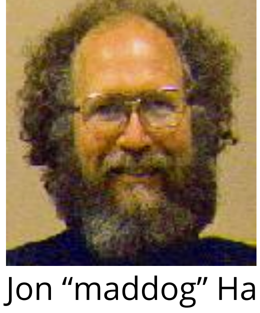
After the meeting ended, Dr. G. P. Yeh, a physicist in the computing division, invited us on a tour to see a particle-collider detector. Fermi is expanding their collider, and the new one is expected to produce more than 20 times the data of its predecessor. To expand the computing power to analyze and store this data in real time with traditional methods would have been very costly, so now Fermi is building a 1000-node Beowulf system to detect quarks (and other little things). Dr. Yeh told us that without Linux and the concept of Beowulf systems, the costs of supplying computer power for the next generation of collider would be many times what they are now forecasting.

Our sincere thanks to Dan Yocum for setting up the meeting at Fermi and advertising it, and to Dr. Yeh for showing us the collider.

On Wednesday S.u.S.E. gave a talk at the Chicagoland Linux Users Group, and on Thursday I gave a two-hour “ramble” to the same group after COMDEX was over. Then, tired and thirsty, most people retired to the Goose Island Brewpub.

The Chicagoland Linux Users Group (http://clug.cc.uic.edu/) helped to staff the Linux International booth, hand out flyers and line up user group meetings. So “thank you” to Clyde Reichie, Don Weimann, Simon Epsteyn, William Golembo, Gennagy “Ugean” Polishchuk, Long Huynh, Perry Mages, Viktorie Navratilova, Ben Galliart, Richard Hinton, and especially to Dave Blondell, the president, who organized the group and the schedules.

Linux International would like to encourage other Linux vendors to join us in the next Linux Pavilion at COMDEX, whether it be in Las Vegas or Chicago. We are definitely looking forward to the next COMDEX in the windy city. For information on membership or other information about Linux International, visit our web site, http://www.li.org/.



Jon “maddog” Hall is Senior Leader of Digital UNIX Base Product Marketing, Digital Equipment Corporation. He is President of Linux International.

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

lex and yacc: Tools Worth Knowing

Dean Allen Provins

Issue #51, July 1998

Today, computers can talk and they can listen—but how often do they do what you want?

This article is about how Linux was used to program a Sun machine to manipulate well-log recordings to support finding oil and gas exploration in Western Canada. It will describe the problem, provide enough background to make the problem understandable, and then describe how the two fascinating UNIX utilities **lex** and **yacc** were used to let a user describe exactly what he wanted to satisfy his particular need.

Some Background

In the fall of 1993, I had been recently downsized and was approached by a former colleague for assistance. He, and another former colleague, both of whom were still with my last employer, were what is known in the industry as well-log analysts.

To understand what a log analyst is requires a little knowledge of oil and gas exploration methods. Energy companies, as they like to be known, employ several different categories of professionals to assist them in finding salable quantities of hydrocarbons. Chief among these are the geologists and geophysicists (of which I am one) who study the recordings made in bore holes, or process and examine seismic recordings to identify what are popularly known as “plays” or “anomalies”.

Bore holes are simply the holes left when a drill rig moves off the drilling platform. Generally, these holes are logged by service companies who lower instruments called *tools* into the hole, and who then record on magnetic tape the readings made by those instruments.

There are many different types of tools, including sonic (which measures the time needed for a pulse of sound energy to travel through the rock wall from one end of the tool to the other), density (a continuous recording of the rock wall density), and gamma ray (a measure of gamma radiation intensity in the rock). These are just a few of the types of measurements that are made, recorded and called *logs*.

The various logs are examined by geologists to gain an understanding of what was happening when the rocks forming the bore hole were laid down, and what has happened to them subsequently as shallower rocks were created above them.

Geophysicists are more inclined to study seismic recordings which in essence are indirect measurements of the properties of the rocks forming the subsurface. Geophysics and Linux will not be discussed here, but you may find Sid Hellman's “Efficient, User Friendly Seismology”, *Linux Journal*, August 1995, Issue 16 of interest.

While seismic recordings provide much greater volumes of interpretable data over large areas, well logs are definitive measurements made at single locations, sometimes close together, and sometimes not. Geologists often correlate adjacent well logs to create cross sections of the subsurface, much like seismic recordings would provide. Detailed interpretation of individual logs, however, is often left to the log specialists.

The Problem

My two acquaintances were log specialists who wanted an additional tool to assist them in the processing and interpretation of individual or combinations of logs. Specifically, they wanted to tell the computer to perform arithmetic operations on individual or some algebraic combination of logs.

For example, they might need to scale a specific log by an arbitrary amount, say 1.73. In another case, they might want to divide one log by another, and then add the result to a third, all before adding a constant and then raising the resulting values to some arbitrary power.

Keeping in mind that logs are composed of individual sample values taken as frequently as every few inches (or centimeters as they are here in Canada and many other places in the world), these example requests would mean a reasonable amount of computation, multiplying every sample of thousands of meters of recorded log values by 1.73, in the first example. The simple scaling problem isn't particularly difficult, but identifying the desired log could be.

The energy company for which my acquaintances worked was using a simple filing method for all the log curves (a curve corresponds to all the recorded samples for one tool in one bore hole) wherein each curve was identified by a name. To this was added some additional information on units and so on, plus all the samples for all the curves for the well. All the data were stored as ASCII. (The file format is known as Log ASCII Standard format, or LAS version 2.0, and although the names for curves were generally the same from well to well, that was not guaranteed.)

As a result, more complicated combinations of curves required a fairly sophisticated and robust mechanism for arbitrary name recognition, while the desired algebraic operation was being described. Given such an interesting challenge, I recognized an opportunity to put some relatively little-used tools to work: **lex** and **yacc**.

The Tools

The program **lex** is a lexical analyzer. Lexical analysis is the recognition of words in a language. As used in this particular application, lex, or more specifically **flex**, is used to recognize characters forming the names of log curves, arithmetic operators and algebraic groupings.

flex is a particular example of the lexical analysis programs available for UNIX systems and is the implementation delivered with Linux distributions. The original implementation was done by Mike Lesk and Eric Schmidt at Bell Laboratories and is documented in the book *lex & yacc* by John R. Levine, Tony Mason & Doug Brown, O'Reilly & Associates, Inc., 1992.

yacc is a language parser. It accepts word items and, given a list of rules describing how these items form larger entities, deduces which items or combinations of items satisfy a rule. This can also be thought of as grammatical analysis. Once a rule is satisfied, a programmer's code is applied to the result.

In the case of English, the words in a sentence can be assigned grammatical types such as noun, verb, adjective and so on. Particular combinations of words form more complex units and these in turn can be described as complete sentences.

For example, the sentence “The lazy dog lay in the sun,” is composed of an article “the”, a preposition “in”, adjective “lazy”, nouns “dog, sun” and a verb “lay”. Combinations of these grammatical items form more complex entities such as noun phrases “The lazy dog” and “in the sun”. The first noun phrase is the subject of the sentence, and the second, in combination with the verb, forms the predicate. Together they form a complete sentence.

It is possible to form parsers for the English language, although given English's many idiosyncrasies, **yacc** may prove to be inadequate for the task. It may also be that the yacc programmer may become exhausted in trying to describe all the nuances of the language.

yacc was originally developed to provide a mechanism to develop compilers, but it could just as easily be used to create interpreters. For example, BASIC is often an interpreted language and could easily be described by a yacc grammar. Once yacc *understood* a particular line of BASIC code, it could cause the execution of the equivalent instructions in the native language of the host computer.

Some Linux distributions provide a choice of yacc programs. One can install either (or both) Berkeley yacc or the GNU **bison** program. You'll probably find them in /usr/bin. They are quite similar; bison was originally derived from yacc, although there has been some divergence over the years.

The combination of lex, yacc and some programmer's C code provides a complete means to interpret and act upon a user's wishes. The lex program uses its regular expression interpretation capability to recognize strings of characters as words or tokens. (The term “words” is used loosely to describe any recognized string of characters.) Once a token is identified, it is passed to yacc where the various rules are applied until some combination of tokens form a recognizable structure to which yacc applies some pre-programmed C code.

How The Tools Are Used

As indicated, lex uses regular expressions to recognize strings of characters as items of interest. Regular expressions are composed of special characters which describe acceptable combinations of characters.

For example, regular expressions often use the character . (period) to indicate that *any* character except a newline (\n) is acceptable.

Similarly, the characters [and] (square brackets) are used to indicate acceptance of any of the characters enclosed within them or within the range of characters described between them. For example, the expression **[abc]** says to accept *any* of the characters a, b or c; the expression **[a-c]** says the same thing. A more complicated example might be **[a-zA-Z0-9]** which says to accept any alphanumeric character.

For a complete description of lex regular expression syntax, see *lex & yacc* by Levine, Mason and Brown (O'Reilly, 1992).

Once a regular expression matches the text stream being interpreted by lex, code created by the programmer is executed. When used with yacc, this code generally amounts to passing an indication of what was just recognized to yacc for further processing. The indication is a token that yacc knows about, and in fact, these are defined in the yacc portion of the analyzer/parser program so that they are common to both lex and yacc.

Also as indicated, yacc uses a grammar description to decode the meaning of the tokens that lex passes to it. As tokens are passed to yacc, it applies its rules until a single token, or some sequence of tokens, becomes a recognizable structure.

Before a programmer's C code is executed, though, yacc may require several structures or token-structure combinations to be recognized. For example, using our sample sentence, our rules might look like the following:

```
sentence  :  subject + predicate
{...execute some C code...}
subject       :  noun
              |  noun_phrase
predicate     :  verb + noun_phrase
noun_phrase   :  preposition + adjective + noun
              |  adjective + noun
```

The first rule says that a sentence is made up of two parts: a subject followed by a predicate. If that rule is satisfied, then the C code between the curly brackets will be executed. To satisfy the first rule, yacc has to recognize both a subject and a predicate. The subsequent rules help yacc to do just that.

For example, the second rule says that a subject is recognized when either a noun or a noun phrase is recognized. A noun is the smallest unit that yacc deals with, and in the yacc grammar, a noun is a token that yacc will want to have lex recognize. Thus, somewhere in the yacc program, a token will be defined (probably called NOUN) that lex and yacc will use to communicate the fact that a noun has been interpreted. How this is done we will see shortly.

Notice that a noun phrase is also used to create the predicate. If a verb is recognized and it is followed by a noun phrase, the predicate is identified. If only the noun phrase is identified, then the subject has been identified.

The example cited is not in yacc syntax, but is meant to provide understanding. Very detailed examples may be found in the references.

You may be wondering how the yacc parser actually works. **yacc** works as a finite-state machine, and it has a stack (think of this as a memory of what has been seen, as it tries to deduce what the incoming stream of tokens represents).

A finite-state machine records its current condition as each recognizable item is interpreted. For example, as a noun phrase is being interpreted, it moves from state 3 when it receives a preposition to state 4 when the adjective is interpreted and finally to state 5 when the noun is recognized. When the entire phrase has been recognized, it switches to another state, perhaps 37, to note that fact. Please do not attach any particular meaning to the numbers used in this example. They have been chosen arbitrarily to demonstrate how yacc progresses as it interprets the tokens received from lex. You should conclude only that to reach state 5 (noun phrase), yacc must progress through several preceding states, each of which might lead to another final state, depending on the grammar yacc is using.

In other words, given its current state, yacc requests from lex the next token (if it needs it) and places onto the stack its new state. In doing so, it may push the new state onto the stack (as when interpreting the noun phrase), or pop the old state off the stack, replacing it with a new state (as when the noun phrase is completely recognized). These actions are called “shift” and “reduce” and describe pushing and popping states to and from the stack.

When the sentence is finally recognized, yacc accepts it and returns to the calling program (the main program which invoked yacc and indirectly lex). For a complete description of how a yacc parser works, see *Inside Xenix* by Christopher Morgan, Howard W. Sams and Co., 1986. This reference describes yacc grammars and how yacc parses its input in exquisite detail.

Basic Coding of lex and yacc Programs

Both tools are coded in a similar manner. There are three sections in each program: declarations, rules and user routines. Each is separated by a line containing only the two characters **%%**.

For yacc, the declarations section contains the tokens it can use while parsing the input. Each has a unique value greater than 256, and the set of tokens is introduced via **%token** at the beginning of the line. **lex** can use the declarations section to define aliases for items that it must recognize while looking for tokens to pass to yacc.

For example, lex needs to know about white space which, while not used in identifying tokens, must be accounted for in some way. Similarly, mathematical symbols such as + or = must be recognized. These are needed in the interpretation of the algebraic statement coded by the user.

Within the rules section, yacc holds its parsing intelligence. This is the section that contains the grammar rules referred to in the English sentence example earlier. In fact, the coding used earlier is typical of a yacc grammar: items to be recognized are separated from the means to recognize them by a colon (:), and alternative means of recognition are separated from each other via a pipe (|) symbol.

lex uses the rules section to contain the regular expressions that allow it to identify tokens to pass to yacc. These expressions may be the aliases from the declaration section, regular expressions, or some combination.

The last section contains C code which may be invoked as each of the tools processes its input.

One requirement is that the yacc tokens be known to the lex program. This is accomplished by including the following statement:

```
#include "y.tab.h"
```

in the lex declarations section and creating it when compiling the yacc program code.

Compilation is accomplished in the following way:

```
yacc -d yacc.file -create 'y.tab.c and y.tab.h'
flex flex.file -create 'lex.yy.c'
```

The **-d** option on yacc's command line creates the y.tab.h file needed by lex.yy.c.

How lex and yacc were employed in Log Analysis

To successfully interpret the user's desired process, the program needs to know which well logs were available for processing. This information is available in the ASCII file selected by the user. This text file contains a one-to-one correspondence between curve description and data values. A very small subset of a typical file is shown in Listing 1.

Listing 1

As can be seen, there are several sections including well information (includes some hole parameters), curve information (notes which curves are in the file) and “A” which holds the recorded data values. Each is introduced with a tilde (~). Because the format of the file is fixed by convention, these are easily parsed, and needed values are stored for subsequent processing.

As written for the client, the program is a Motif application. The user selected the file to be processed; it was read in its entirety and numeric text items were converted to double-precision values.

Besides allowing file and curve merging and editing, there is a menu item for curve processing. Upon selecting this menu item, a dialog box is presented containing a list of available curves and arithmetic operations. The user selects curve names, numeric constants and operations which in turn are displayed as an algebraic operation on a text input line. When satisfied with the mathematical operation, the user clicks **OK** and the lex and yacc magic occurs. The result is stored as a separate curve and can be included in subsequent operations.

lex processed the incoming algebraic statement with the code shown in Listing 2.

Listing 2

Between lines 1 and 16 are declarations to be used in the program generated by lex. In particular, you will notice the inclusion of the header file y.tab.h which contains the following definitions:

```
#define INTEGER 257
#define FLOAT 258
#define DOUBLE 259
#define NUMBER 260
#define VARIABLE 261
#define EQUAL 262
#define LPAREN 263
#define RPAREN 264
#define PLUS 265
#define MINUS 266
#define TIMES 267
#define DIVIDE 268
#define RAISE 269
#define LHS 270
```

These definitions are used by both lex and yacc to describe the items yacc expects to receive from lex. They are generated by statements 73 to 77 of the yacc source which will be examined shortly.

From lines 17 to 31 of the lex listing are declarations which amount to aliases for common items that we wish lex to recognize. For example, we declare DIGIT to be any single numeric between 0 and 9 on line 21. Doing this allows us to declare INT (an integer) to be one or more DIGIT's.

Lines 33 to 90 contain the rules by which lex interprets incoming text items. For example, on line 34 we recognize an equal sign (=) and return the token EQUAL to the caller. In y.tab.h, EQUAL is defined to be 262.

As you can see, the lex rules simply recognize text items and then advise the caller what was seen in the input stream.

Listing 3

yacc interprets the token stream passed to it by lex with the following code, only a subset of which is shown in Listing 3. The code for the yacc routine (with the calling subroutine **do_arithmetic** and its accessory functions) was in excess of 900 lines. For those interested, it is available for your perusal from SSC's public FTP site. Listing 3 is a sample indicating what needed to be done.

Like the lex routine, yacc begins with lines to be included in the output code. Programs written for graphical user interfaces sit in a loop waiting for the user to click on something. When the user's needs are so indicated, the GUI-based program calls a function to perform the required action. These “called functions” are popularly called *callbacks*. In this program, one of the callbacks was **do_arithmetic**, which in turn called the yacc routine, which in its turn called the lex routine.

In Listing 3, do_arithmetic is described in the first few lines, and a portion of the code may be seen in lines 428 to 532. They are shown only to give some indication of what was being accomplished.

yacc does the work with its rules section beginning at line 79, and ending at line 426. Although too long to be included completely, you can see that an *equation* is defined to be something called an **lhs** (left hand side) **EQUAL rhs** (right hand side) at line 80. Looking down the listing, you will see that an equation may also be described by an **expr** (expression). When either of these are encountered, yacc pops a key from an internal stack created by a function called **push** (see near line 557) and then causes a log curve to be returned to the caller by calling another function called **get_curve** (not shown here, but included with the yacc and lex code).

Between lines 118 and 139, you can see how some of the tokens yacc expects are processed when recognized. The complete listing has many more.

Results

The lex, yacc and supporting code was successfully employed to allow the log analysts to process various log curves. To have written the C code to accomplish the lexical analysis and parsing logic would have taken much longer than the four weeks allowed. As it turned out, this code was much easier to create and debug than it was to introduce into the final Motif application, even though it was written as a callback.

In fact, the number of lines of lex (152) and yacc (953) code were far fewer than the number of lines generated by the two (2765). Of course, one could take the time to write much tighter code than these general purpose tools deliver.

Nevertheless, should you be faced with a similar problem, I strongly recommend using lex and yacc. They are powerful, reliable tools worth knowing.

All listings referred to in this article are available by anonymous download in the file ftp://ftp.linuxjournal.com/pub/lj/listings/issue51/2227.tgz.

Dean Provins (provinsd@cuug.ab.ca) is a professional geophysicist and licensed amateur radio operator (VE6CTA) in Calgary, Alberta. He has used UNIX systems since the mid-1980s and Linux since January, 1993. Dean uses Linux as a development system for geophysical software, and as a text processing system for a newsletter and other articles. He is currently enrolled as a graduate student in Geomatics Engineering at the University of Calgary

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

New Products

Amy Kukuk

Issue #51, July 1998

STREAMS Data Comm Protocols, Raven SSL Module for Apache, Java Workshop 2.0 for Linux and more.

STREAMS Data Comm Protocols

Gcom, Inc. has announced the Linux STREAMS application of their BASS line of packaged synchronous adapter and software systems. The Gcom kits allow the addition of synchronous capability for a PC platform. Protocols include X.25, SNA, Frame Relay, SDLC, HDLC, LAPD, LAPB, QLLC and Bisync. The high-performance communications board delivers line speeds of up to 5Mbps and offers ISA or PCI bus design and many interface options. Prices start at $2,288 US for the end user. Gcom is the developer of Linux STREAMS.

Contact: Gcom, Inc., 1800 Woodfield Drive, Savoy, IL 61874, Phone: 217-351-4241, E-mail: sales@gcom.com, URL: http://www.gcom.com/.

Raven SSL Module for Apache

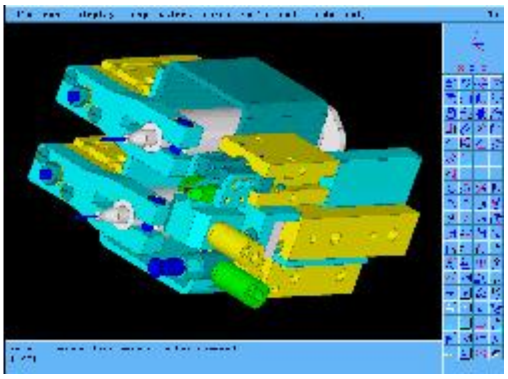
Covalent Technologies has announced the release of Covalent Raven, an SSL module for the Apache web server. Raven functions can be used for secure Internet transactions, including on-line banking, credit card purchases, safe document transfers and more. Purchase price for Raven is $357 US.

Contact: Covalent Technologies, Inc., 121 South 13th Street, Suite B-105, Lincoln, NE 68508, Phone: 402-441-5710, Fax: 402-441-5720, E-mail: info@covalent.net, URL: http://www.covalent.net/.

Java Workshop 2.0 for Linux

S.u.S.E. has announced Java Workshop 2.0 for Linux. This product brings Java Workshop's visual development tool for Java programmers to the Linux community. Java Workshop allows developers to use the Java platform to create leading-edge Internet and Intranet applications. It offers a tool set for building JavaBeans, Java applets and applications. Java Workshop also allows developers to create and reuse JavaBeans. It includes support for the latest JDK and comes with a compiler and profiler. Java Workshop for Linux is available at a list price of $109 US.

Contact: S.u.S.E., LLC, 458 Santa Clara Avenue, Oakland, CA 94610, Phone: 1-510-835-7873, Fax: 1-510-835-7875, E-mail: info@suse.com, URL: http://www.suse.com/.


VariCAD

VariCAD has announced its Mechanical CAD software—VariCAD. VariCAD is equipped with all the basic tools necessary for mechanical design. It includes true 3-D modeling optimized for construction and design, allowing one to create, evaluate and modify a concept any way the user wants. Its other functions include 2-D drawing, editing, transformations, working with user-defined objects, blocks, groups and symbol-creation capabilities. The whole system is customizable and easy to use. The price for VariCAD varies between $100 and $2,000, according to platform and support.

Contact: VariCAD, P.O. Box 38, Liberec 460 02, Czech Republic, E-mail: mail@varicad.com, URL: http://www.varicad.com/.

Cobalt Qube Microserver

Cobalt Microserver Inc. has announced the Cobalt Qube microserver. The Qube microserver supports communication and collaboration services for the Internet and Intranets. It features quick setup and “hands-off” administration. With a suggested starting price of $999 US, the Qube microserver is aimed at work groups and branch offices, Internet Service Providers, Web developers and educational organizations. It provides basic services such as e-mail, web publishing and file sharing as well as other services such as threaded discussions and automated searching and indexing. The Qube microserver has a complete Linux 2.0 operating system and includes the Apache web server.

Contact: Cobalt Microserver, Inc, 440 Clyde Avenue, Building B, Mountain View, CA 94043, Phone: 650-930-2500, Fax: 650-930-2501, E-mail: sales@cobaltmicro.com, URL: http://www.cobaltmicro.com/.

Screamer 633MHz Custom Workstations

Microway has announced the Screamer 633MHz motherboard and custom workstations. These products deliver performance in such areas as CAD/CAM/CEA, 3-D rendering, animation and multimedia. The Screamer motherboard design offers four megabytes of cache and features the Samsung 633MHz Alpha processor. For pricing details, look at the company's web site.

Contact: Microway, Inc., Research Park, Box 79, Kingston, MA 02364, Phone: 508-746-7341, Fax: 508-746-4678, E-mail: info@microway.com, URL: http://www.microway.com/.

NetTracker Proxy 3.5

Sane Solutions, LLC has announced the release of NetTracker Proxy 3.5. NetTracker Proxy is a web-based proxy and firewall log file analysis software. Priced at $795 US, NetTracker Proxy 3.5 contains new features including standardized summary reports with drill-down capabilities, as well as sorting capabilities that allow administrators to select and sort the information they analyze. It can export data, which allows administrators to import NetTracker Proxy reports into popular software products.

Contact: Sane Solutions LLC, 35 Belver Ave., Suite 230, North Kingstown, RI 02852, Phone: 401-295-4809, E-mail: info@sane.com, URL: http://www.sane.com/.

WebSite Professional 2.2

O'Reilly & Associates has announced the release of WebSite Professional 2.2, which includes Uplink, O'Reilly's utility designed for Internet Content Providers and Internet Service Providers. Another new feature is enhanced log file management and generation. The inclusion of Live Software's new JRun 2.1 adds support for the Java Development Kit version 1.2 of the JavaSoft Servlet Advanced Programming Interface 1.1. Suggested list price is $799 US. The upgrade to version 2.2 is free for downloading by registered version 2.0 and 2.1 customers.

Contact: O'Reilly & Associates, 101 Morris Street, Sebastopol, CA 95472, Phone: 707-829-0515, E-mail: software@oreilly.com, URL: http://software.oreilly.com/.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Miscellaneous Character Drivers

Alessandro Rubini

Issue #51, July 1998

Alessandro tells us how to register a small device needing a single entry point with the misc driver.

Sometimes people need to write “small” device drivers, to support custom hacks—either hardware or software ones. To this end, as well as to host some real drivers, the Linux kernel exports an interface to allow modules to register their own small drivers. The **misc** driver was designed for this purpose. The code introduced here is meant to run with version 2.0 of the Linux kernel.

In UNIX, Linux and similar operating systems, every device is identified by two numbers: a “major” number and a “minor” number. These numbers can be seen by invoking **ls -l /dev**. Every device driver registers its major number with the kernel and is completely responsible for managing its minor numbers. Use of any device with that major number will fall on the same device driver, regardless of the minor number. As a result, every driver needs to register a major number, even if it only deals with a single device, like a pointing tool.

Since the kernel keeps a static table of device drivers, frivolous allocation of major numbers is rather wasteful of RAM. The Linux kernel, therefore, offers a simplified interface for simple drivers—those that will register a single entry point. Note that, in general, allocating the whole name space of a major number to every device is beneficial. This allows the handling of multiple terminals, multiple serial ports and several disk partitions without any overhead in the kernel proper: a single driver takes care of all of them, and uses the minor number to differentiate.

Major number 10 is officially assigned to the **misc** driver. Modules can register individual minor numbers with the misc driver and take care of a small device, needing only a single entry point.

Registering a Minor Number

The misc driver exports two functions for user modules to register and unregister their own minor number:

```
#include <linux/miscdevice.h>
int misc_register(struct miscdevice * misc);
int misc_deregister(struct miscdevice * misc);
```

Each user module can use the register function to create its own entry point for a minor number, and deregister to release resources at unload time.

The miscdevice.h file also declares **struct miscdevice** in the following way:

```
struct miscdevice {
        int minor;
        const char *name;
        struct file_operations *fops;
        struct miscdevice *next, *prev;
};
```

The five fields have the following meaning:

- **minor** is the minor number being registered. Every misc device must feature a different minor number, because such a number is the only link between the file in /dev and the driver.
- **name** is the name for this device, meant for human consumption: users will find the name in the /proc/misc file.
- **fops** is a pointer to the file operations which must be used to act on the device. File operations have been described in a previous “Kernel Korner” in April 1996. (That article is available on the web at http://www.linuxjournal.com/issue24/kk24.html.) Anyway, the topic is refreshed later in this article.
- **next** and **prev** are used to manage a circularly-linked list of registered drivers.

The code calling **misc_register** is expected to clear **prev** and **next** before invoking the function and to fill the first three fields with sensible values.

The real question with the misc device driver is “what is a sensible value for the **minor** field?” Assignment of minor numbers is performed in two ways: either you can use an “officially assigned” number, or you can resort to dynamic assignment. In the latter case, your driver asks for a free minor number, and the kernel returns one.

The typical code sequence for assigning a dynamic minor number is as follows:

```
static struct miscdevice my_dev;
int init_module(void)
```

```
{
    int retval;
    my_dev.minor = MISC_DYNAMIC_MINOR;
    my_dev.name = "my";
    my_dev.fops = &my_fops;
    retval = misc_register(&my_dev);
    if (retval) return retval;
    printk("my: got minor %i\n",my_dev.minor);
    return 0;
}
```

Needless to say, a real module will perform some other tasks within **init_module**. After successful registration, the new misc device will appear in /proc/misc. This informative file reports which misc drivers are available and their minor numbers. After loading *my*, the file will include the following line:

```
63 my
```

This shows that 63 is the minor number returned. If you want to create an entry point in /dev for your misc module, you can use a script like the one shown in Listing 1. The script takes care of creating the device node and giving it the desired permission and ownership.

You might choose to find an unused minor number and hardwire it in your driver. This would save invoking a script to load the module, but the practice is strongly discouraged. To keep the code compact, drivers/char/misc.c doesn't check for duplication of minor numbers. If the number you chose is later assigned to an official driver, you'll be in trouble when you try to access both your module and the official one.

If the same minor number is registered twice, only the first one will be accessible from user space. Although seemingly unfair, this can't be considered a kernel bug, as no data structure is corrupted. If you wish to register a safe minor number, you should use dynamic allocation.

The file Documentation/devices.txt in the kernel source tree lists all of the official device numbers, including all the minor numbers for the misc driver.

Kernel Configuration

If you have tried to write your own misc driver but *insmod* returned **unresolved symbol misc_register**, you have a problem in your kernel configuration.

Originally, the misc driver was designed as a wrapper for all the “busmouse” drivers—the kernel drivers for every non-serial pointer device. The driver was only compiled if the current configuration included at least one such mouse driver. Just before version 2.0, the generality of the implementation was widely accepted, and the driver was renamed from “mouse” to “misc”. It is still true, however, that the driver is not available unless you chose to compile at least one of the misc devices as either a module or a native driver.

If you don't have any such devices installed on your system, you can still load your custom misc modules, provided you reply affirmatively, while configuring your kernel, to the question:

```
  Support for user misc device modules (CONFIG_UMISC)
```

This option indicates that the misc driver is to be compiled even if no misc device has been selected, thus allowing run-time insertion of third-party modules. The file /proc/misc and support for dynamic minor numbers were implemented when this option was introduced, as there's little point in having custom modules unless the allocation of a minor number is safe.

Note that if your kernel is configured to load busmice only as modules, everything will work with the exception of /proc/misc. The /proc file is created only if miscdevice.c is directly linked in the kernel. **CONFIG_UMISC** takes care of this situation as well.

How Operations are Dispatched

Every time a process interacts with a device driver, the implementation of the system call gives control to the correct driver by means of the **file_operations** structure. This structure is carried around by **struct file**: every open file descriptor is associated to one such structure, and **file.f_op** points to its own file_operations structure.

This setup is similar to object-oriented languages: every object (here, every file) declares how to act on itself, so that high-level code is independent of the actual file being accessed. The Linux kernel is full of object-oriented programming in its implementations, and several “operations” structures exist in it, one for each different “object” (inodes, memory regions, etc.).

Back to the misc driver. How does **my_dev.fops** participate in the game? At open time, the kernel allocates a new **file** structure to describe the object being opened, and initializes its operations structure according to what the file is. Sockets, FIFOs, disk files and devices get their own, different, operations. When a device is opened, its operations are looked up according to the major device number by referencing an array. The **open** method within the driver is then called. Any other system call that acts on a file will then use **file.f_op** without checking any other source of information. As a result, a driver can replace the value of **file.f_op** to tailor the behaviour of a **struct file** to some inner feature, even if that feature is at a finer grain than the major number, and thus is not visible from the kernel proper.

The open method of the misc driver is able to dispatch operations to the actual low-level driver by modifying **file.f_op**; the assigned value is the one in **my_dev.f_op**. After the operations have been overridden, the method calls **file.f_op->open()**, so that the low-level driver can perform its own initialization. Every other system call invoked on the file will use the new value of **file.f_op**, and the low-level driver keeps complete control over its device.

An Example: Keyboard LEDs

Since the discussion up to now has been much too philosophical, it's time to move to a working example. The *kiss* module (Keyboard Informative Status Signals) is a simple tool to play with the keyboard LEDs. It registers itself as a misc device using dynamic minor-number assignment and is controlled by writing textual commands to it. It accepts several one-byte commands, such as “N” and “n” (to turn the Num-Lock LED on and off), the digits from 0 to 7 (to display binary numbers in that range using the LEDs) and so on.

I don't think there's any need to include source code here, as the driver does little more than the **misc_register** code shown above. Most of the additional code deals with interpretation of the commands and actual lighting of the LEDs. The tar file with source code and a **README** file can be retrieved from ftp://ftp.linuxjournal.com/pub/lj/listings/issue51/2920.tgz.

As usual, the sample module that accompanies this article is pretty simple and is of little interest in the real world. This time, however, I think it can be of some interest. As a matter of fact, custom hardware in my computer includes three LEDs to monitor the number of running processes. In my opinion, this is useful information when you are wondering why the computer is not responding—a situation quite common whenever you write buggy drivers or drivers that print too many diagnostic messages.

Alessandro is still using Linux 2.0, because he's spending his spare time building his own misc devices with a soldering iron. He enjoys open source and open air, and reads e-mail as rubini@linux.it.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

The Yorick Programming Language

Cary O'Brien

Issue #51, July 1998

Yorick is an interpreted language for numerical analysis used by scientists on machines from Linux laptops to Cray supercomputers.

Linux leverages a vast amount of academic software, either easy ports of existing UNIX packages or, increasingly in recent years, software that is ready to run under Linux. One example is Yorick, and this article is an attempt to provide a brief overview of the nature and capabilities of this system.

Yorick is not just another calculator. Readable syntax, array notation and powerful I/O and graphics capabilities make Yorick a favorite tool for scientific numerical analysis. Machine-independent I/O, using the standard NetCDF file formats, simplifies moving applications between hardware architectures. Yorick is an interpreted language developed by David H. Munro at Livermore Labs. Implemented in C, it is freely distributed under a liberal copyright. Yorick runs on a vast range of machines, from 486SX Linux Laptops (in my case) to Cray YMP supercomputers.

Who uses Yorick? The majority of users are physicists, many with access to the most powerful computers in the world. Specific applications include Astrophysics, Astronomy, Neurosciences, Medical Image Processing and Fusion Research.

In this article I will discuss the basics of running Yorick, describe the key array operations, and briefly discuss array operations, programming and graphics. I hope that this quick look is enough to get the more mathematically inclined readers to give Yorick a try.

Basic Operations

When invoked without arguments, Yorick presents a typical command-line interface. Expressions are evaluated immediately, and the result is displayed.

Primitive types include integers, floating-point values and strings. All the built-in functions and constants you would expect to be present are present. Variable names are unadorned, with no leading $ character and need not be pre-declared. C-style comments are supported.

One might not expect an interpreted language to be suitable for numerical analysis, and indeed, this would be the case if arrays were not built into the language. Arrays are first-class objects that can be operated on with a single operation. Since the virtual machine understands arrays, it can apply optimized compiled subroutines to array operations, eliminating the speed penalty of the interpreter.

Arrays can be created explicitly:

```
> a1 = [1.1, 1.2, 1.3, 1.4, 1.5]
```

Elements can be accessed singly or as a subset, with **1** being the origin. Parentheses indicate the indexing operation, and a single index or a range of indices can be specified.

```
> a1
[1.1,1.2,1.3,1.4,1.5]
> a1(2)
1.2
> a1(1:3)
[1.1,1.2,1.3]
```

Since array operations are built into the language, functions applied to the array are automatically applied to all elements at once.

```
> sqrt(a1)
[1.04881,1.09545,1.14018,1.18322,1,2.23607]
```

Arrays are not limited in dimension. The rank (number of indices) of an array is not limited to one (a vector) or two (a matrix), but can be as large as desired. Arrays of rank 3 can be used to represent the distribution of a parameter across a volume, and an array of rank 4 could model this over time.

Yorick also provides a simple but effective help system. Executing the help command describes the help system. Executing it with a command name as an argument provides information on that command.

Yorick provides a complete programming language that closely matches C in terms of control flow, expressions and variable usage. For example, the statement:

```
> for(i=1; i<10; i++) { print,1<<i; }
```

will print the powers of two just as you would expect. Function declarations, introduced with **func**, also work as expected:

```
> func csc(x) {
> return 1/sin(x);
> }
```

There are differences—variables need not be declared, and arrays are much more powerful than in C. The major difference is in function invocation. Passing arguments to a function in parentheses causes an evaluation and printing of the result; however, passing arguments separated by commas simply executes the function and does not return the result. Since in most cases intermediate results are not required, many scripts contain function calls of the form f,x,y rather than the more familiar f(x,y).

Having a programming language close to C allows easy migration between Yorick for prototyping and C for final implementation. However, as several Yorick users have indicated, moving to C is often unnecessary—the Yorick program proved to be fast enough to get the job done with a minimum of programming effort.

If C extensions are required, a straightforward framework allows extending the Yorick command language with whatever new operations are necessary.

Advanced Array Operations

Yorick has a compact and sophisticated mechanism for describing array indexing and operations, which are used to precisely specify the desired operation to the interpreter. Applying an operation to an array causes the operation to be applied to each element of the array. For example:

```
> a = [1,2,3,4,5]
> sqrt(a)
[1,1.41421,1.73205,2,2.23607]
```

What about multiplying two vectors? The default is to perform an element by element multiplication.

```
> b = [2,4,6,8,10]
> a*b
[2,8,18,32,50]
```

Those of you who remember physics or linear algebra will recall inner and outer products. The inner product is defined as the sum of the pairwise products:

```
> a(+)*b(+)
110
```

The outer product creates a matrix out of each possible multiplication:

```
> a(-,)*b(,-)
[[2,4,6,8,10],
 [4,8,12,16,20],
 [6,12,18,24,30],
 [8,16,24,32,40],
 [10,20,30,40,50]]
```

The **+** and **-** symbols, used where an index would be placed, are called special subscripts and provide precise control over how array operations are executed. The **+** is the matrix multiplication pseudo-index, which indicates to Yorick along which dimension the addition part of a matrix multiply should be performed. The **-** is a pseudo-index, creating an index where one did not exist before.

The rank-reducing operators **sum**, **min**, **max** and **avg** can be used in place of indices.

```
> a(max)
5
> b(avg)
6
```

One might wonder why this is necessary, when the equivalent function operators (i.e., **min()** or **avg()**) exist? The reason is that for matrices of rank 2 or greater, the rank-reducing index operators allow you to specify exactly how to perform the operation. For example, given a 3x3 array, do you want to average across rows, columns or the entire array?

```
> c = [[1,2,3],[4,5,6],[7,8,9]]
> dimsof(c)
[2,3,3]
> avg(c)
5
> c(avg,avg)
5
> c(avg,)
[2,5,8]
> c(,avg)
[4,5,6]
```

Here we have also introduced the **dimsof()** function operator, which reports the dimensions of the argument. In this case, the result tells us that **c** is an array of rank 2 with three elements in each direction.

Graphics Operations

Under Linux, Yorick is linked with the GIST graphics subsystem, allowing immediate display of plots and diagrams. Plots are interactive, allowing the user to zoom in and out, stretch axes, and crop the displays using the mouse. Yorick is capable of displaying sequences of plots over time as in a movie, and because of this, the command to prepare for a new image is **fma** or frame advance.

To plot the value of a function at evenly spaced points, we must first create the **x** values:

```
> x = span(0,10,256)
> dimsof(x)
[1,256]
```

x is now a 256-element array with values that range from 0 to 10.

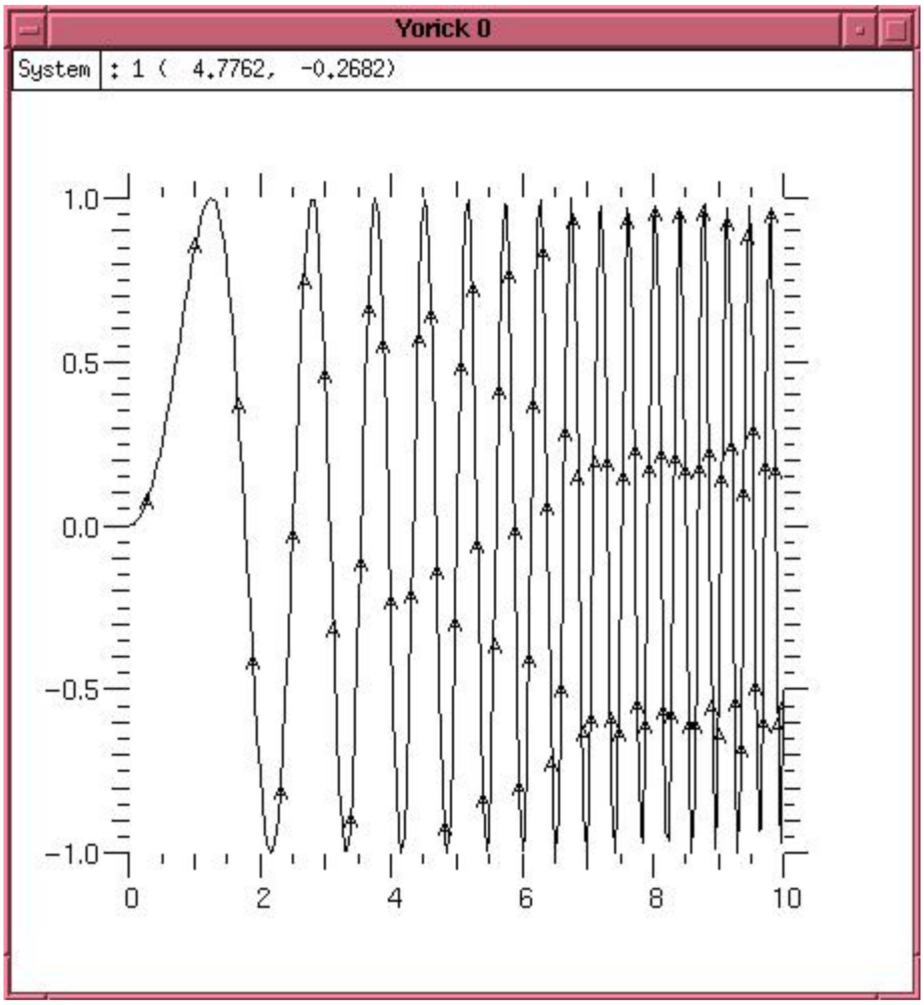


Figure 1. x-y Plot

The **plg** function, given vectors for the **x** and **y** values, plots x-y graphs.

```
plg, sin(x^2), x
```

The results of this command are shown in Figure 1. Note that the arguments are supplied **y,x** (not **x,y**). This allows Yorick to supply a default **x** vector (ranging from **1** to the number of **y** points), if desired.

Parametric plots are also supported. Consider the following commands which produced the spiral in Figure 2:

```
&GT; window, style="vgbox.gs"
&GT; a = span(0,20,256)
&GT; x = a * sin(a)
&GT; y = a * cos(a)
&GT; plg, y, x
```

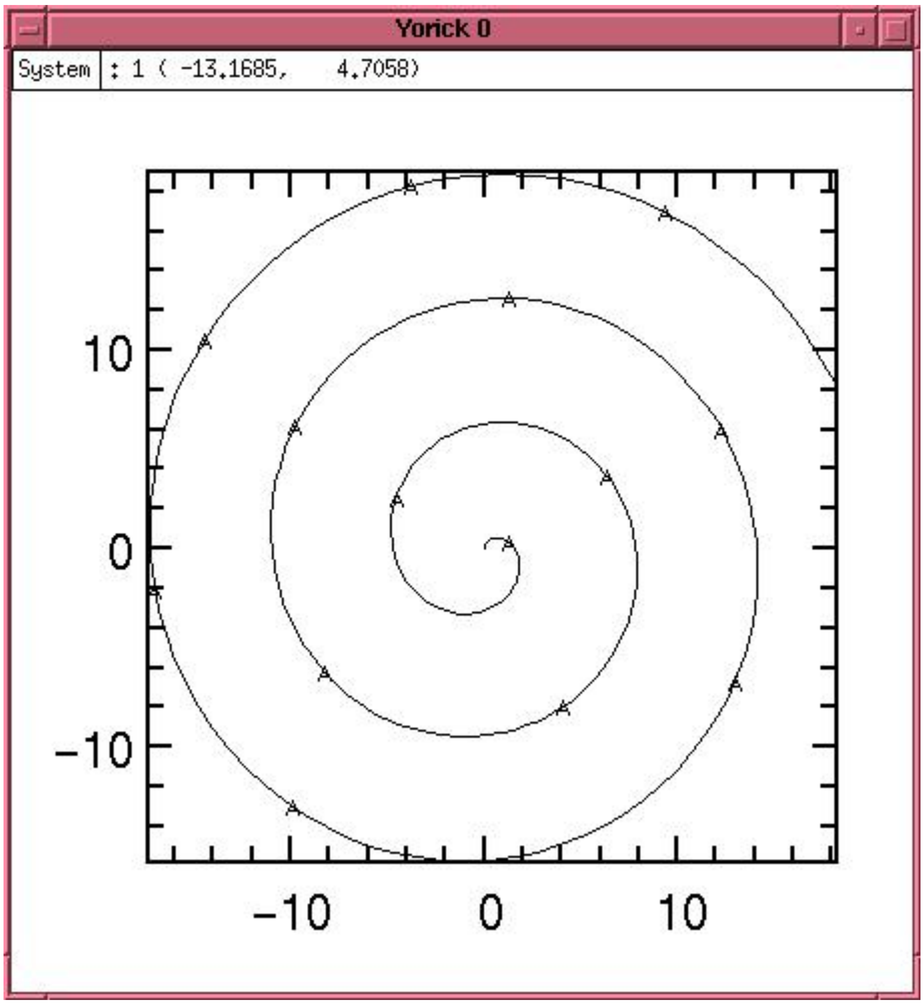


Figure 2. Spiral Plot

Surface plots are also available, either as a wire frame as in Figure 3:

```
&GT; #include "plwf.i"
> orient3
> x = span(-pi,pi,32)(,-:1:32)
> y = transpose(x)
> fma
> plwf, sin(x)*cos(y)
```

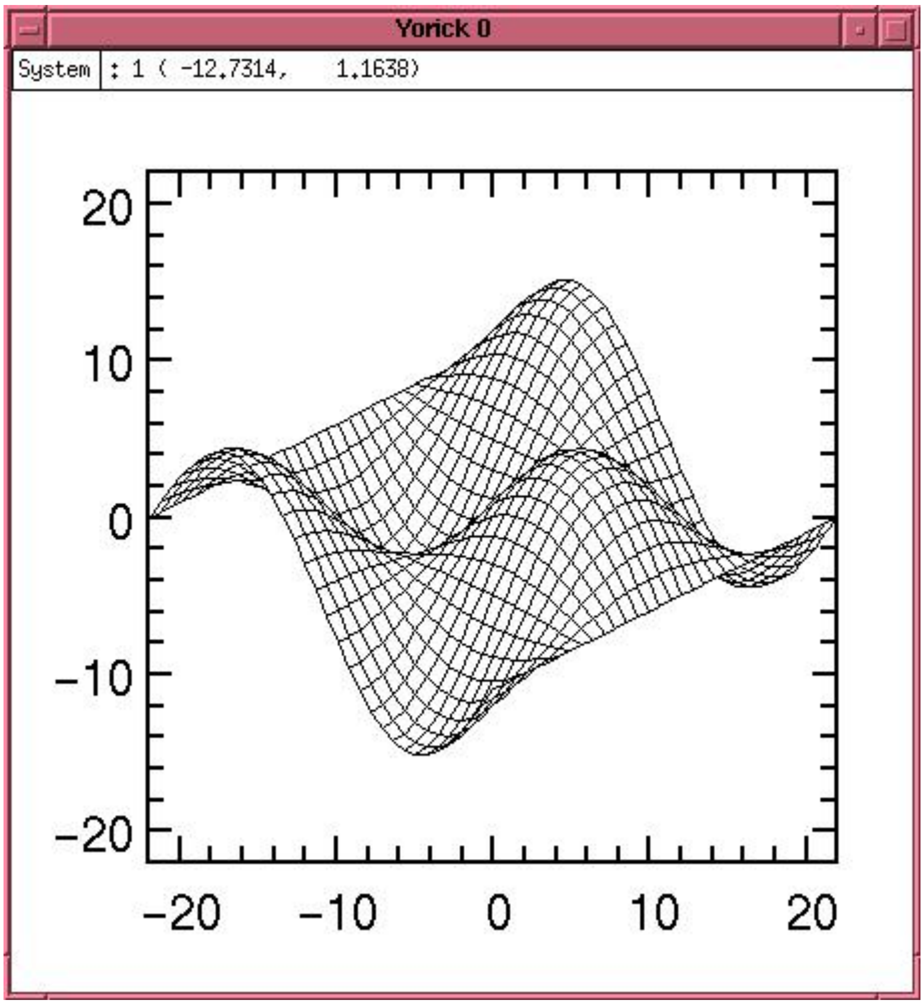


Figure 3. Wire Frame Surface Plot

Or a shaded surface rendition as in Figure 4:

```
> fma
> plwf, sin(x)*cos(y), shade=1, edges=0
```

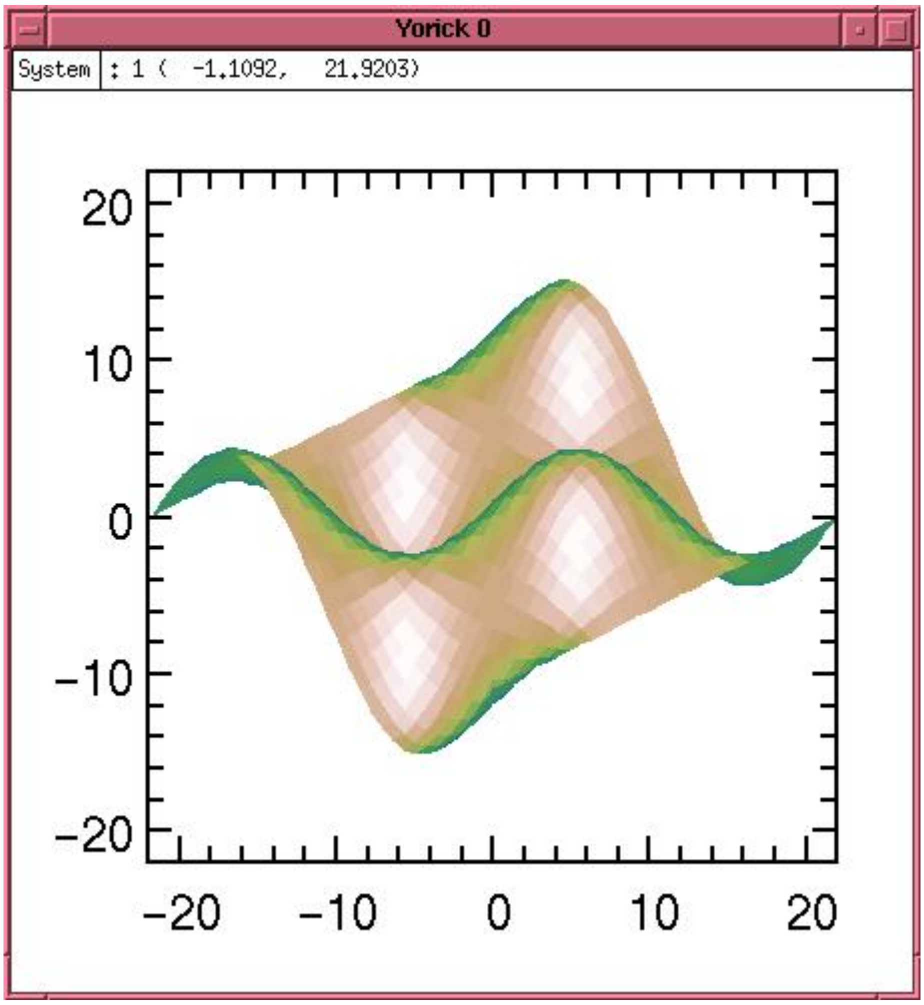


Figure 4. Shaded Surface Plot

A host of advanced graphics options are used in the demonstration programs distributed with Yorick, and the latest copy of the documentation has an extensive description of graphics options. In addition, libraries to read, write, and display PNM-format images are provided.

Closing Remarks

Yorick is an exceptionally rich environment for numerical analysis. Many of its capabilities such as file I/O, debugging, animation and distributed operation using MPY have not been explored in this article. Please take the time to read through the documentation and the example programs. You will not be disappointed.

Resources

This article was first published in Issue 26 of LinuxGazette.com, an on-line e-zine formerly published by Linux Journal.

Cary O'Brien (cobrien@access.digex.net) lives in Washington DC, and refers to himself, when pressed, as a “systems engineer”. He is currently Vice President of Optim Systems, Inc., which provides products and services to the telecommunications industry. He has been messing with computer hardware and software since high school. He is married with two children, 4 and 7, who are starting with computers even earlier.

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.

Best of Technical Support

Various

Issue #51, July 1998

Our experts answer your technical questions.

Unfinished Boot

When I try to boot, either from floppy or CD-ROM, my computer does a soft re-boot partway through the process. It doesn't even make it to the screen which asks if you have a color monitor, although I suspect it gets close to that point. I've tried running the boot disk on my kid's computer and it works fine. There is not a lot of difference between my computer and my kid's, but here are a few:

mine: Pentium 166	kid's: Pentium 133
mine: 96MB RAM	kid's: 32MB RAM
my video card: Creative's	kid's: Cirrus logic
	video blaster 3D
mine: 1GB drive + Syquest SparQ	kid's: 540MB drive + CD-ROM
	(IDE) + CD-ROM

Any thoughts? Thanks for any help you can give me.

—Bruce Matthews Red Hat 4.2

It sounds as if the kernel doesn't care for your video card. This is highly unusual, but it may be the case. Try a different video card to see if it helps. You might also try the boot disks from Red Hat 5.0 to see if a newer kernel helps with the problem. If so, you may have to use the newer version.

—Donnie Barnes redhat@redhat.com

Crashing XFree86

While using XFree86 3.3.1, it will crash unexpectedly and return the following message:

```
Fatal server error: Caught signal 4.
     Server aborting
X connection to :0.0 broken
     (explicit kill or server shutdown).
X connection to :0.0 broken
     (explicit kill or server shutdown). xinit:
connection to X server lost.
```

Sometimes, instead of **Caught signal 4**, it returns **11**, but the rest of the message is the same. This usually happens when I'm opening or using a program.

—Aaron Walker Red Hat 5.0

This may point to a faulty memory chip. Try stress testing your system, e.g., by compiling a huge package with optimization, such as your usual kernel compile. If a similar thing happens, it's the memory.

—Ralf W. Stephan stephan@tmt.de

This is usually indicative of a hardware problem. You should take a look at http://www.bitwizard.nl/sig11/ for information on how to determine if this is the case and how to fix it.

—Donnie Barnes redhat@redhat.com

Damage After a Crash

My /var partition filled to 100% recently (I installed some alpha software that bombed 40MB of logs), and basically, I crashed. On reboot, there was damage to the file systems, so I know some things are broken.

My <path>/lost+found directories contain some chunks of data after running **fsck**, and I assume I can do something to discover what has been damaged and reinstall those packages. How do I do that?

The specific problems I'm now having are:

1. xdm launches the xserver, but the login screen does not come up and the background graphic is a very large text “Red Hat Linux”. I think this is probably related to #2.
2. bash (and other shells) can't seem to find/execute scripts. **ls** sees them and **vi** edits them, but bash says it can't find them on execute attempts. Thanks in advance.

—Rob Collins Red Hat 5.0

First, you can go to the lost+found directories and use the **file** command to see what type of data they are. Then use an appropriate viewer to look at the file. For ASCII text you can use **more**, **less**, etc. For files of type “data”, a good way to figure out what they are is to run **strings** ***`filename`*** **| less** to look at any strings that appear in the file. Those strings may yield some clues.

The next thing you'll want to do is run **rpm -Va** on your system. That will tell you about any files existing in your RPM database that have changed in any way. Some of them are normal (things in /dev, for example), but it should be easy to tell what else has changed. Look at the man page for RPM (**man rpm**) for an explanation of the Va output. Once you find files that have changed or are missing, you will want to fix them. The best way is to reinstall the package completely.

Both of your problems will probably be fixed by going through the above steps.

—Donnie Barnes redhat@redhat.com

Interlace Mode

How do I take my monitor out of interlaced mode? I run it in non-interlaced mode in Windows 95, so I know it can run non-interlaced. I use the FVWM window manager. Thanks.

—Cliff Slackware 2.3

In order to change the mode in which X will be running, you need to change the XF86config file, generally located in the /etc/X11 directory. This file contains important information about your X server, such as the horizontal/vertical frequencies supported by your monitor. Editing this file by hand can be tricky, so I suggest you:

1. Back up your running XF86config. (Do a **find / -name XF86config** to discover the correct location.)

2. Run XF86config and change the desired features. Be very careful with the horizontal/vertical frequencies.
3. Type **startx** at the prompt.

—Mario de Mello Bittencourt Neto mneto@buriti.com.br

Stuck with Multi-serial Port Troubles

I bought an HP NetServer Pro (Pentium Pro) to run a multi-serial port (DigiBoard Xem 16 RS232 ports). I loaded in the 2.0.30 kernel sources from InfoMagic and recompiled the kernel to recognize the DigiBoard. I used a boot disk to boot-up Linux, but it didn't work. Each time, it hangs, and the error message returned is “vfs: kernel panic... etc.”

The original kernel on the boot disk works, but any recompilation of the kernel causes that error message. The DigiBoard module is loaded properly before the error appears.

I need to set the machine up urgently, and yet I'm stuck. I need help. Thank you.

—Weng-Yue Boey InfoMagic 2.0.30

Unfortunately, what you've described seems to be a hardware problem (bad cache, bad memory). You say that you can run any precompiled kernel, but if you try compiling yours (or any other huge program), you end up with a “got signal 11...” message.

I would suggest turning off the cache or removing some of the memory, then try compiling once more.

—Mario de Mello Bittencourt Neto mneto@buriti.com.br

Firewall Troubles

I am setting up a firewall and masquerade at my office to service approximately fifty workstations. The masquerade will be used to allow multiple users to access the Internet with a private IP addressing scheme (10.0.0.0\8), and the firewalling is for added security. I have successfully implemented masquerading in both a test and production environment; I have been successful with the firewall only in a test environment.

The problem in the test environment is when I change to the /sbin/init.d directory and attempt to execute **firewall list**, I get a “command not found” error. The firewall script is present in that directory. **firewall start** and **firewall stop** also will not execute.

Any suggestions on where to go next would be appreciated. Thanks,

—Doug Ford S.u.S.E. 5.0

It sounds like your current directory isn't included in your PATH. You can either set your PATH to include the current directory (generally a bad idea for root) or just prepend the command you want to run with **./**, for example, **./firewall stop**.

—Mark Bishop mark@bish.midwest.net

Adding an Ethernet Driver to the Kernel

My Ethernet card was installed improperly. How do I add a new Ethernet driver into the kernel? I mean, how do I recompile the kernel to include the new driver?

—John Liu Slackware 2.0.29

As root, go into /usr/src/linux, run **make menuconfig** and when you're done, **make zImage**. The new kernel is then in /usr/src/linux/arch/i386/boot/. Put it somewhere else, such as the / directory, preferably with a new name, then configure and run LILO. (You should keep your old kernel in lilo.cfg in case the new one has problems.) Reboot.

—Ralf W. Stephan stephan@tmt.de

Editing motd and issue

How do I edit the motd and issue files? Do I need to be in single-user mode? How do I get into single-user mode?

—Scott Slackware

Edit the file /etc/issue or /etc/motd with your favorite editor. No, you don't need to be in single-user mode.

—Mark Bishop mark@bish.midwest.net

Archive Index Issue Table of Contents

Advanced search

Copyright © 1994 - 2019 *Linux Journal*. All rights reserved.