77

AIX update

March 2002

In this issue

AIX Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38342
From USA: 01144 1635 38342
E-mail: trevore@xephon.com

North American office

Xephon
PO Box 350100
Westminster, CO 80035-0100
USA
Telephone: 303 410 9344

Subscriptions and back-issues

A year’s subscription to *AIX Update*, comprising twelve monthly issues, costs £180.00 in the UK; $275.00 in the USA and Canada; £186.00 in Europe; £192.00 in Australasia and Japan; and £190.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the November 1998 issue, are available separately to subscribers for £16.00 ($24.00) each including postage.

AIX Update on-line

Code from *AIX Update*, and complete issues in Acrobat PDF format, can be downloaded from our Web site at http://www.xephon.com/aix; you will need to supply a word from the printed issue.

Editors

Trevor Eddolls and Richard Watson

Contributions

When Xephon is given copyright, articles published in *AIX Update* are paid for at the rate of £170 ($260) per 1000 words and £100 ($160) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £50 ($80) per 100 lines. In addition, there is a flat fee of £30 ($50) per article. To find out more about contributing an article, without any obligation, please download a copy of our *Notes for Contributors* from www.xephon.com/nfc.

Shell script basics

SAMPLE SHELL SCRIPT

Shell programming is a useful skill to acquire, but to begin using it you do not need to be a programmer, and you do not need to know much more than you already know. To produce professionally written scripts you will obviously need to learn and practise using the more complex constructions, but initially this is not the case. A shell script is often nothing more than a file containing a collection of already familiar commands.

Consider the following shell script called simple_script:

```
$ vi simple_script

print The date and time are:
date
print The following users are logged in:
who
```

When you run this file, each of the commands will be run in order, much the same as if you had typed each at the shell prompt. You will note that we have used the more 'modern' **print** command, which can display text just like **echo**, but, as we shall see later, it allows formatting not achievable by using **echo**. The traditionalists may quake in their boots, but the majority of scripts I will describe will use **print** rather than **echo**.

Create this simple shell script as shown above, and then run the file by following the instructions in the next section (assuming that you don't know how to do this, or perhaps you are looking for alternative ways to run commands).

RUNNING A SHELL SCRIPT

You should be aware that problems may arise when you try to execute a script by calling it from some other command, such as the cron daemon, for example. If the command calling the script executes in the

Bourne shell, and your script contains commands that are specific to the Korn shell, then you will get error messages. You can overcome this by ensuring that the first line of the script is:

```
#!/bin/ksh
```

The # and ! combination is not interpreted as a comment and the whole line tells the system that the commands within the script are to be executed in the Korn shell; it is considered good programming to always start the first line of your scripts with this construction.

There are a number of different ways that a shell script, or any executable command for that matter, can be run. The most obvious way is by first changing its protection mode so that you have permission to execute it. After making the script executable, you can then run it whenever you want to by simply entering its name.

To make it executable use the **chmod** command. If you set the mode of the file to 755, everyone will be able to read and execute it, but only you will be able to modify it. You can alternatively use the **+x** flag to make it executable by all users. You should be aware that when you want to execute a shell script, the specified file must also have read permission.

You can also run the command in your current shell by preceding it with a dot (ie full stop or period). This method of running shell scripts is covered in detail in the section on the .profile file.

A further, but less common, way to run the script is to use the **exec** command, which executes the script directly without creating a new process. In effect the current shell is replaced with the program specified after the **exec** command on the command line. Once the specified command has been run, it then returns to the parent of the process that was running before the **exec**. Unless the executed program is a shell, the command may return to the initialization process (init), thus logging out the user. The reason for using **exec** to run commands may become more apparent when the relationship between programs and processes is explained in future articles.

Another way to run a shell script is by using the **ksh** command, which instructs the operating system to run it as a Korn shell command. The name of the shell script is used as an argument to the **ksh** command, and to be able to run the shell script you must either be in the directory that contains the script or you must specify its full pathname.

When you run a shell script this way, you do not need to use **chmod** to make it executable. To run the shell script named simple_script, enter:

```
ksh simple_script
```

This will run the script, provided that it is in your current directory.

Executing a shell script using **ksh** allows you to use options that alter the way the shell operates. Any options you specify affect only the shell script; they do not affect your login shell.

One of the options to **ksh** is **-x**, which instructs the shell to print each line before it is run (after expanding any variables, etc). This can be useful for debugging shell scripts since it allows you to trace the flow of the script as it is running. When the **-x** option is used, any output directed to standard output by the commands themselves is displayed on the screen and intermixed with the lines that show which commands are being run. Similarly you can put the set **-x** line anywhere in your script and, with one caveat relating to functions (discussed in future articles), from that point onwards in the script you can have your commands displayed as they are executed.

You can, if you wish, use the **sh** command instead to run your script in the Bourne shell, rather than using the Korn shell.

SHELL SCRIPTS VERSUS COMPILED PROGRAMS

When you create a program, one of the first things you must decide is whether to write it as a shell script or in another programming language. Basically, shell scripts just combine existing commands, and although a shell script could be written in another language, the converse is not true – many tasks cannot be performed easily (or at all) by a shell script.

In most circumstances, however, it is much easier to combine efficient AIX commands than it is to write a C program. For instance, if you have data to sort, then it may be worthwhile combining the sort program with other commands to accomplish the task. Similarly, if the task involves searching files for a word or phrase, you may be able to use the **grep** command.

One of the benefits of using shell scripts is that they can be developed quickly, and you don't have to compile them every time you make a modification. However, shell scripts generally run more slowly than compiled programs, which is due in part to the fact that they are interpreted rather than compiled; bad script writing also plays its part.

If execution speed is an issue, you will probably be better off creating a C program. If the program is an interactive one requiring regular user input, then a shell script is probably preferable since the speed of execution is usually determined by the speed at which a user inputs data rather than the execution speed of individual commands.

THE SEARCH PATH

When you enter the name of a command, the shell searches the directories listed in your search path to find the file that is the command. Your search path is listed in the shell variable PATH and the string of characters assigned to PATH is a list of directory names separated by colons. The shell looks in each of these directories in the order they are listed, terminating the search as soon as the executable file with the proper name is found.

As you are no doubt aware, you can display your search path by entering:

```
echo $PATH
/usr/bin:/usr/sbin
```

In this simple example, the search path contains just two directories, /usr/bin and /usr/sbin. When you enter the name of a command, the shell will look for a file with the correct name in the directory /usr/bin. If the executable file with that name is found in /usr/bin, the shell will arrange for it to run. Otherwise the shell will continue to look for the file in /usr/sbin.

If none of the directories in the path contain the name of the executable file, and the command cannot be located as an alias or built-in command (see below), the shell prints a message such as:

```
ksh: filename: not found
```

If this happens, you can usually run the shell script by specifying its full pathname, or by using a relative pathname. For example:

```
/home/userid/simple_script
```

or:

```
./simple_script
```

When you create a shell script, it will normally reside in your current directory, which will be /home/*userid* for most users. Typing the full pathname every time you want to run a shell script is cumbersome, and a better way is to include the current directory in the search path. The simplest way to do this is to begin the path with a colon; identical results can be achieved with a period (full stop) followed by a colon, or a colon followed by a period (full stop) followed by another colon.

If your work involves creating a number of executable shell scripts, it is advisable to place them all in a bin sub-directory, residing in your home directory. You may find that as soon as your userid is created, AIX automatically places /home/*userid*/bin in your PATH, although you will have to create the sub-directory bin.

Similarly, you may also find that AIX adds your current directory to the end of the path, shown as ':.' – this is usually determined by the system administrator who edits the default .profile file copied from /etc/security/.profile when your userid is created. Whether your current directory should be at the end or the beginning of your path is determined to a great extent by your normal working directory, or by security requirements.

If, for example, you are working on a group project, and you need to switch to a particular directory after you have logged in, it will be more efficient for the operating system to search the current directory first if you are continually executing commands located in this directory. In this situation the colon is placed at the beginning of the path. If you are a user who does not have a great number of executable shell scripts, then it will be better if the colon is placed at the end of the path, so that the operating system only searches your current directory as a last resort.

By convention, directories containing commands are named bin, and operating system shell scripts that are for use by many users are mainly stored in the /usr/sbin directory. If you are working on a project where a number of users need to execute customized shell scripts, then it is advisable to place them in /usr/local/bin, a directory customarily created for housing system-wide user-defined scripts.

It is advisable to avoid using the names of existing commands for your shell scripts, since the shell script may be executed when you intended to run the standard command. This may happen if your search path starts with a colon and the shell looks for commands in the current directory first.

There are a number of built-in commands that the shell runs without using the search path. For example, the shell does not use the search path to find the **cd**, **export**, **set**, **test**, and **echo** commands.

If you name a shell script with any of the above names, you will find that the built-in command is run instead of your shell script (unless you specify the full path name), regardless of what your search path is.

ALIASES

Aliases are commonly used to create your own names for commands, along with their command line arguments, so that they are either easier to remember, or, as in many cases because of idleness (sorry, efficiency!), allow you to type in a shorter version of an easily remembered command. If an alias is defined, the command for that alias is resolved/ substituted when you use the alias.

When the alias command is used without arguments, it lists the current aliases that have been set. These are displayed, one per line, in the form:

```
name=command
```

You can define an alias by using, for example, a command such as:

```
alias lse="ls -aceli"
```

Any spaces contained within the alias must be inside quotes in order to prevent the shell from interpreting command line options as additional commands to be executed.

The quoting of alias values is also important. Using double quotation marks will expand most metacharacters when the alias is defined, whereas single quotes will only expand the metacharacters when the alias is actually used.

For example, suppose you define the alias:

```
alias path='echo $PATH'
```

then, if you enter the alias command to display all the aliases, *path* will be shown to equal echo $PATH. The actual expansion of $PATH will not occur until *path* is used.

If, however, you had defined *path* with:

```
alias path="echo $PATH"
```

then the substitution would be made with the current value of PATH as soon as *path* was defined, and on entering the alias command this value would be shown as an argument to echo. Any subsequent changes to your search path would not be shown when you later run the path command.

The first character of an alias can be any printable character, but all other characters must be alphanumeric. Do not use a metacharacter as the first character (apart from possibly an underscore) since this will cause conflict with the shell's interpretation of the alias. The command to be executed within the alias can be any shell script or AIX command, and may include some metacharacters; the best rule is to avoid metacharacters.

If an alias ends in a blank space or tab, the word following the alias is also checked for alias substitution. For example:

```
alias dir="ls ";
alias opts="-CF"
dir opts
```

will produce the appropriate directory listing.

To remove an alias, the unalias command is used. For example:

```
unalias opts
```

You should be aware that when you enter the name of a command to be executed, the system will first check to see if it exists as an alias, even before it checks to see whether it is a built-in shell command. If you have an alias that has the same name as another command, then this command will never be executed.

Using in sub-shells

To allow aliases to be used in sub-shells they must be exported, and this can be achieved by first creating a file to contain the alias commands. For example, you can create a file, which is conventionally called .kshrc (for Korn shell users), locate it in your home directory, and place in it commands such as:

```
alias lse='ls -acle'
alias path='echo $PATH'
```

Once you have created the file, you must then set and export the ENV system variable within your .profile file:

```
export ENV=/home/userid/.kshrc
```

You must now log off and log in again so that the system can read both the .profile and .kshrc files. Whenever you log in, or create a sub-shell, the .kshrc file is executed so that its alias contents are available as part of your environment.

This type of file can also be used to contain function definitions, which are discussed in a future article, and you could, if you wished, put in other commands that you would like executed.

Tonto Kowalski
Guru (UAE)

Migrating from NT4.0 to AIX

This article describes how to easily migrate Windows NT 4 users to AIX (or even other Unix variants).

For a list of the users, firstly, under NT4.0, type:

```
net user /domain > users.txt
```

This creates a file called users.txt, which is then copied to the AIX machine.

Next, the program nt40pawd is submitted (./nt40pawd).

You have to answer Y to the first question, */export/home* to the second, then answer Y to the third question, and *ntadm* to the last one.

A file, passwd.new, is created, and it must be added at the end of the /etc/passwd file using the command:

```
cat passwd.new >> /etc/passwd
```

NT40PAWD

```
#!/usr/bin/perl
#nt40pawd
if ( ! ($#ARGV == 1) ) {
   printf STDERR "Usage : nt2passwd <inputfile>\n";
   printf STDERR " <inputfile>   - exit from the command 'net user /
domain'\n";
   exit (-1);
}

# open the file for entries
open ( USER_LIST, "$ARGV[0]" ) || die $!;

# recover the user id for leavers
print "Enter the uid for leavers : ";
$start_uid = <STDIN>;
$start_uid = int ( $start_uid );

if ( $start_uid eq 0 ) {
   printf STDERR "The uid 0 is refused!\nEnd of program...\n";
   exit (-1);
}
```

```
$current_uid = $start_uid;

print "Enter the group id for leavers : ";
$gid = <STDIN>;
$gid = int ( $gid );
if ( $gid eq Ø ) {
   printf STDERR "An empty group id does not allow the generation of
IDs for the group!\n";
   printf STDERR "End of program...\n";
   exit (-1);
}

print "\nDo you want to create a personal index for the users? (y/n) ";
$make_home = <STDIN>;
$make_home = substr ($make_home, Ø, 1);
$make_home =~ s/./\l$&/g;
if ( "$make_home" eq "o" ) {
   print "Enter the index root : ";
   $homebase = <STDIN>;
   chop ( $homebase );
   print "Do you want the individual indexes to be created
automatically? (y/n) ";
   $create = <STDIN>;
   $create = substr ($create, Ø, 1);
}

$comment = 'Compte NT factice';
$shell = '/bin/False';

# open the leavers file
open ( PASSWD, "> passwd.new" ) || die $!;
open ( MAPFILE, "> username.map" ) || die $!;

# browsing the leavers file
while ( $string = <USER_LIST> ) {

   chop ( $string );

   # verify the exit command and preserve the list of users
   $string = &checkInput ( $string );

   # Split up the entries
   if ( "$string" ne "" ) {
      @users = split (/\ +/, $string );

      foreach $user ( @users ) {
         ($name) = getpwuid ( $current_uid );
         while ( "$name" ne "" ) {
            $current_uid++;
            ($name) = getpwuid ( $current_uid );
         }
         $user =~ s/./\l$&/g;
         if ( length($user) > 8 ) {
            $got_user = Ø;
            while ( $got_user == Ø ) {
               print "The name of the users [$user] must not exceed
eight characters : ";
               $new_user = <STDIN>;
               chop ($new_user);
               ($new_username) = getpwnam ($new_user);
               if ( "$new_username" ne "" ) {
                  print "[$new_username] exist.  Do you want to enter
another name? (y/n) ";
                  $username_result = <STDIN>;
                  $username_result = substr ($username_result, Ø, 1);
                  $username_result =~ s/./\l$&/g;
                  if ( "$username_result" eq "n" ) {
                     $got_user = 1;
                  }
               }
               else {
                  $got_user = 1;
               }
            }
            if ( $got_user == 1 ) {
               printf MAPFILE "$new_user=$user\n";
               $user = $new_user;
            }
         }
         if ( length($user) <= 8 ) {
            ( $username ) = getpwnam ( $user );
            if ( "$username" eq "" ) {
               if ( "$make_home" eq "o" ) {
                  $homedir = $homebase . '/' . $user;
                  if ( "$create" eq "o" ) {
                      mkdir($homedir, Ø755);
                      if ( $! ) {
                         printf STDERR "There was an error during the
creation of $homedir !\n";
                      }
                      chown ( $current_uid, $gid, $homedir );
                  }
               }
               else {
                  $homedir = '/dev/null';
               }
               printf PASSWD
"$user:*:$current_uid:$gid:$comment:$homedir:$shell\n";
```

```
            }
            $current_uid++;
         }
      }
   }
}

# closing the files
close ( USER_LIST );
close ( PASSWD );
close ( MAPFILE );

# end of program
exit (0);
#################################################################

sub checkInput {
   local ( $input ) = @_;

   if ( $input =~ '\\\\' ) {
      $input = '';
   }
   elsif ( $input =~ "End of program" ) {
      $input = '';
   }
   elsif ( $input =~ '————' ) {
      $input = '';
   }

   $input;
}
```

For the groups, under Windows NT the command

```
net group /domain > group.txt
```

must be typed.

The file group.txt is copied to the Unix machine. Under Unix, enter the command to run the program groupnt on the text file:

```
groupnt40 group.txt
```

A file group.new is created. Type:

```
cat group.new >> /etc/group
```

This appends the new list to the existing list.

Claude Dunand
(France) © Xephon 2002

Some tips for shell programming

WHAT HASH-BANG REALLY DOES (#!)

Once upon a time there was only one Unix shell. When a script was written and executed, the Bourne shell read in the script and executed the commands.

The Bourne shell had some limitations and it wasn't long before a proliferation of shells started to appear. Each shell had its own syntax and some of them, such as the C shell, were very different from the original Bourne shell.

This meant that if a script took advantage of the features of one shell or another, the script had to be run using that particular shell.

Instead of typing:

```
doit
```

The user had to know to type:

```
/bin/ksh doit
```

or:

```
/bin/csh doit
```

To handle this, a clever change was made to the Unix kernel. This is now available in most Unix systems. A script can be written that starts with a hash-bang (#!) combination on the first line followed by the name and path of the shell to use to execute the script.

The following script, named doit, uses this technique to specify the Korn shell as the shell to use to execute it:

```
#! /bin/ksh
#
# do some script here
#
```

The kernel starts reading in the script, doit, sees the hash-bang and reads the rest of the line where it finds /bin/ksh. The kernel starts the Korn shell and then feeds the doit script to it.

Actually the kernel launches the Korn shell with doit as an argument to it as if the user had typed:

```
/bin/ksh doit
```

When /bin/ksh starts reading in the doit script it sees the first line containing the hash-bang as a comment (because it starts with a hash) and ignores it.

The full path to the shell to be run is required, because the kernel does not search your PATH variable.

The hash-bang handler in the kernel does more than just run an alternative shell. It actually takes the argument following the hash-bang and uses it as a command. It then adds the name of the current file as an argument to the command.

You can start a Perl script named doperl by using the hash-bang:

```
#! /bin/perl

# do some perl script here
```

In this example, if you start the script by simply typing doperl, the kernel spots the hash-bang, extracts the /bin/perl command, and then runs this as if you had typed:

```
/bin/perl doperl
```

There are actually two mechanisms in play here that allow this to work. The first is the kernel interpretation of the hash-bang (#!). The second is the fact that Perl itself sees the first line as a comment and ignores it.

You cannot use this technique for any scripting language that does not treat a line starting with a hash (#) as a comment. If the language does not treat a hash mark as the beginning of a comment, the language will see the first line and try to do something with it. This will most likely cause an error. This is not a very big problem because the hash mark as a comment is used in almost every Unix scripting language.

This technique is not limited simply to running scripts, although that is where it is most useful.

The following script, named helpme, types itself to the terminal when you type the command helpme:

```
#! /bin/cat
vi      unix editor
man     manual pages
sh      Bourne Shell
ksh     Korn Shell
csh     C Shell
bash    Bourne Again Shell
```

This kernel trick also allows one argument after the name of the command to execute. You could change the helpme file to use more, as shown, but add the **+2** argument to force more to start at line 2. This will skip the initial line. This is the line containing the hash bang starter. Be sure to use the correct path to your version of more:

```
#! /bin/more +2
vi      unix editor
man     manual pages
sh      Bourne Shell
ksh     Korn Shell
csh     C Shell
bash    Bourne Again Shell
```

Typing helpme as a command causes the kernel to convert this to:

```
/bin/more +2 helpme
```

and displays everything from line 2 onwards:

```
helpme
vi      unix editor
man     manual pages
sh      Bourne Shell
ksh     Korn Shell
csh     C Shell
bash    Bourne Again Shell
etc.
```

You can also create apparently useless activities such as a file that removes itself:

```
#! /bin/rm
```

If you named this file flag, then running flag would cause the command to be issued as if you had typed:

```
/bin/rm flag
```

You could use this in a script to flag that you are running something and then later simply execute the script to remove it:

```
#! /bin/ksh
# First refuse to run if the flag file exists

if [-f flag ]
then
    exit
fi

# create the flag file

echo "#! /bin/rm" >flag
chmod a+x flag

# do some logic here

# unflag the process by executing the flag file

flag

# The flag file has now disappeared itself
```

Before you start building long commands using this technique, know that your system may have a limit on the number of characters in the #! line. This will usually be about 32 characters.

TESTING COMMAND LINE ARGUMENTS AND USAGE

When you write a shell script it is common for the script to need arguments for it to function correctly. It is necessary to validate those arguments to ensure that they make sense for the script.

The simplest level of validation is ensuring that there are enough arguments. For example, if you have created a shell script that requires two file names to operate, then the minimum validation that you should do is to test that there are two arguments on the command line. To test this in Bourne and Korn shells, check the value of **$#**. This is a shell-provided variable that contains the count of arguments on the command line. This is the count of arguments other than the command itself.

It is also good practice to provide some sort of message as to why the command failed. This is usually created in a **usage()** function. The listing of twofiles tests that there are two arguments on the command line:

```
#! /bin/ksh

# twofiles script handles two files named on the command line

#  a usage function to display help for the hapless user

usage ()
{
     echo "twofiles"
     echo      "usage: twofiles file1 file2"
     echo      "Processes two files"
}

# test if we have two arguments on the command line
if [ $# != 2 ]
then
    usage
    exit
fi

# we are OK at this point so continue processing here

# processing logic starts here
```

A safer programming practice is to validate as much as you can before launching into your execution. The following version of twofiles checks the argument count and then tests both files. If file1 doesn't exist (**if [! -f $1]**) an error message is set up and a usage message is displayed. Then the program exits. The same is done for file2:

```
#! /bin/ksh

# twofile script handles two files named on the command line

#  a usage function to display help for the hapless user
# plus an additional error message if it has been filled in

usage ()
{
     echo "twofiles"
     echo      "usage: twofiles file1 file2"
     echo      "Processes two files"
     echo " "
     echo $errmsg
}

# test if we have two arguments on the command line
if [ $# != 2 ]
```

```
then
    usage
    exit
fi

# test if file one exists and send an additional error message
# to usage if not found

if [ ! -f $1 ]
then
    errmsg=${1}":File Not Found"
    usage
    exit
fi

# same for file two

if [ ! -f $2 ]
then
    errmsg=${2}":File Not Found"
    usage
    exit
fi


# we are OK at this point so continue processing here

# processing logic starts here
```

Note that in the Korn shell you can also use the double bracket test syntax, which is faster. The single bracket test syntax actually calls a program named test to test the values. The double bracket test is built into the Korn shell and does not have to call a separate program. The double bracket test will not work in the Bourne shell:

```
if [[ $# != 2 ]]
```

or:

```
if [[ ! -f $1 ]]
```

or:

```
if [[ ! -f $2 ]]
```

Doing this type of thorough validation can prevent later errors in the program logic when a file is suddenly found missing. It is good programming practice.

SHELL SCRIPT ARGUMENTS

Before we launch into arguments and options we have to back up a bit to discuss the Unix command line. This is a very short description of some of the behaviour and features of the command line.

A Unix command line is a sequence of characters in the syntax of the target shell language. Of the characters in a command line, some are known as metacharacters. The metacharacters have a special meaning to the shell. The metacharacters in the Korn shell are:

- ; – separates multiple commands on a command line.
- & – causes the preceding command to execute synchronously (at the same time as the next command on the command line).
- () – commands enclosed in parentheses are launched in a separate shell.
- | – pipes the output of the command to the left of the pipe to the input of the command on the right of the pipe.
- > – redirects output to a file or device.
- < – redirects input from a file or device.
- newline – ends a command or set of commands.
- space – separator between command words.
- tab – separator between command words.

Some of these metacharacters can be used in combinations such as '||' and '&&'. Consult your manual for a complete description.

With these metacharacters in mind, you can define a command line word.

A command line word is a sequence of characters separated by one or more non-quoted metacharacters. In the following example, the passwd file is piped through **cut** and fields 1 and 3 are output based on a colon delimiter.

In the following command line:

```
cat /etc/passwd|cut -d ":" -f 1,3 >usruid.txt
```

the words are:

cat

/etc/passwd

cut

-d

“:”

-f

1,3

usruid.txt

Note that the metacharacters |, >, and space have been removed.

The metacharacters &, |, (), ; and newline are used to separate or terminate multiple commands within a command line.

Within the example command line, there are two commands separated by the pipe (|) symbol.

```
cat /etc/passwd
cut -d ":" -f 1,3
```

The final portion of the command line (> usruid.txt) could be thought of as the command, and output the result to usruid.txt; however, redirection is not usually considered part of a command.

When a command executes a Unix program, utility, or shell script, it is usual for the command to include arguments. In the example above the argument to **cat** is **/etc/passwd**. The arguments to **cut** are **-d**, **":"**, **-f**, and **1,3**.

In general, arguments are all the words (note the definition of word above) that follow an executable program name in a command. Arguments within a command are separated from one another by spaces or tabs (metacharacters).

Most Unix programs have been written with some standards as to how arguments and options are arranged for the program’s use.

Options are the letters or numbers that follow a minus sign. These are sometimes called switches. A simple example of arguments and options would be the use of the **cat** command:

```
cat -v -e -t doodah.txt
```

In the above example, the arguments are **-v**, **-e**, **-t**, and **doodah.txt**. The options are the entries for **-v**, **-e**, and **-t**.

Just so you don't have to run for your manual: the **-v** option asks **cat** to display all characters (even non-printable ones). The **-e** option specifies that the end of a line will be displayed as **$**. The **-t** option specifies that a tab should be displayed as ^I instead of expanding the tab into spaces on the screen.

Unfortunately no standard terminology has been developed to indicate an option versus a non-option argument. To add some complexity to the terminology problem, an option can itself have an argument. In the first example using **cut** (repeated below), the **-d** option has an option argument of **":"**, and the **-f** option has an option argument of **1,3**:

```
cut -d ":" -f 1,3
```

In order to clarify these, various manuals have adopted some sort of standards for naming conventions for the parts of a Unix command. These two examples are used to illustrate the parts:

```
cat -v -e -t doodah.txt

cat /etc/passwd|cut -d ":" -f 1,3 >usruid.txt
```

The program name itself, **cat** in the first example or **cat** and **cut** in the second example, is called the name, progname, executable, or program-name.

The non-option arguments to a command, doodah.txt in the first example and /etc/passwd in the second, are called an operand, a cmdarg, or an argument.

The options, **-v**, **-e**, and **-t** in the first example and **-d** and **-f** in the second, are called options, opts, or switches.

The arguments to options **":"** and **1,3** in the second example are called option-arguments or optargs.

The standards used in creating Unix executables are:

1 Command names must be between two and nine characters long.

Command names must include only lowercase letters and digits.

3 Option names (options above) must be one character long.

4 All options must be preceded by '-'.

5 Options with no arguments may be grouped after a single '-'. For example **-v**, **-e**, and **-t** could also be written as **-vet**.

6 The first option-argument following an option must be preceded by white space.

7 Option-arguments cannot be optional.

8 Groups of option-arguments following an option must either be separated by commas or separated by white space and quoted (eg **-f 1,3** or **-o "xxx z yy"**).

9 All options must precede operands on the command line.

10 '-' may be used to indicate the end of the options.

11 The order of the options relative to one another should not matter.

12 The relative order of the operands (cmdargs) may affect their significance in ways determined by the command with which they appear.

13 '-' preceded and followed by white space should be used only to mean standard input.

Not all Unix commands follow these rules, but all the newer ones do. Older executables were written before the standard was established but the executables were in such regular use that it was decided not to change them. For example, **cut** will function with or without rule 6 requiring a space before the option-argument. Both of the following commands will work on most systems:

```
cat /etc/passwd|cut -d ":" -f 1,3 >usruid.txt
cat /etc/passwd|cut -d: -f1,3 >usruid.txt
```

The **find** command is another example of an older style program that hangs on. It uses options that are longer than a single character, violating rule 3, and allows options to appear after the operand, violating rule 9. In the following example, dot (.) is the operand, **-name** and **-print** are options, and data.txt is the option-argument for **-name**:

```
find . -name data.txt -print.
```

A rule has been added to many newer commands to allow switches longer than one character. This rule allows a '-' to precede an option that is longer than a single character.

If **find** had been written recently, it would have looked more like the following on the command line:

```
find . —name data.txt —print.
```

Right about now I hear you saying, "What is all this blather leading up to?". Well Unix provides a handy tool for separating option arguments and operands. It is the **getopts** function. The **getopts()** function requires an argument of a string containing the list of valid option characters, and a shell variable that will receive the result of searching the arguments. The **getopts()** function can be called several times. Each time it is called it steps forward through the list of arguments picking up the next option. It can also pick up an option-argument, and the index of the argument that it has processed. This is easy to illustrate.

First let's spec out a small problem. In this problem we need a shell script that will archive a file by copying it to an archive directory. The default directory is /u/arch, but the path of the archive directory can be changed on the command line. The archive program will also stop and ask you for an *OK* to proceed, if it is about to overwrite an earlier archive. This behaviour can be changed by an option that can be set to overwrite without warning. The syntax for this command line for this archive program would be:

```
arch [-r] [-a /new/archive/path] filename
```

where the **-r** option will automatically replace an existing archive file without warning. The default is to warn. The **-a** option is followed by an alternative archive directory to use instead of /u/archive. Finally filename is the name of the file to archive.

The following is a listing for **arch** that covers the processing of the option arguments. It does not include the logic for doing the actual archive operation. After the complete listing is a step-by-step analysis of how the program works.

```
#! /bin/sh
#:----------------------------------------------------------------------
usage () {
 echo "Usage:"
 echo "arch - archives a file to /u/arch directory"
 echo "syntax:"
 echo "    arch [-r] [-a /new/archive/path] filename"
 echo "where"
 echo "    -r will automatically replace an existing archive file"
 echo "       (default is to warn)"
 echo "    -a specifies an alternative archive directory"
 echo "    filename is the name of the file to archive"
 exit
}
#:----------------------------------------------------------------------

replace="w"
arch="/u/arch"
filename=""

optstr=":ra:"

while getopts $optstr opt
do

    case $opt in
        r) replace="r";;
        a) arch=$OPTARG;;
        *) usage;;
    esac
done

shift 'expr $OPTIND - 1'

filename=$1

echo "Archiving" $filename " to " $arch "with" $replace "replace
option"

# rest of the code to actually do the archiving goes here
```

Portions of the listing with line numbers are used below for illustration.

The **getopts()** function does not always work correctly with the Korn shell, so line 1 forces the script to run in the Bourne shell. The program starts at lines 2 through 15 with a comment describing its actions that also doubles as a usage function. The usage function is called when the user makes a mistake.

```
 1   #! /bin/sh
 2   #:-----------------------------------------------------------------
 3   usage () {
 4    echo "Usage:"
 5    echo "arch - archives a file to /u/arch directory"
 6    echo "syntax:"
 7    echo "    arch [-r] [-a /new/archive/path] filename"
 8    echo "where"
 9    echo "    -r will automatically replace an existing archive file"
10    echo "       (default is to warn)"
11    echo "    -a specifies an alternative archive directory"
12    echo "    filename is the name of the file to archive"
13    exit
14   }
15   #:----------------------------------------------------------------
```

At lines 17 and 18, shell variables are set up to contain the default values to be used when archiving (the archive directory and a value indicating whether to warn the user when replacing). Line 19 sets up a variable to hold the file to be archived:

```
17   replace="w"
18   arch="/u/arch"
19   filename=""
```

This program will have two possible options, **-r** and **-a**. The **-a** option will require an option-argument that names the directory to use. An options string should contain the list of single character identifiers to be used for options or **ra**. In addition, if an option is to be followed by an option argument, that option should be followed by a colon. So far the option string to be used becomes **ra:**.

Finally, **getopts** will produce an error message if an invalid option is placed on the command line. In order to suppress the error message, start the option string with a colon, thus **:ra:**. That string is set up at line 21 of the script.

```
21   optstr=":ra:"
```

Whenever **getopts()** is called, it locates the next available option, retrieves the character, and places it in the passed variable name. At line 23 this variable, $opt, is passed as the second argument to **getopts()** after $optstr.

The **getopts** function returns true as long as it continues to find arguments that start with a leading hyphen. When it finds **-r** on the command line it places **r** in $opt. When it finds **-a**, it places **a** in $opt. Whenever **getopts()** finds an option that is expecting an option-argument, it retrieves the argument and places it in a variable named $OPTARG. The loop at lines 23 through 31 processes all options and option-arguments by repetitively calling **getopts()**.

Inside a case statement the various results are processed. If **-r** was encountered, then **r** will appear in $opt and $replace is set to **r**. If **-a** was encountered, then **a** will appear in $opt and the value in $OPTARG is used to set the value of $arch. If anything else is encountered, the user has entered an invalid option. This calls the **usage()** function. The **usage()** displays a usage description message and exits the program:

```
23   while getopts $optstr opt
24   do
25
26       case $opt in
27           r) replace="r";;
28           a) arch=$OPTARG;;
29           *) usage;;
30       esac
31   done
```

The **getopts()** function also keeps one other variable, $OPTIND, which contains the index of the next argument to be processed. When the shell script is first started, $OPTIND is set to 1. If **-r** is processed as the first argument, $OPTIND will contain 2. If **-a** is processed as the second argument, and the name of an archive directory as the third argument, $OPTIND will contain 4. On the next call to **getopts**, **getopts** returns false and the loop at lines 23 through 31 ends. At this point $OPTIND still contains the value 4. This value can now be used as the index of the next argument, which will be the first argument that does not start with a hyphen. This should be the name of the file to archive.

At line 33 the shift command is used to shift all arguments by $OPTIND – 1. This will cause the argument that was at position 4 ($4) to be shifted to the left by 3 positions, so that it now becomes argument $1. At line 35 this value is picked up and stored in $filename:

```
33   shift 'expr $OPTIND - 1'
34
```

```
filename=$1
```

At this point, a good script would do further error checking such as checking that the file named in $filename does exist, and that the archiving directory in $arch also exists. In this example the results of the extracted values are displayed.

```
echo "Archiving" $filename "
                         to " $arch "with" $replace "replace
option"

# rest of the code goes here
```

Using **getopts()** is an excellent way to create scripts that comply with the Unix command standard. It also makes it possible to add features to your scripts fairly easily. Assume that you wanted to enhance your arch script to include an option to somehow put a date and time stamp on an archive. The options are easy to add by extending the $optstr variable to allow for a **-d** option. Add a variable and extend the case statement as in the following example and you have easily added a **-d** option to the arch command. Of course, you have to add the code to handle $datestamp="Y", but the user interface is taken care of easily:

```
replace="w"
arch="/u/arch"
filename=""
datestamp="N"

optstr=":ra:d"

while getopts $optstr opt
do

    case $opt in
        r) replace="r";;
        a) arch=$OPTARG;;
        d) datestamp="Y";;
        *) usage;;
    esac
done
```

Using these tools and tips, I hope you will be able to create more powerful scripts.

Mo Budlong
President
King Computer Services (USA)

Advanced emacs use

(E-)LISP: SOMETHING TO LEARN

You will have seen references to ‘functions’ and lisp in my previous article (*An introduction to emacs*, *AIX Update*, Issues 75 and 76, January and February 2002), and in any further investigations into emacs as well.

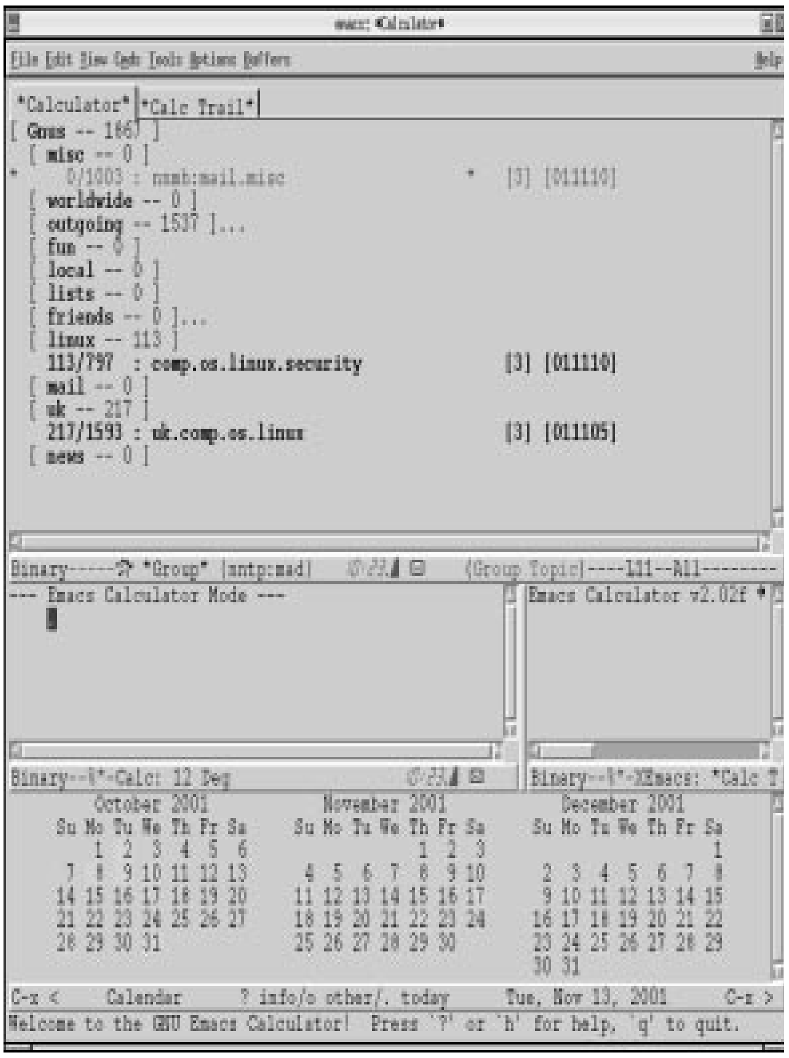
Emacs is based entirely on its own native dialect of lisp (see http://www.lisp.org/). This shows through in several ways, but most notably in the fact that you can ask it what pressing any key (combination) performs, and the answer always comes back in terms of a function. Right now, ‘C-h k e’ tells me that ‘e’ is bound to the ‘self-insert-command function’, which inserts a character corresponding to the key pressed. Try it. Try doing ‘C-h k C-h k’ as well. For reference, note that pressing ‘C-h C-k’ works equally well with a menu selection afterwards, not just a conventional keypress. There is no difference!

Emacs is a complete elisp environment where one of the side-effects is text editing. This allows you to program or script in the environment yourself, implementing or using entire packages of elisp source. There are many modes already defined for handling different types of buffer; there are whole packages provided that implement, amongst other things, a telnet client for MOOs/MUDs, two e-mail clients, one of which is both a news and an e-mail client in one (Gnus, see http://www.gnus.org), a Web browser (W3), a calculator, a calendar, an address-book, a shell... you name it, it’s probably been done already, and if not, you have a whole lisp environment in which to do it.

Figure 1 shows a reasonably well configured Xemacs session showing Gnus, the calculator, and the calendar all at once.

We have already seen references to ‘M-x’. This invokes a command in emacs lisp, which is a function defined to be interactive. There is also ‘M-:’, which evaluates a given lisp expression, putting the results in the minibuffer.

Try:



```
[ Gnus -- 1867 ]
  [ misc -- 0 ]
*     0/1003 : nnmh:mail.misc                *    [3] [011110]
  [ worldwide -- 0 ]
  [ outgoing -- 1537 ]...
  [ fun -- 0 ]
  [ local -- 0 ]
  [ lists -- 0 ]
  [ friends -- 0 ]...
  [ linux -- 113 ]
    113/797  : comp.os.linux.security             [3] [011110]
  [ mail -- 0 ]
  [ uk -- 217 ]
    217/1593 : uk.comp.os.linux                   [3] [011105]
  [ news -- 0 ]

Binary-----?? *Group* {nntp:mad}          (Group Topic)----L11--All--------
--- Emacs Calculator Mode ---                Emacs Calculator v2.02f *

Binary--%*-Calc: 12 Deg                      Binary--%*-XEmacs: *Calc T
        October 2001            November 2001            December 2001
    Su Mo Tu We Th Fr Sa    Su Mo Tu We Th Fr Sa    Su Mo Tu We Th Fr Sa
        1  2  3  4  5  6                 1  2  3                       1
     7  8  9 10 11 12 13     4  5  6  7  8  9 10     2  3  4  5  6  7  8
    14 15 16 17 18 19 20    11 12 13 14 15 16 17     9 10 11 12 13 14 15
    21 22 23 24 25 26 27    18 19 20 21 22 23 24    16 17 18 19 20 21 22
    28 29 30 31             25 26 27 28 29 30       23 24 25 26 27 28 29
                                                    30 31
C-x <     Calendar      ? info/o other/. today      Tue, Nov 13, 2001     C-x >
Welcome to the GNU Emacs Calculator!  Press `?' or `h' for help, `q' to quit.
```

Figure 1: A reasonably well configured Xemacs

```
M-: (current-time-string)
```

which should put the current timestamp in the minibuffer.

Now try:

```
M-: (insert (current-time-string))
```

which should insert the current timestamp into your current buffer.

These little fragments of code are lisp ‘s-expressions’. The first (in these cases, only) word after the opening parenthesis is evaluated, with the following words as arguments. That is, in the example above it evaluates the function *current-time-string*, which returns a string 'object' containing the current time, and passing the results to the *insert* function, which returns a nil object, but has the side-effect of altering the current buffer.

You can also evaluate any piece of lisp that you find, in a buffer. Type:

```
(current-time-string)
```

into the scratch buffer, and press 'C-x C-e'. The minibuffer should display the current time as a string. You can also insert the value into the buffer by pressing 'C-u C-x C-e'.

Given that all these s-expressions are 'weighted', having the first argument evaluated, and because arithmetic operators are also regarded as functions, it is apparent that this is a prefix language: if you were to evaluate an expression such as:

```
(+ (* 3 2 ) (- 9 4))
```

you should not be surprised that 11 is the result. Note that there is no ambiguity in what this means, and therefore no need for conventional 'bracketing' of sub-expressions against a variable evaluation order. Note also that we speak of 'evaluation', 'expressions', 'functions', and 'side-effects', never of 'executing' (unless using M-x). These are the terms used when talking about a functional programming language, as distinct from an imperative language.

Detail and philosophy

Functions are themselves valid arguments to other functions in an expression (this is a defining characteristic of a functional language).

Whole programs are expressible as nothing but a series of functions. For example, consider this short snippet – the functional programmer's equivalent of 'hello world':

```
(defun fact (n)
  "Return the factorial of N"
  (if (= 0 n)
      1
      (* n (fact (- n 1)))))
```

```
(setq fact 13)
```

```
(mapcar #'fact '(1 3 5 7 9 11))
```

Obviously, the result returned from evaluating these three top-level expressions is the list:

```
(1 6 120 5040 362880 39916800)
```

(You should check this, evaluating all three expressions yourself, by pressing C-x C-e after each of them in the scratch buffer.) However, it is worthwhile studying the data as it flows through the above.

The fundamental unit of data in lisp is the symbol. Given that emacs lisp is a Lisp-2, this means that each symbol has both a function and a data 'component'. The symbol known as *fact* in the above has both aspects made clear: first, we define a function that does the job of a factorial, and we store it in the function part of the symbol *fact*. Then we gratuitously set the data part of the symbol to 13. The point is that what you get out of the symbol depends on how you look at it later; in this case, we take the function part by using *#'* as a quote (to delay it against one level of evaluation), and map it across the list of integers given. (The list of integers is also quoted, but as data, in order that it does not evaluate '1' as a function.) The notation is that #' quotes as a function, ' quotes as a data variable, and "" are used to quote a string containing spaces.

The result of mapping a 1:1 function across a list of integers, is another list, in this case of integers; however, you can still evaluate the symbol name *fact*, eg by typing 'M-: fact RET', and it will tell you '13', the number that was assigned to it, in its data aspect.

A quick example

For various reasons, you want to produce a graph, but the data you have is of the form:

```
Tim 30+200
Nat 29+119
...
```

so you want to evaluate the second column, adding the two numbers together, replacing it with just one number, that you can then feed to *gnuplot*. But that column is not an expression that the lisp interpreter will recognize.

The easiest way to effect this transformation is to define a keyboard macro that is going to transform one line, and leave you at the start of the next line.

This macro should jump forward to the first digit (eg by calling M-x search-forward-regexp “[0-9]”).

The macro then inserts the text ‘(+ ’ in front of the first number, uses M-f (or C-right) to jump forward to the existing ‘+’ symbol, deletes it and replaces with a space, then goes to the end of the line and appends a ‘)’.

The clever step: the keyboard macro uses ‘C-u C-x C-e’ to evaluate the lisp expression (eg (+ 30 200)) at the current point, inserting the results as it goes.

From there, the keyboard macro only has to remove the expression (defined as everything between parentheses on the current line – selecting the current line and running ‘M-x replace-regexp’ is one approach, as is defining a region between ‘(’ and ‘)’ characters using C-s and C-r), before moving on to the next line.

If this proves confusing, the intermediate stage above looks like:

```
Tim (+ 30 200)230
```

before the expression that was evaluated is removed. The point is that this method combines both keyboard macros and emacs-lisp evaluation. It would also be possible to write out the emacs lisp as one massive long function, eg by using *name-last-kbd-macro* and *insert-kbd-macro*.

When the author did this, the resultant macro looked like:

```
(fset 'foo
   [?\M-x up return ?[ ?0 ?- ?9 ?] return left ?( ?+ ?\S- C-right ?\C-d
   ? ?\C-e ?) ?\C-u ?\C-x ?\C-e C-left ?\C- ?\C-a C-down C-up C-right
   ?\C-w tab down ?\C-a])
```

All you need for configuration

The above is all fine and well, but most people have other things to be doing than pondering the philosophy of lisp. Here’s a set of expressions I use in order to customize my emacs and Xemacs sessions. Take, examine, clone, adjust to taste:

```
(setq load-path (cons "~/elisp" load-path))        ; add a directory to
                                                   ; the search-path

(if (string-match "XEmacs" (emacs-version))    ; choose a config file -
    (load "~/.xemacsrc")                       ; will be written to by
    (load "~/.gnuemacs"))                      ; customize

(defun indent-buffer ()
  "Apply indentation to whole current buffer."
  (interactive)
  (indent-region (point-min) (point-max) nil))
                                                  ; bind C-x C-z to the
(global-set-key [?\C-x ?\C-z] 'indent-buffer)     ; function

(add-hook 'gnus-group-mode-hook 'gnus-topic-mode)  ; when we load Gnus,
                                               ; invoke Topic mode too

(setq-default indent-tabs-mode nil)                ; use space, not tab

(put 'narrow-to-region 'disabled nil)          ; no confirmation prompt

(setq default-frame-alist' ((tool-bar-lines . 0)
                                               ; configure life without
                            (menu-bar-lines . 1)
                                                ; toolbar, with menubar
                            (width . 80)            ; given dimensions
                            (height . 42)))
(setq initial-frame-alist '((tool-bar-lines . 0)   ; start one of these
                            (menu-bar-lines . 1)
                            (top . 20)
                            (left . 100)
                            (width . 80)
                            (height . 42)))

(setq text-mode-hook 'turn-on-auto-fill)      ; in text mode, auto fill
(setq-default fill-column 75)                 ; most things are 75 cols
(display-time)                                ; time in the status bar
(setq line-number-mode t)                     ; line# in the status bar

(defun our-mail-echelon ()                   ; extra headers in e-mails
  "Insert a spooky string from a list"
  (save-excursion
      (load "~/.sig_echelon")
      (goto-char (point-min))
      (insert "X-Anti-Echelon: ")
      (insert (concat (nth (random (length our-mail-echelon-quotes))
                           our-mail-echelon-quotes) "\n" ))))

(transient-mark-mode 1)                          ; switch mode on, always
```

As you use emacs more, expect to spend time setting configuration variables, and producing small amounts of elisp to set them in your ~/.emacs file.

This example configuration file introduces several new lisp functions:

- *setq* is used to assign a value to a symbol; it stands for 'set quote', as it's a contraction of *(set (quote symbolname) value)*.
- *if* introduces a 2-way condition statement; the structure is

```
(if (condition)
    (then-clause)
    (else-clause))
```

- *load* loads (reads & evaluates) a file of source code.
- *defun* starts a function definition; the structure is

```
(defun function-name (arguments)
 "docstring documenting the function"
 (body))
```

- *indent-region* – used to indent every line in the region (between the mark (set using C-space) and current point).
- *global-set-key* – introduces a key binding, valid in global (ie all) contexts. This is where emacs' own syntax for representing a keypress is important – the way we write 'C-something' is the same format as emacs itself uses to parse it.
- *hooks* – a 'hook' is a list of functions to be evaluated on a given condition: for example, *gnus-startup-hook* is evaluated when the Gnus package starts up, and *gnus-article-mode-hook* runs when Gnus' article buffer is created. The *add-hook* function adds a function to the list.
- *display-time* – a function that enables showing of the current timestamp in the status bar.
- *save-excursion* – pushes the current point (and various other bits

and pieces) onto the stack, so that you can perform various actions on the current buffer, and then, at the end of the function, return to where you started. This is useful, for example, when you want to write a function to add headers to an e-mail. When Gnus (or *VM*, if that's what you use) creates a new e-mail buffer, it runs various hooks, so you can define a function that saves the current location (useful if you have other functions creating the signature and asserting a personality), jumps to the top of the buffer, inserts the header you want (maybe including a random selection), and then pops back to wherever it was when it was called.

- *goto-char* – moves the current point to the new location.
- *point-min* – the first character in the buffer.
- *insert* – inserts a piece of text at the current point location.
- *concat* – takes two strings, and produces the concatenation thereof.
- *nth* – takes a number and a list, returning the number *n*th item of that list.
- *length* – returns the length of a list.

The utility of 'C-h a' (*hyper-apropos*) in finding emacs-lisp functions and variables pertaining to a given subject cannot be stressed enough.

EMACS AS A DEVELOPMENT ENVIRONMENT

Using emacs to compile things

If your current directory contains a Makefile, you can invoke **Toolbar/ Tools/Compile**, agree that 'make -k' is the command of choice to run, and emacs will spawn that command, displaying output in a buffer.

It is traditional to demonstrate this with a standard Unix C build process. However, a more imaginative and pertinent example might be the writing of this document; it was originally authored in *Latte*, but there is a Makefile in this directory that generates .html, .ps, .pdf, .tex, and .dvi format documents all in batch mode. By putting:

```
(setq compilation-read-command nil) ;don't ask for how to compile stuff
```

on the end of ~/.emacs, and an appropriate (global-set-key) instruction thereafter, a simple ‘C-x C-\’ will compile the entire document.

If running the *make* generates errors, then emacs will try to parse the error output (this is particularly well implemented against *gcc*; other Unixes’ *cc* implementations’ outputs might not be perfectly parsed); you can use C-x ' in order to bounce to the next error, jump to the corresponding source file, so you can edit it and try again.

CVS manipulation within emacs

Emacs supports ‘cvs-mode’, which is just like any other emacs mode, except that it provides a means of interacting with the CVS system. Specifically, if you open a file from a directory containing a CVS/ subdirectory, emacs will load the CVS-mode, so that, for example, this document has a status line saying ‘(Latte CVS:1.1 Fill)’, for the full list of modes involved in editing it.

With CVS-mode, you can open your document, edit away to your heart’s content, re-run ‘M-x compile’ to make sure that it still builds, and then use ‘C-x v v’ to check in your adjustments to the upstream repository; emacs will even pop up a buffer in which you can type any comments you wish to put in the CVS history.

The other options available under the CVS menu include reversion to previous version, diff against previous version, history, Changelog support, and register a new file into CVS. The **Tools/Version Control** menu contains more details including the keyboard shortcuts for all these and a few more.

Putting CVS-mode and the compile ability together makes for a very powerful development environment – this document was developed in exactly this fashion: edits, a quick compile to make sure it was still valid, and changes committed regularly to CVS.

gnuclient

gnuclient is a commandline utility whereby you can keep one emacs session open, and connect to it in order to evaluate lisp (eg to edit another file), but from a (potentially remote) session.

For example, I have an Xemacs21 session running here, within which the gnuserv-start function has been evaluated. There is now a gnuserv process spawned from xemacs21 to handle connections made to that Xemacs session:

Process tree:

```
'-xinit(563)-+-XFree86(564)
              '-blackbox(568)-+-emacs21(700)
                              '-xemacs21-mule(782)—gnuserv(783)
```

Socket listener:

```
tcp    0  0 0.0.0.0:22490       0.0.0.0:*       LISTEN      783/gnuserv
```

It is now possible to connect to this server and open a buffer editing a file, remotely:

```
bash$ gnuclient -h localhost somefile
```

The file *somefile* will be opened in the (X)emacs session from which the gnuserv process is started, but in a frame at the gnuclient end of the connection – ie if you have a DISPLAY variable then it'll use that, otherwise it'll be in the current term window; you can double-check that it is the existing emacs session by pressing 'C-x C-b' to obtain a buffer-list window. It's even more fun to be editing the same buffer in two frames simultaneously.

Of course, gnuclient also allows you to evaluate some elisp in an already-running emacs session:

```
bash$ gnuclient -h localhost -eval '(shell-command "ls")'
```

This comes in handy if you want to run emacs in a loop from a shell script or Makefile, but don't want the overhead of having to load a 10 or 20MB image from disk for each iteration(!).

It is important to stress the need for security in dealing with this process; the port is relatively wide open, so read up on the use of the GNU_SECURE environment variable (it points to a file whose contents are a list of permitted client hosts), or the X-based authority system that can also be used. Beware that data travels across the network in plaintext unless you go to some pains to tunnel it through, for example, *ssh*.

AIX SPECIFICS

Availability

Emacs and Xemacs have both been packaged for various versions of AIX. One place to download them from is http://www-1.ibm.com/servers/aix/products/aixos/linux/download.html. Amongst other things, one gotcha is the potential for a segfault if you don't have your locale set up correctly; see http://www.faqs.org/faqs/aix-faq/part4/section-38.html for more, but note that it might help if you *export LC_ALL=C*.

Building your own

Gnu emacs is available from http://www.gnu.org/software/emacs/emacs.html; Xemacs is available from http://www.xemacs.org.

Both use a standard GNU build procedure:

```
./configure
     make
     sudo make install
```

where 'configure' takes several options – there are the normal '-prefix=' and target/admin/location options, and also -enable and -with options recognized:

- -without-gcc – don't use GCC to compile Emacs if GCC is found.
- -without-pop – don't support POP mail retrieval with movemail.
- -with-kerberos – support Kerberos-authenticated POP.
- -with-kerberos5 – support Kerberos Version 5 authenticated POP.
- -with-hesiod – support Hesiod to get the POP server host.
- -with-mail-spool-directory=DIR – system mail spool is DIR.
- -with-x-toolkit=KIT – use an X toolkit (KIT = yes/lucid/athena/motif/no).

- -with-xpm – use -lXpm for displaying XPM images.
- -with-jpeg – use -ljpeg for displaying JPEG images.
- -with-tiff – use -ltiff for displaying TIFF images.
- -with-gif – use -lungif for displaying GIF images.
- -with-png – use -lpng for displaying PNG images.
- -without-toolkit-scroll-bars – don’t use Motif or Xaw3d scroll bars.
- -without-xim – don’t use X11 XIM.
- -disable-largefile – omit support for large files.
- -with-x – use the X Window System.

Of course, you’ll need to have the relevant development packages installed to compile with these options, eg libpng etc.

For Xemacs, the list is even more extensive:

- -with-gtk – support GTK on the X Window System. (experimental).
- -with-gnome – support GNOME on the X Window System. (experimental).
- -with-x11 (*) – support the X Window System.
- -x-includes=DIR – search for X header files in DIR.
- -x-libraries=DIR – search for X libraries in DIR.
- -with-msw (*) – support MS Windows as a window system.
- -with-toolbars=no – don’t compile with any toolbar support.
- -with-wmcommand=no – compile without realized leader window which will keep the WM_COMMAND property.
- -with-athena=TYPE – use TYPE Athena widgets.
- -with-menubars=TYPE – use TYPE menubars (lucid, motif, or no).
- -with-scrollbars=TYPE – use TYPE scrollbars.

- -with-dialogs=TYPE – use TYPE dialog boxes (lucid, motif, athena, or no).
- -with-widgets=TYPE – use TYPE widgets (lucid, motif, athena, or no).
- -with-dragndrop – compile in the generic drag and drop API.
- -with-cde – compile in support for CDE drag and drop.
- -with-offix – compile in support for OffiX drag and drop.
- -with-xmu=no (*) – for those unfortunates whose vendors don’t ship Xmu.
- -external-widget – compile with external widget support.
- -with-tty=no – don’t support ttys.
- -with-ncurses (*) – use the ncurses library for tty support.
- -with-gpm (*) – compile in GPM mouse support for ttys.
- -with-xpm (*) – compile with support for XPM images.
- -with-png (*) – compile with support for PNG images.
- -with-jpeg (*) – compile with support for JPEG images.
- -with-tiff (*) – compile with support for TIFF images.
- -with-xface (*) – compile with support for X-Face mail headers.
- -with-gif=no – compile without the (builtin) support for GIF images.
- -with-sound=TYPE[,TYPE[,...]] (*) – compile with sound support. Valid types are ‘native’, ‘nas’, and ‘esd’.
- -native-sound-lib=LIB – native sound support library. Needed on Suns.
- -with-database=TYPE (*) – compile with database support.
- -with-ldap (*) – compile with support for the LDAP protocol.

- -with-postgresql (*) – compile with support for the PostgreSQL RDBMS.
- -mail-locking=TYPE (*)
- -with-pop – support POP for mail retrieval.
- -with-kerberos – support Kerberos-authenticated POP.
- -with-hesiod – support Hesiod to get the POP server host.
- -with-tooltalk (*) – support the ToolTalk IPC protocol.
- -with-workshop – support the Sun WorkShop (formerly Sparcworks) development environment.
- -with-socks – compile with support for SOCKS (an Internet proxy).
- -with-dnet (*) – compile with support for DECnet.
- -with-modules – compile in experimental support for dynamically loaded libraries (Dynamic Shared Objects).
- -with-netinstall – compile in support for installation over the internet. Only functional on the MS Windows platforms.
- -with-ipv6-cname=yes – try IPv6 information first when canonicalizing host names. This option has no effect unless system supports getaddrinfo(3) and getnameinfo(3).
- -with-site-lisp=yes
- -with-site-modules=no
- -package-path=PATH – directories to search for packages.
- -infodir=DIR – directory to install *Xemacs Info* manuals and dir in.
- -infopath=PATH – directories to search for Info documents.
- -moduledir=DIR – directory to install dynamic modules in.

Plus a batch of internationalization and debugging options.

Unsurprisingly, it’s pretty easy to run:

```
./configure —prefix=/usr/local/xemacs
make
sudo make install
```

Users can optionally have more extensive customization options. For example, compiling in GTK support means you can use the same theme as for the rest of your desktop – no more boring black/white/grey colours for emacs, themes and pixmaps instead.

If you intend to use Gnus as a mail/news client, it may well also be useful to make sure JPEG and PNG files are supported; emacs can display these inline, if supported when compiled. Who said emacs was just a text editor?

If compiling Xemacs, in order to have a reasonable set of elisp packages once you've started, download the *sumo* package set, which you then unpack into one of the package directories specified above.

Xemacs packages

Xemacs has a complete package system all its own, which allows you to keep all versions of installed packages up-to-date very easily.

Under the toolbar, run **Packages/Tools/Add package site**. This adds an upstream site to the list. Then you run 'Update package list', to synchronize the index of what's available upstream.

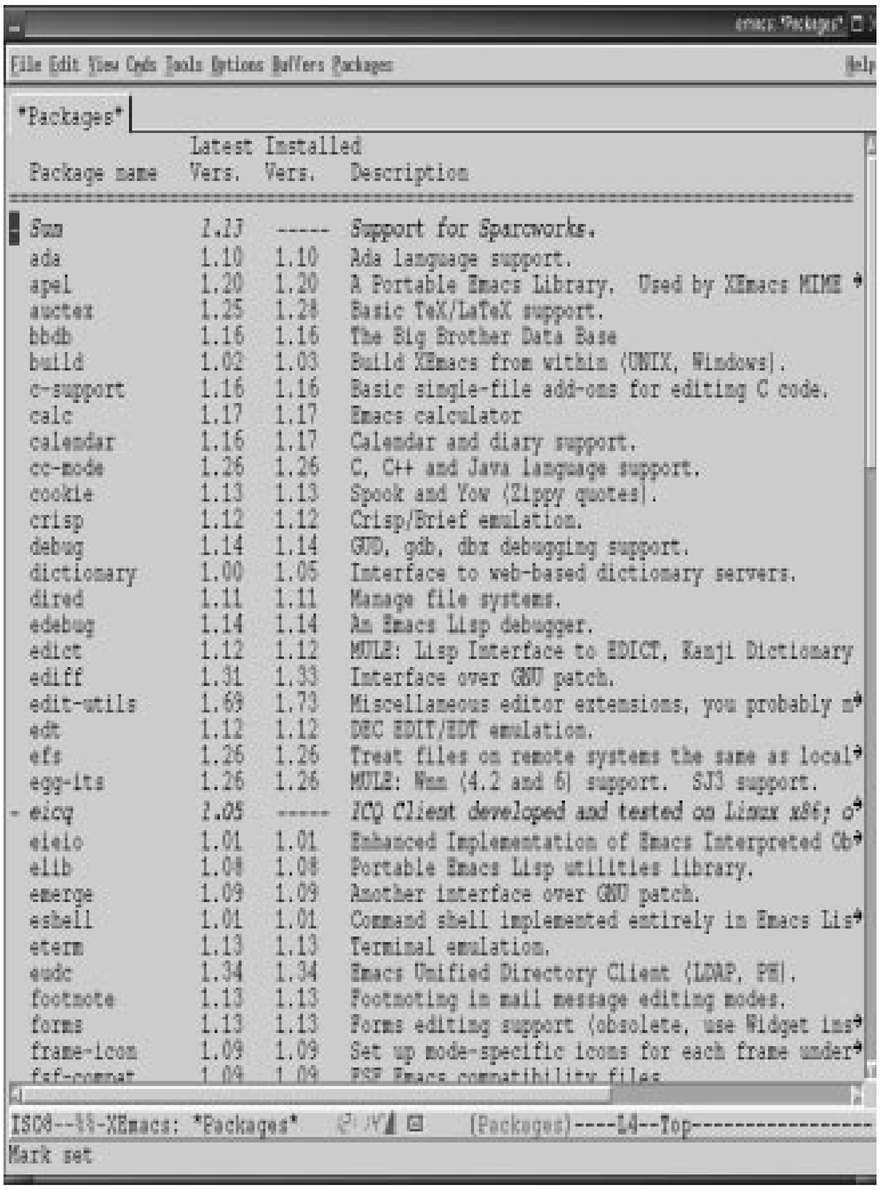
Finally, select **Packages/List & Install**, to open a buffer window with a table of packages, local and remotely available versions, and short descriptions. For each package you can elect to remove, install, or update it, and finally commit your changes with 'x'. (See the key at the bottom of the buffer.)

Figure 2 shows the Xemacs packaging system in operation.

This totally automates installing any of these packages – no more downloading tarballs, unpacking, reading install instructions!

Compiling emacs lisp files

When you load an ordinary emacs elisp '.el' file, emacs has to both load and compile it before its functions are made available. You can



```
*Packages*
                 Latest Installed
 Package name    Vers.  Vers.   Description
===============================================================================
- Sun            1.13   -----   Support for Sparcworks.
  ada            1.10   1.10    Ada language support.
  apel           1.20   1.20    A Portable Emacs Library.  Used by XEmacs MIME
  auctex         1.25   1.28    Basic TeX/LaTeX support.
  bbdb           1.16   1.16    The Big Brother Data Base
  build          1.02   1.03    Build XEmacs from within (UNIX, Windows).
  c-support      1.16   1.16    Basic single-file add-ons for editing C code.
  calc           1.17   1.17    Emacs calculator
  calendar       1.16   1.17    Calendar and diary support.
  cc-mode        1.26   1.26    C, C++ and Java language support.
  cookie         1.13   1.13    Spook and Yow (Zippy quotes).
  crisp          1.12   1.12    Crisp/Brief emulation.
  debug          1.14   1.14    GUD, gdb, dbx debugging support.
  dictionary     1.00   1.05    Interface to web-based dictionary servers.
  dired          1.11   1.11    Manage file systems.
  edebug         1.14   1.14    An Emacs Lisp debugger.
  edict          1.12   1.12    MULE: Lisp Interface to EDICT, Kanji Dictionary
  ediff          1.31   1.33    Interface over GNU patch.
  edit-utils     1.69   1.73    Miscellaneous editor extensions, you probably n
  edt            1.12   1.12    DEC EDIT/EDT emulation.
  efs            1.26   1.26    Treat files on remote systems the same as local
  egg-its        1.26   1.26    MULE: Wnn (4.2 and 6) support.  SJ3 support.
- eicq           1.05   -----   ICQ Client developed and tested on Linux x86; o
  eieio          1.01   1.01    Enhanced Implementation of Emacs Interpreted Ob
  elib           1.08   1.08    Portable Emacs Lisp utilities library.
  emerge         1.09   1.09    Another interface over GNU patch.
  eshell         1.01   1.01    Command shell implemented entirely in Emacs Lis
  eterm          1.13   1.13    Terminal emulation.
  eudc           1.34   1.34    Emacs Unified Directory Client (LDAP, PH).
  footnote       1.13   1.13    Footnoting in mail message editing modes.
  forms          1.13   1.13    Forms editing support (obsolete, use Widget ins
  frame-icon     1.09   1.09    Set up mode-specific icons for each frame under
  fsf-compat     1.09   1.09    FSF Emacs compatibility files
ISO8--%%-XEmacs: *Packages*         (Packages)----L4--Top----------------
Mark set
```

Figure 2: Xemacs packaging system in operation

precompile files in a directory with the *byte-compile-file* function. You can also run this across a whole directory with *byte-recompile-directory*. This will create a corresponding '.elc' file for the given, or all, '.el' files, as appropriate, thus saving interpretation time when reloading the file another time.

Other resources

http://unix.about.com/cs/emacs/ contains links to an emacs FAQ, manual, and lisp reference.

There is a comprehensive list of emacs lisp packages available at the ELL, to be found at http://www.anc.ed.ac.uk/~stephen/emacs/ell.html.

Gnus, the mail and news client, has a complete site of its own. There's even a portal-style site with yet more links to the tutorials, to be found at http://my.gnus.org/.

Tim Haynes
Open Source and Free Software Consultant (UK)

Hex to decimal and decimal to hex conversion

I read an article in Issue 72 of *AIX Update* entitled *Number conversion utility*, which concerned itself with hex to decimal and decimal to hex conversion. It employed a quite ingenious and rather lengthy script to achieve this.

This is the way I am used to doing these conversions:

1 Dec to hex:

```
=>printf "%x\n" 2000000
1e8480
```

2 Hex to dec:

```
=>printf "%d\n" 0x1e8480
2000000
```

This seems to me a bit more efficient.

The shell refuses to go over the 2^32 limit, (4 billion something), but I imagine that that is, for most applications, not a problem.

Rens Groenewegen (Netherlands)

AIX news

Rogue Wave Software has increased support for new platform and database clients with the introduction of the third version of Rogue Wave SourcePro C++.

The new edition is now available on AIX 5L Version 5.1, Windows XP Professional, Red Hat Linux 7.1, and HP-UX 11i.

SourcePro C++ Edition 3 now provides support to recent database releases, including Oracle9*i*, DB2 7.2, Sybase 12.5, and Informix 2.7.

It includes support for the latest XML schema specification and for the SOAP 1.2 specification.

For further information contact:
Rogue Wave Software, 5500 Flatiron Pkwy, Boulder, CO 80301, USA.
Tel: (303) 473 9118.
URL: http://www.roguewave.com/products/sourcepro/.

* * *

IBM has announced multi-year contracts and simplified ordering processes through its new Software Maintenance acquisition model, which includes technical support for IBM distributed software.

Replacing the existing Software Subscription for AIX and OS/400 and stand-alone upgrades previously ordered under each licensed program, the new scheme enables a single site or worldwide enterprise to maintain software entitlement to current or future versions or releases of the eligible products during the Software Maintenance contract period.

One year of Software Maintenance is included automatically for new licences, and a three-year option is also available.

The company also announced that Software Subscription for AIX has been restructured to become IBM Enhanced AIX Operating System Subscription, providing a one-year or three-year prepay contract. An After-Licence feature is also available that enables customers to order upgrades to the current releases or versions being marketed by IBM.

For further information contact your local IBM representative.
URL: http://www-1.ibm.com/servers/eserver.

* * *

Sybase has announced its New Era of Networks Integration Package, designed to integrate and streamline business processes and information flow between departmental systems.

It includes an integration server, New Era of Networks e-Biz Integrator, and two connection points that enable the integration of two existing business applications or systems. With each connection point, users can choose from a library of New Era of Networks Adapters designed to help integration by providing pre-defined connections to target applications or systems.

The Integration Package is available now for AIX, HP-UX, Solaris, and Windows.

For further information contact:.
Sybase, 5000 Hacienda Drive, Dublin, CA 94568, USA.
Tel: (925) 236 5000.
URL: http://www.sybase.com/products.