83

AIX update

September 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 1999 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

Disclaimer

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

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.

Printed in England.

Grep this

The **grep** utility allows one or more files to be searched for strings of words. Its syntax is somewhat similar to the regular expression syntax of the vi, ex, ed, and sed editors. It comes in three basic flavours – **grep**, **fgrep**, and **egrep**. The article mainly discusses **grep** and **egrep**, with a final note on **fgrep** at the end because it is the simplest of all three.

The name **grep** is derived from an editor command:

```
g/re/p
```

which meant "**g**lobally search for a **r**egular **e**xpression and **p**rint what you find". That action is essentially what **grep** does, so the name seemed appropriate. Regular expressions are at the core of **grep** and are covered after a brief description of some of the command options.

The simplest **grep** command is:

```
grep search-pattern files-list
```

An example of this appears in Listing 1 to search all files in the current directory for the string "hello".

Listing 1:

```
grep hello *
```

The output of this command might be something like Listing 2.

Listing 2 – output of **grep**:

```
$ grep hello *
story.txt: so I said I was thinking about saying hello when she smiled
intro.txt: the hello.c program is an example of C programming
$
```

The search by **grep** is case-sensitive. In order to change the search to include "hello", "Hello", or "HELLO" use the **-y** or **-i** option. Earlier versions of **grep** used **-y**. Later versions use **-i**, and **-y** is considered obsolete, although some versions of **grep** support both. In Listing 3, more "hellos" show up because the search is case-independent.

Listing 3 – output from a case-insensitive **grep** search:

```
$ grep -i hello *
```

```
story.txt: so I said I was thinking about saying hello when she smiled
story.txt: I could hear my echo, "HELLO, HELLO, HELLO."
intro.txt: the hello.c program is an example of C programming
hello.c:     printf("Hello, world. \n");
$
```

The output from **grep** varies depending on whether you are searching one file or several files. If only one file is named on the command line, the output does not include the file name as in Listing 4.

Listing 4 – output of a one file **grep**:

```
$ grep -i hello hello.c
    printf("Hello, world. \n");
$
```

This one file rule applies whether you use a wildcard in your file list or not. If hello.c were the only file in the current directory, using a wildcard to locate the file would still produce an unnamed file output.

As a side note, the reason for this is that the wildcard on the **grep** command line is actually expanded by the shell, not by the **grep** command.

The shell pre-expands the * by locating all files in the current directory, making a list of the files, and replacing the * with the list of files. If there is only one file in the directory, then the command is executed as:

```
grep hello hello.c
```

In Listing 5, the user is searching for any C files containing “hello”. Where there is only one file with a .c extension in the current directory the output is identical to the previous example.

Listing 5:

```
$ grep -i hello *.c
    printf("Hello, world. \n");
$
```

There is a clever work-around for this limitation. By always adding the */dev/null device* file to the list of files to search you ensure that multiple files are searched. Because **grep** accepts a list of files, the command in Listing 6 searches all files with a *.c* extension, and the file named */dev/null*. Even if there is only one file with a *.c* extension in the current directory, the output prints the file name, because it is actually searching for files.

Listing 6 – using */dev/null* as a file:

```
$ grep -i hello *.c /dev/null
hello.c:    printf("Hello, world. \n");
$
```

If you are familiar with */dev/null*, you are probably most familiar with it as a sort of character trash can – somewhere to send unwanted output. In Listing 7, errors that would normally be output to standard error (usually the terminal screen) are redirected to */dev/null*, which effectively throws them away.

Listing 7 – throwing output away in the */dev/null* waste basket:

```
find . -name \*.c -print 2>/dev/null
```

But the */dev/null* device can also be used as an input file equivalent to an empty file. This use is seen less often, but is perfectly valid.

The **-l** option can be used to extract a list of files containing the string. Only the file name is printed, and it is printed only once even though the string may appear in multiple lines. In Listing 8, story.txt appears only once even though it contains more than one occurrence of “hello”.

Listing 8 – using the -l option:

```
$ grep -il hello *
hello.c:
intro.txt:
story.txt:
$
```

The **-l** option suppresses most of the other output options from **grep**.

The **-n** option will print a line number as well as the text as in Listing 9.

Listing 9 – using **-n** to print line numbers:

```
$ grep -in hello *
hello.c:27:     printf("Hello, world. \n");
intro.txt:144: the hello.c program is an example of C programming
story.txt:101: so I said I was thinking about saying hello when she
smiled
story.txt:287: I could hear my echo, "HELLO, HELLO, HELLO."
$
```

The **-v** option outputs the complement of the search – all lines not containing the requested search pattern.

Listing 10 – output of lines not containing the search text:

```
$ grep -iv hello intro.txt
You will be able to get more practice if you
at its simplest
$
```

The **-c** option prints only a count of lines matched. It has an interesting and useful side-effect in that it lists all of the files that it searches, not just the successful hits. Listing 11 provides a sample output.

Listing 11 – using **-c** to print a count of hits:

```
$ grep -ic hello *
data.txt:0
hello.c:1:
intro.txt:1
intro2.txt:0
story.txt:2
$
```

Some versions of **grep** come with **-r** as an option to allow **grep** to recursively search through subdirectories. The default behaviour is to search only one directory and the **-r** option, as provided in GNU **grep** for example, is the exception rather than the rule.

You have now seen some of the input and output options, but the real power of **grep** is in the search pattern, which uses regular expressions.

Grep can match simple strings such as the “hello” example, but it can also use a variety of wildcards and special symbols (meta characters) to create a regular expression to search for more complex strings.

Let’s start with some of the simpler regular expression characters. A ^ (caret) character means start of the line and a $ (dollar) character means end of the line. You could use these to specify matching a word only at the beginning or end of a line.

The wildcards used by **grep** frequently clash with the special characters that the shell uses, so the usual practice is to enclose complex search strings in single quotes. Listings 12 and 13 would match any case version of *hello* at the start of a line and at the end of a line respectively.

Listing 12 – matching at the start of a line:

```
$ grep - I '^hello' *
```

Listing 13 – matching at the end of a line:

```
$ grep -I 'hello$' *
```

The . (dot or period) character will match any single character. Listing 14 will match any character followed by *ello*. It would match aello, bello, cello, on down to zello, as well as odd combinations such as 1ello, 2ello, and ?ello. Any combination of one initial character followed by *ello* is valid. The dot only matches printable characters and does not match the beginning or end of line, therefore *ello* at the start of a line would not be matched.

Listing 14 – matching any character:

```
$ grep '.ello' *
```

Optional characters can be enclosed in square brackets ([ ]) causing any of the enclosed characters to be matched. The **grep** command in Listing 15 would match hello, cello, or bello, but would not match aello.

Listing 15 – matching optional characters:

```
$ grep '[hcb]ello' *
```

Optional characters can also be specified as a range by using two characters separated by a hyphen. Listing 16 would match bay, cay, or day.

Listing 16 – matching a range of optional characters:

```
$ grep '[b-d]ay' *
```

A list of optional characters, or range of characters, can be preceded by a caret (^) to invert the sense of the match. Listing 17 would match any character followed by *ay* except the combinations bay, cay, and day.

Listing 17 – inverting optional characters:

```
$ grep '[^b-d]ay' *
```

Note that an option list or range represents a match of a single character.

Any single character match (including a single character matched by an option/range specification) can be repeated by using the repeat character, * (asterisk). An asterisk following a single character means zero or more occurrences of the preceding single character match. Listing 18 requests any line containing hello followed by dolly, where the words are separated by zero or more spaces. Note that the asterisk follows the space after hello and therefore applies to the space character.

Listing 18 – matching zero or more spaces:

```
$ grep 'hello *dolly' *
```

This search would match any of the combinations shown in Listing 19 without regard to the number of spaces between the words.

Listing 19:

```
hellodolly
hello dolly
hello                    dolly
```

The asterisk can be applied to an option list or range of characters. Listing 20 matches *c* and *t* with any number of vowels (or no vowels) in between.

Listing 20 – using repeat on an option list:

```
$ grep 'c[aeiou]*t' somewords.txt
cat
coat
coot
cot
cout
cut
ct
$
```

At this point we have to start exploring **egrep** because **grep** and **egrep** depart from one another. **Egrep** stands for **e**xtended **grep**. The POSIX 1003.2 standard defined a set of regular expression characters called modern, extended, or full regular expressions. The earlier regular expressions used by **grep** are usually called older or basic regular expressions. There is some overlap between the two types of regular expressions and recent versions of **grep** can be made to behave like **egrep** by using a **-E** option that forces **grep** to use extended regular expressions.

The **egrep** utility uses extended regular expressions. One of the useful extended regular expressions is the plus (+) character as a single character repeater. This works like the asterisk (*) but means one or more rather than zero or more. Using **egrep** in the example in listing 20 with plus (+) instead of the asterisk, the search would be changed to skip finding ***ct*** because it does not contain one or more vowels. Since this is probably what was intended in the first place, it is more useful.

Listing 21 – using + from the extended regular expressions:

```
$ egrep 'c[aeiou]+t' somewords.txt
cat
coat
coot
cot
cout
cut
$
```

If you use **grep** to achieve the same results, the search pattern becomes clumsier. Listing 22 asks for *c*, followed by any vowel, followed by zero or more occurrences of any vowel, followed by *t*.

Listing 22 – matching one or more using **grep**:

```
$ grep 'c[aeiou][aeiou]*t' somewords.txt
cat
coat
coot
cot
cout
cut
$
```

The **egrep** utility also adds a question mark (?) as yet another version of multiple occurrence matching. The ? means zero or one occurrence.

Using **egrep** with extended regular expressions:

- * = zero or more occurrences of the preceding character pattern.
- + = one or more occurrences of the preceding character pattern.
- ? = zero or one occurrence of the preceding character pattern.

The next useful character in extended regular expressions is the vertical bar (|). This creates an 'or' condition between two possible search patterns. In Listing 23, **egrep** searches for *c*, followed by one or more vowels followed by *t*. It also searches for p followed by one or more vowels followed by *l* as an option. Since the search string does not specify that the word must actually end after the closing *t* or *l*, Listing 23 has matched paula and paella as well as words that just end in *l*.

Listing 23 – searching using an 'or':

```
$ egrep 'c[aeiou]+t|p[aeiou]+l' somewords.txt
cat
```

```
coat
coot
cot
cut
cet
cit
pal
paella
paul
paula
peal
peel
pool
$
```

You can fudge this multiple optional pattern with **grep** by using a feature that all three of the **grep** utilities have. They can match multiple search patterns that have been entered by using newlines in between the patterns. This feature can be used with **egrep** and **fgrep** as well, but I am introducing it here just to show you the difficulty of imitating **egrep** with **grep** when it would be simpler to use **egrep**.

In Listing 24 the user enters the first part of the command on one line, and then presses *Enter* while the single quotes are still open. The shell prompts for additional input and continues to accept lines until the closing quote appears. Each individual line represents a separate search string to **grep**. This trick is useful with any version of **grep**.

Listing 24:

```
$ grep 'c[aeiou][aeiou]*t
> p[aeiou][aeiou]*l' somewords.txt
cat
coat
coot
cot
cut
cet
cit
pal
paella
paul
paula
peal
peel
pool
$
```

In **egrep**, simple parentheses can be used to group sections of a search

pattern together. This is usually used to cause branches within a search pattern. The example in Listing 25 illustrates this better than an explanation. The search pattern will match any of the words shown in the result list. The parentheses group '[Ss]ome' and '[Aa]ny' as optional strings followed by 'one'.

Listing 25 – using parentheses in **egrep**:

```
$ egrep '([Ss]ome|[Aa]ny)one' somewords.txt
someone
Someone
anyone
Anyone
$
```

A single character can be modified by a count bound. A count bound can specify a minimum or maximum number of characters (or both). **Egrep** uses curly braces ({}) to specify a bound, **grep** uses back-slashed curly braces (\{\}). A bound consists of one or two comma-separated numbers. The first number specifies the minimum number and the second number specifies the maximum number of preceding characters. The examples in listing 26 will clarify the usage.

Listing 26 – some example bounds:

egrep	*grep*	*Meaning*
[a-z]{2,4}	[a-z]\{2,4\}	Two to four lower-case characters.
[a-z]{4}	[a-z]\{4\}	Exactly four lower-case characters.
[0-9]{4,}	[0-9]\{4,\}	Four or more digits.
[A-Za-z]{,4}	[A-Za-z]\{,4\}	Zero to four upper- or lower-case characters.

The last useful character is the escape character or backslash (\). This character removes the special meaning of a character and turns it back into a standard character. Some simple examples are illustrated below. Note that the backslash has a special meaning, so when you want to search for a backslash itself, it must be escaped (\\). Listing 27 illustrates the different behaviour of special characters when preceded by a backslash.

Listing 27 – using the backslash:

- . – any character.
- \. – a period.
- $ – end of line.
- \$ – dollar sign.
- * – zero or more occurrences of the preceding expression.
- * –an asterisk.
- \ – nothing; it is an escape character.
- \\ – a backslash.
- | – create an ‘or’ branch between two expressions.
- \| – a vertical bar.

The definition of the escape character states that if you escape a character that does not need to be escaped, then the escape is ignored and the character is treated as if you had entered the character on its own. If you place ‘\a’ in a search pattern, it is the same as ‘a’ because it did not need to be escaped in the first place.

It is hard to remember all the characters that have a special meaning in **grep** and **egrep**. It would be wonderful if you could follow a simple rule-of-thumb such as: when in doubt about whether a character has a special meaning or not, use the backslash in front of it. Unfortunately regular expressions are not quite that regular. You have already seen that curly braces when escaped in **grep** acquire a special meaning. The same is true for parentheses and angle brackets. The following characters or combinations of characters have special meanings in **grep** or **egrep**:

In egrep, | ^ $. * + ? ( ) [{ } \.

In grep, ^ $. * \(\) [\{ \} \.

Because regular expressions are used by vi, ex, sed, and ed, it is worth mentioning that these editors use the following special characters or combinations: ^ $. * \(\) [\ \< \>.

As you can see, regular expressions aren’t as regular as they should be, and you need to be aware of which version of regular expressions you are using before you begin throwing the backslash around

indiscriminately.

The last collection of **grep** or **egrep** search pattern options is really a simple shorthand for describing a class of characters.

Listing 28 – shorthand for types of character:

- [:alpha:] – any alphabetic character.
- [:lower:] – any lowercase character.
- [:upper:] – any uppercase character.
- [:digit:] – any digit.
- [:alnum:] – any alphanumeric character (alphabetic or digit).
- [:space:] – any white space character (space, tab, vertical tab).
- [:graph:] – any printable character, except space.
- [:print:] – any printable character, including the space.
- [:punct:] – any punctuation, a printable character that is not white space or alphanumeric.
- [:cntrl:] – any non-printable character.

You may use these inside a range option. Note that the class name includes the left and right brackets, so these must be doubled inside a range as in Listing 29, which searches for any string of 10 digits or uppercase letters. Note the apparently doubled brackets. This is in fact an option of [:digit:][:upper:] inside the square brackets for a range. This could also be written as [0-9A-Z].

Listing 29 – using shorthand types:

```
$ egrep '[[:digit:][:upper:]]{10}' somenumbers.txt
1234554321
$
```

Listing 30 offers some sample search patterns. Pattern 1 searches for phone numbers by looking for an open parenthesis, followed by three digits, followed by a closing parenthesis, followed by three digits, a hyphen, and four digits.

Pattern 2 searches for zip codes either with or without the following hyphen and four digit extension. It searches for five digits followed by

zero or one hyphen followed by between zero and four digits.

Pattern 3 searches for lines containing PO Box number style addresses. It does a case-independent search for *p* followed by zero or one period, followed by zero or more spaces, followed by *o*, zero, or one period, and one or more spaces, and finally followed by box or drop. This should match most of the styles of data entry for a PO Box including P. O. Box, PO BOX, P.O. Box, P O Box, P. O. Drop, and so on.

Pattern 4 matches just the word *cat* by searching for it where it is preceded by a beginning of line or one or more spaces and followed by one or more spaces, or an end of line. This search will not match *concatenate*.

Listing 30 – some sample search patterns:

```
1    egrep '\([0-9]{3}\)[0-9]{3}\-[0-9]{4}' somenumbers.txt
2    egrep '[0-9]{5}\-?[0-9]{0,4}' somenumbers.txt
3    egrep -i 'p\.? *o\. +(box|drop)' someaddresses.txt
4    egrep '(^| +)cat( +|$)' sometext.txt
```

So far I haven't mentioned **fgrep** and you are probably wondering how it fits in with **grep** and **egrep**. **Fgrep** is **grep** (or **egrep**) without special characters. To search for a simple string without wildcards, use **fgrep**. The **fgrep** version of **grep** is optimized to search for strings as they appear on the command line, and does not treat any characters as special.

You could use **fgrep** in all of the examples that searched for the plain string 'hello', and it would be more efficient. You can also use **fgrep** to search for strings that contain special characters used in their usual sense. For example if you wanted to search for hello at the end of a sentence (hello followed by a period), you would want to search for 'hello.'.

The dot or period is a special character in **grep** or **egrep**, and it would be simpler to use **fgrep**. **Fgrep** will treat a period as a period and not as a special character.

Listing 31:

```
$ fgrep 'hello.' *
```

There are two important final notes about searching for multiple strings. Multiple search patterns can be placed on a single command line by

using the **-e** option before each search pattern. Listing 32 will search for *cat* or *dog*.

Listing 32 – using the **-e** option:

```
$ fgrep -e 'cat' -e 'dog' *
```

Possibly one of the most powerful features of the **grep** family is the ability to create a file containing a list of search patterns and then name the file on a **grep** command line with the **-f** option. Listing 33 is a file named searchfor.txt containing a list of search patterns to search for the singular or plural of various animals. The question mark at the end of each animal name applies to the preceding *s* and means zero or one occurrences of *s*.

Listing 33 – putting search patterns in a file:

```
dogs?
cats?
ducks?
snakes?
```

To use this file to search another list of files, name it on the command line with the **-f** option as shown in Listing 34 instead of a search pattern. The **egrep** utility will search for all the possible strings listed in searchfor.txt.

Listing 34:

```
$ egrep -f searchfor.txt *
```

Mo Budlong
Middleware and Data Translation Specialist
King Computer Services (USA)

Command return values

All commands, whether they be shell scripts or system commands, have an exit status, called a return value, when they finish running. The return value is available as the value of the shell variable **$?**.

When a command is run from the command line, the return value is that of the last command that was executed in the foreground. If we execute

a pipeline, then the value of **$?** is the exit status of the last command run in the pipeline.

Scripts, by their nature, contain many commands, all of which themselves have a return value and each of these can be tested within the script to determine whether certain conditions have been met. The script itself also has an exit status, which can be the return value of the last command executed within the script, or, as we shall see, a value which we can generate ourselves.

SYSTEM COMMAND RETURN VALUES

When we speak of system commands, we are referring to the group of commands described as executable (RISC System/6000) or object modules (determined by running the **file** command against them). There are also a number of shell scripts which are part of the AIX operating system, and these are also included in this section.

System commands almost always return a value of 0 when they are successfully executed. A noted exception is the **false** command, which returns a value 1; **false** and its sister **true** are used almost exclusively in shell scripts to ensure that a loop continues executing.

Most commands at least return the value 1 when they are unsuccessfully executed, but some have multiple non-zero exit statuses; these usually indicate error conditions over and above the mere failure to execute properly, such as you have specified an inaccessible file or given an invalid option to the command. There are few rules regarding command return values and you should refer to the on-line documentation for each command to be certain of what to expect.

You should be aware that the above convention in shell programming is the reverse of that used in C programming. In shell programming a value of 0 means true, or successful, and a non-zero value means false, or unsuccessful.

As a simple example, the following illustrates that the **cp** command returns 0 when it successfully copies a file:

```
$ cp search search.old
$ print $?
0
```

If, instead, we had asked **cp** to copy a file that does not exist, then **cp**

would return a value of 1:

```
$ cp a b
cp: a: A file or directory in the path name does not exist
$ print $?
1
```

The return values of **cp** make no distinction between a syntax error and some other error condition, like the one above where we tried to copy a file which did not exist. You can test this by running:

```
$ cp a b c
Usage: cp [-fhip] [-r|-R] [--] src target
. . .
$ print $?
1
```

Even though none of the files are likely to exist, the syntax check takes precedence and returns a value 1 because of the error condition.

As a further example, the **grep** command returns 0 if it finds any matches, 1 if it does not find any matches, and 2 if there are syntax errors or inaccessible files. You can verify this with the following examples:

```
$ grep answer search
```

will produce output similar to:

```
read answer
case $answer in
print "search: '$answer' is not a valid choice"
```

and the exit status will be:

```
$ print $?
0
```

In the next example, **grep** returns 1, since it does not find the string “Johann Sebastian Bach” in the file search:

```
$ grep "Johann Sebastian Bach" search
$ print $?
1
```

Finally, since the file nofile does not exist (we hope), **grep** returns the value 2:

```
$ grep no_word nofile
grep: can't open nofile
$ print $?
2
```

The **grep** command return values are by no means perfect since **grep** does not make a distinction between syntax errors (2) and trying to find a match in a non-existent file (also 2). Ideally, commands should return different values for all possible error and operating conditions, which would make life easier for shell programmers, but this is rarely the case.

RETURNING VALUES IN AND FROM SHELL SCRIPTS

Within scripts, two commands are used to provide return values:

- The **exit** command is used to define the exit status of the script itself when it finishes executing, either normally or abnormally because of some error condition.
- The **return** command is used within functions to return a particular value to the main part of a script, which is then tested to determine the next piece of code to be executed.

One major difference between **exit** and **return** is that, when **return** is used within a function, it merely exits the function and returns control to the part of the script from where the function was called, whereas **exit** will terminate the script execution, no matter how deeply it is nested within a function.

Since the variable **$?** is always reset after each command, in scripts you must save its value if you want to use it later in the script. For example, if you copy a file and want to check the exit status of the **cp** command later, you must save the exit status after you run **cp**, and before you run another command. You could, for example, use:

```
STAT=$?
```

THE EXIT COMMAND

There are several ways that a shell script can be terminated. A shell script will continue running until one of the following happens:

- The **end-of-file** is reached. This usually occurs when the last command in the file has been executed.
- The process is killed by a signal. This can happen when, for example, you press **Ctrl C** while the script is running, although signals can be trapped so as to ensure that scripts are not terminated

before they have completed a normal execution.

- An explicit **exit** command is executed.

You can use the **exit** command within a shell script to cause the script to terminate before the last command is reached. You may, for example, want the script to terminate if a particular condition is true and continue otherwise.

Type the following shell script, and then execute it, first using a filename you know to exist, and then one which does not exist:

```
$ vi example

#!/bin/ksh
print "Enter filename: \c"
read file
cp $file ${file}.old 2>/dev/null
STAT=$?

# check whether cp was successful
case $STAT in
[!Ø])
print error using cp
exit
;;
esac
print the copy was successful
```

If the copy is unsuccessful, the script prints an error message and terminates, otherwise it continues. This script serves no value other than to show the use of exit statuses and the **exit** command.

RETURNING PARTICULAR VALUES WITH EXIT

You can also use the **exit** command to make a shell script return a particular value. This is useful when a number of different conditions can occur and you want to test which one has before proceeding. The general form of the command is:

```
exit [number]
```

The *number* is optional. When used in a script, the script terminates with an exit status of *number*. If a number is not specified, the exit status which the script returns is the exit status of the last command that was run in the script.

The exit number can be any value you wish. The normal range is 0 - 255, so that exit 0 is the same as exit 256, and if *number* is greater than 256, then this is subtracted from *number* to give the exit status. For example, exit 456 will produce an exit status of 200 (456 - 256).

Consider the following example:

```
$ vi myexit

print Enter '"yes" or "no" \c'
read answer

case $answer in
yes)
      exit 1 ;;
no)
      exit 2 ;;
*)
      exit 3 ;;
esac
```

This shell script will return different exit values, depending on the response entered.

Although this particular example has no useful purpose, the principle can be used in any script to return specific values so that when *myexit*, or any script like it, is used within another script, its exit status can be tested and the second script can then be made to execute different sets of commands, depending on the value returned.

You should get into the habit of using **exit *number*** type commands in your scripts, particularly when the script can terminate because of a variety of different error conditions. There is not a great deal of use, though, in having a single **exit** command on the last line of your script since, if you do not specify *number*, the script will exit with the exit status of the last command run. If you are certain that the script has successfully executed at that stage you should always use **exit 0** to avoid confusion.

FUNCTION RETURN VALUES

Function return values are an exceedingly useful way of checking out what happens when a batch of code is executed from within a script. Like the **exit** command, the syntax is **return [*number*]**, and the same

rules apply if *number* is greater than 255. If you do not use the optional *number*, then the return value will always be the exit status of the last command run in the function.

Let us use an updated version of the **lvman** command to illustrate the use of **return** and **exit**.

```
#!/bin/ksh
# Script name: lvman
# Usage: lvman {[-v VGname]|[-p PVname]}
################################################################
# Version History
################################################################
#-----------------------------------------------
# Function: f_dsp_usage
# Displays usage messages
#-----------------------------------------------
f_dsp_usage()
{
    print "Usage: $(basename $Ø) {[-v VGname] | [-p PVname]}"
    print "Where:"
    print "\t-v VGname specifies a single volume group"
    print "\t-p PVname specifies a single physical volume"
    print "Note: Use either the -v OR -p option"
}
#-----------------------------------------------
# Function: f_chk_valid
# Arguments: $1 - volume group or physical volume
# Checks the volume group or physical volume name is valid
#-----------------------------------------------
f_chk_valid()
{
    DEV=$1
    lsattr -El $DEV >/dev/null 2>&1
    #
    # lsattr returns Ø for valid device,
    # or 255 for non valid device
    #
    case $? in
    Ø)
      return Ø ;;
    *)
      return 1 ;;
    esac
}
#-----------------------------------------------
# Function: f_get_vg_space
# Arguments: $1 - volume group name
# Gets the total and free space of the volume group
#-----------------------------------------------
f_get_vg_space()
```

```
{
    VG=$1
    #
    # Get total space and free space
    #
    TOTAL=$(lsvg $VG | grep "TOTAL PPs" | cut -f2 -d "(" |
      tr ' ' '\t' | cut -f1)
    FREE=$(lsvg $VG | grep "FREE PPs" | cut -f2 -d "(" |
      tr ' ' '\t' | cut -f1)
    eval ${VG}_LVNUM=$(lsvg -l $VG | tail +3 | wc -l | tr -d " ")
    #
    # Print output
    #
    printf "%-20s %-15s %-15s\n" "Volume Group" "Total Size" \
"Free Space"
    printf "%-20s %-15s %-15s \n" $VG "$TOTAL MB" \
"$FREE MB"
    eval print Number of LVs in $VG = '$'${VG}_LVNUM
}
#---------------------------------------------
# Function: f_get_pv_space
# Arguments: $1 - physical volume name
# Gets the total and free space on a physical volume
#---------------------------------------------
f_get_pv_space()
{
    PV=$1
    #
    # Get total space and free space
    #
    TOTAL=$(lspv $PV | grep "TOTAL PPs" | cut -f2 -d "(" |
      tr ' ' '\t' | cut -f1)
    FREE=$(lspv $PV | grep "FREE PPs" | cut -f2 -d "(" |
      tr ' ' '\t' | cut -f1)
    eval ${PV}_LVNUM=$(lspv -l $PV | tail +3 | wc -l | tr -d " ")
    #
    # Print output
    #
    printf "%-20s %-15s %-15s\n" "Physical Volume" "Total Size" \
"Free Space"
    printf "%-20s %-15s %-15s \n" $PV "$TOTAL MB" \
"$FREE MB"
    eval print Number of LVs on $PV = '$'${PV}_LVNUM
}

#########################################################
# Main section
#########################################################

getopts :v:p: opt
case $opt in
```

```
v)
    VG=$OPTARG
    f_chk_valid $VG
    case $? in
    0)
      f_get_vg_space $VG
      ;;
    1)
      print $VG is not a valid volume group
      exit 1
      ;;
    esac
    ;;
p)
    PV=$OPTARG
    f_chk_valid $PV
    case $? in
    0)
      f_get_pv_space $PV
      ;;
    1)
      print $PV is not a valid physical volume
      exit 2
      ;;
    esac
    ;;
*)
    f_dsp_usage
    exit 3
    ;;
esac
```

You will note that there is a new function, called **f_chk_valid**, which checks that the volume group or physical disk is a valid device; it does this by running the **lsattr** command against the device name, which is passed in as **$1** to the function. **lsattr** produces output, either in the form of device information or as an error message, so we have discarded this output since we are only interested in the return value of the command, which is either 0 for a valid device or 255 for an invalid one.

This function does not perform all the checks that we would like. For example, if we enter **lvman -v hdisk0**, we will get unwanted errors since, although **hdisk0** is a valid device, and **lsattr** will thus produce return value 0, it is not a valid volume group. In a future article you will see how this can be overcome using the **if** statement.

The main section now contains nested **case** statements where the

f_chk_valid function is called to determine the validity, or otherwise, of the volume group or physical volume name entered on the command line. If the function returns a value 0, then the device name is valid and the appropriate function is then called to display the space usage. If the function returns a value 1, an error message is generated and the script exits.

Each error condition generated, including the incorrect usage message, has been given a different *exit number*, which could, if desired, be checked by any other script which calls **lvman**. If no error conditions are detected, the script will execute normally with exit value 0.

Tonto Kowalski
Guru (UAE)

Communications Server failures under AIX 4.3.3

We have recently come across a situation where both Versions 5 and 6 of the Communications Server for AIX failed to start under AIX 4.3.3. The failure occurs under very specific instances. With Version 5 of the Communication Server it happens only if the bos.rte.libc fileset is at the 4.3.3.17 or later level, and the sna.rte fileset is below the 5.0.4.2 level, or below the 6.0.0.1 level for Version 6 of the Communication Server. If you think you may have this issue at your shop check the */var/sna/sna.err* file; it may contain the following errors:

- Version 5: Unable to start the TN Server executable snatnsrvr_mt
- Version 6: error reading configuration: define_tn3270_ssl_ldap.

Since the time of writing, a fix for this problem has been made available for Communications Server Version 5 (APAR IY12351) and Version 6 (APAR IY12677).

Systems Programmer (UK)

awk

INTRODUCTION

awk is an input-driven Unix utility that can be used mainly for reporting purposes (eg reporting input data file validation errors). That is, nothing happens unless there are lines of input on which **awk** can act.

This input can come from a file or another command. When you invoke the **awk** program, it reads the script that you supply, checking the syntax of your instructions. Then **awk** attempts to execute the instructions for each line of input. There is a later version of this utility called **nawk** (new awk), which offers more flow controls, functions, and a system ( ) function.

PROGRAMMING WITH AWK

Input to awk

Input to an **awk** script can be provided in the following ways:

- From the command line.
- From a file.
- From the output of another command.

Example of command line input

Command entered:

```
awk  '{print $1, $2}'
```

awk will be expecting you to type a line on the command line. Every time you type in a line, **awk** will try to print two fields which are delimited by one or more spaces, separated by comma, from the line. The program will terminate only if you press the appropriate key to terminate the program.

Example of input from file:

```
Input   file, f1
Arif    Zaman
```

Command entered:

```
awk '{print $1, $2}'  f1
```

awk will read the only line in the file as input and print out Arif Zaman and terminate.

Example of input from output of another command:

```
echo  "Arif  Zaman" | awk  '{print $1,$2}'
```

awk will accept the output from echo as its input and print out Arif Zaman.

INPUT SPECIFICATION

Record

A record in **awk** is defined as the input line, which comprises one or more words – known to awk as fields – each of which is delimited by blank spaces, tabs, or any other specified field separator.

Example:

```
Arif  Zaman
```

Record separator

A record separator is one or more characters that are appended to the end of an input line.

By default **awk** looks for a newline character as a record separator, but this can be changed by redefining the system variable, RS.

Example (using default record separator):

```
Arif  Zaman
69 The Avenue
Pinner
Middlesex
HA5  5BW
```

Note: the newline character is used here as a record separator.

Example (using redefined record separator):

```
Arif  Zaman|69 The Avenue|Pinner|Middlesex|HA5  5BW
```

Note: the pipe symbol, |, is used here as a record separator.

Referencing record

Within a script, the current record can be referenced using $0.

Field

awk interprets a record as comprising one or more fields, which are separated by spaces, tabs, or any other field separator. This default can be changed by redefining the system variable, FS.

Field separator

A field separator is one or more characters that are appended to the end of a field in a record. By default, **awk** looks for one or more white spaces.

Example (using default field separator):

```
Arif      Zaman
```

Note: the white space characters are used here as a field separator.

Example (using redefined field separator):

```
Arif|Zaman
```

Note: the pipe symbol, |, is used here as a field separator.

Field operator

The specific field in a record can be referenced by $<numeric no>, where $ is known as a field operator and *numeric no* is the field in question.

Referencing field

awk allows you to reference fields in actions using the field operator $. This operator is followed by a variable that identifies the position of a

field by number. $1 refers to first field, $2 to the second field, and so on.

Example (using default field separator):

```
Input  Line
Arif  Zaman       666-555-1111
```

awk '{print $2, $1, $3 }' will output Zaman Arif 666-555-1111

The commas that separate each argument in the print statement cause a blank space to be output between the values.

Example (using a specified field separator):

Input line:

```
Arif Zaman,69 The Avenue, Pinner, Middlesex,HA5 5BW
```

awk command:

```
awk -F"," '{print    $1
                     print    $2
                     print    $3
                     print    $4
                     print    $5 }'
```

Output:

```
Arif  Zaman
69 The  Avenue
Pinner
Middlesex
HA5 5BW
```

Formatted output

Only a limited amount of formatting can be achieved with the **print** command. **awk** offers an alternative to the **print** statement, **printf**, which is borrowed from the C programming language.

Example:

Printing a record, right-justified by 20 characters:

```
echo  "Arif  Zaman" |  awk  '{printf ("%20s\n",$0 )}'
```

Output:

```
         Arif  Zaman
```

Printing a record, left-justified by 20 characters:

```
echo  "Arif  Zaman" |  awk  '{printf ("%-20s\n",$0 )}'
```

Output:

```
Arif  Zaman
```

Printing fields, left-justified by 10 characters:

```
echo  "Arif  Zaman" |  awk  '{printf ("%-10s %-10s\n",$1,$2 )}'
```

Output:

```
Arif      Zaman
```

Printing fields, right-justified by 10 characters:

```
echo  "Arif  Zaman" |  awk  '{printf ("%10s %10s\n",$1,$2 )}'
```

Output:

```
      Arif    Zaman
```

Printing fields, left-justified by 10 characters with additional formatting:

```
echo  "Arif  Zaman  02088687985" |  awk  '{printf ("First  Name==>%-
10sLast  Name==> %-10sTel No==>%-10s\n",$1,$2,$3 )}'
```

Output:

```
First Name==>Arif      Last Name==>Zaman    Tel No==>02088687985
```

Printing integer fields:

```
echo  "Arif  Zaman    40" |  awk  '{printf ("First  Name==>%-10sLast
Name==> %-10sAge==>%d\n",$1,$2,$3 )}'
```

Output:

```
First Name==>Arif      Last Name==>Zaman    Age==>40
```

Printing number fields:

```
echo  "2.50" |  awk  '{printf ("Price = £%10.2f ",$1 )}'
```

Output:

```
Price =       £2.50
```

Note: the newline character, *n*, must be provided exclusively to the **printf** statement.

SYSTEM VARIABLES

There are a number of system or built-in variables defined in **awk**. **awk** has two types of system variable. The first type defines values whose default can be changed, and the second type stores values that can only be referenced.

FS (Field Separator)

FS is more precisely an input field separator. This is used to scan the input line and therefore is modifiable. By default, it is a space or a tab.

Example:

- FS = \t defines FS to be one tab.
- FS = \t+ defines FS to be one or more tab.
- FS = [‘:\t] defines FS to any combination of ‘ , : or a tab.

OFS (Output Field Separator)

OFS is equivalent to FS, but is used to define a field separator for output and is a space by default.

This can be changed.

NF

awk defines NF as the number of fields for the current input record. This variable cannot be redefined.

NR

awk defines NR as the number of the current input record. This variable cannot be changed.

RS

awk defines RS as the record separator. The default value is a newline which can be changed.

ORS

ORS is the equivalent to RS, but used to define a record separator for

an output record. The default is also a newline.

FILENAME

The variable FILENAME contains the name of the current input file. This variable cannot be changed.

Comments in script

All comments must start with # (hash) as in other shell scripts.

Script file extension

awk recognizes a file with an *.awk* extension to be an **awk** script. If the script file has a different file extension, it must be invoked with the **-f** option (eg **–f script.awksource**)

Program construct

An **awk** script has three sections, as shown below.

The code in the initialization section will be executed only once.

The code in the main body will be executed as many times as there are input records.

The code in the end section will be executed only once.

```
INITIALIZATION  SECTION
#
#  The initialization section is implemented using the keyword BEGIN
#  Code here will be executed only once
#
FUNCTION   DEFINITION
#
#  Define any user functions here
#  this is a feature of nawk
#
MAIN   BODY
#
#  main body contains any legal commands
#  Code here will be executed once for each line of input to awk
#

END   SECTION
#
```

```
# The end section is implemented using the keyword END
# Code here will be executed only once
#
```

Example recctr.awk (print out the total no of input records):

```
BEGIN  {
         TOTAL_NO_RECORDS=Ø   #  initialize  the  counter
               }
#
function  report_header()  {

printf("Record         Details\n")
printf("==============\n")

}
#
#
{
     TOTAL_NO_RECORDS = TOTAL_NO_RECORDS + 1   #  increment  the
counter for  each  input  record  read
}
#
#  print  report  header
#
NR == 1  { report_header() }
{

     print  NR, $Ø   #  print  out each record   with  record  no
}

END  {
            Print  TOTAL_NO_RECORDS   #  print  out  the  total
          }
```

Notes:

1 The key words BEGIN and END and opening curly brace must be placed on the same line and there must also be an ending curly brace.

2 All the commands in the main body can be put within one or more pairs of curly braces.

VARIABLE ASSIGNMENTS

Variables are assigned without declarations.

Example (number assignment):

```
X   =  1
Y    =  2
LOOP  =  1
result = X  +   Y
result  =  X *  Y
```

Example (string assignment):

```
NULL = ""
```

Example (concatenating variables):

```
fname  =  "Arif"
lname  =  "Zaman"
Name   = fname "   " lname
```

PROGRAM FLOW CONTROL

Conditional statements

The syntax of a conditional statement is:

```
         If  ( expression)
                    action1
          else
                   action2
```

or:

```
     if  (  expression )  {
                  action1
                  action2
           }
```

or:

```
if  (  expression )      {
                  action1
                  action2
           }

else  if  (  expression )  {
                                          action1
               action2
               }
 else
               action1
```

Notes:

1 The else clause is optional.

2 If the expression evaluates as true (non-zero), action(s) are performed.

3 The variables are referenced using the variable name only and not using the field operator, $.

If the variable *i* has been assigned a value of 1, **awk** will interpret $i as field one and the statement 'print $i' will try and print the value of field 1 within current record. In order to print the value of variable *i* , use **print i**.

For example:

```
If  (x)
        print  x
```

If *x* is zero (or undefined), the **print** statement will not be executed.

Example:

```
If  (  avg  >=   65  )
      grade =  "Pass"
else
      grade = "Fail"
```

LOOPING

While loop

The syntax is:

```
               while  ( condition )
                    action
```

Example:

```
i = 1

while (  i <=  4  }
      {
          print  i
           i++
```

```
}
```

Note: the body of the loop may or may not be executed, depending on the initial value of *i*.

Do loop

For example:

```
i = i
do  {
          print   i

}  while (  i <= 4 )
```

Note: the body of the loop will be executed at least once.

For loop

The syntax is:

```
          for  ( set_counter; test_counter; increment_counter )
         action
```

Example:

Print out all the fields in current record in reverse order:

```
for ( i = NF ;   i >= 1;  i–  )
    print $1
```

OTHER STATEMENTS

break

The break statement breaks out of a loop such that no more iterations of the loop are performed.

Consider the following construct:

```
For  (  i =  1;  i  <  NF;  ++i  )
         if  (   field1  ==  $i  )
                    {
                       print   field1,  $i
                             break
                      }
```

A loop is set up to examine each field of the current input record. We're interested in printing out the first field value and therefore, as soon as *$i* matches field1, the loop is terminated by a break statement.

continue

Consider the following construct:

```
For  (  i =  1;  i <  NF;  ++i  )
        if  (   i  ==  3   )
                   continue
        print   i,  $i
```

A loop is set up to examine each field of the current input record and print them out. However, we're not interested in printing out the third field and therefore, as soon as *i* is equal to 3, the **continue** statement is executed to return the control to the top of the loop.

Consider the following construct:

```
{
   #  read  all    input  records  from  the  file  and
   # store  them  into  an  array
   #
   RECORD=RECORD  $Ø  "\n"
   next   # read  next  record
  }
END  {
               #  now  print  all  read  records
       print    RECORD

           }

exit
```

The exit statement exits the main input loop and passes control to the END rule, if there is one. If the END rule is not defined, or the exit statement is used in the END rule, then the script terminates.

The exit statement can take an expression as an argument. The value of this expression will be returned as the exit status of **awk**. If the exit status is not supplied the exit status is 0.

ARRAYS

An array is a variable that can be used to store a set of values.

The syntax is :

```
Array[subscript]=value
```

For example:

```
BEGIN  {
        Name [Ø] = "Arif  Zaman"
   Name[2]  = "Henry Paul"
                #
                 print  Name[Ø]
                 print  Name[1]
              }
```

Using system variable NR as a subscript

NR keeps the running total of number of input records that are being read. Therefore, we can use this as a subscript to store the corresponding record in the array.

For example:

```
#
#  body   of  the  program
#
{
     RECORD[NR] = $Ø   # this will store record corresponding to NR
into  this  array

}
```

Reading elements of an array variable

There is a special looping syntax for reading all the elements of an array:

```
For  (  rec  in   RECORD  )
        print RECORD[rec]
```

where *rec* is any variable and *RECORD* is the array in question.

ARRAY SYSTEM VARIABLES IN NAWK

Nawk provides system variables that are arrays.

ARGV

ARGV is an array of command-line arguments, excluding the script

itself, and any options specified with the invocation of **awk**. The number of elements in this array is available in **ARGC**.

The index of the first element of the array is 0 and the last is ARGC – 1.

ARGV contains arguments that will be passed on to the script.

For example:

```
BEGIN {
                    for ( ind = 0 ;  ind <  ARGC ;  ind++ )
                            print  ARGV[ind]
               }
```

ENVIRON

ENVIRON is an array of environment variables. Each element in the array is the value in the current environment and the index is the name of the environment variable.

Example (reading all environment variables):

```
BEGIN {
                  for   ( env  in  ENVIRON  )
                              print  ENVIRON[env]
             }
```

Example (reading specific environment variables):

```
BEGIN {
          print  ENVIRON[ "DISPLAY" ]
              }
```

Note: the index to an array element is the name of the variable. This is known as an associative array, which is explained below.

Example (changing environment variable):

```
print  ENVIRON[ "TERM"]    ==> Vt100
```

Change this to Vt200:

```
ENVIRON[ "TERM" ] = "Vt200"
```

Note:this change won't affect the user's actual environment; once **awk** terminates, the TERM variable will have the value Vt100.

ASSOCIATIVE ARRAYS

In **awk**, all arrays are associative arrays. What makes an associative array unique is that its index can be a string or a number.

For example:

```
acro [ "BA"] = "British  Airways"
acro [ "BB" ] = "Bangladesh  Biman"
```

Execution of statement using relational operators

The relational operators are:

< Less than

> Greater than

<= Less than or equal to

>= Greater than or equal to

== Equal to

!= Not equal to

~ Matches

!~ Does not match.

The syntax is:

```
    Comparison statement  {  action  }
```

Example – if a record has five fields, then print that record:

```
NF == 5  {  print   $Ø  }
```

Example – if field5 matches regular expression /NA/ , then print Not Applicable for this field:

```
$5  ~  /NA/  { printf ("Not Applicable") }
```

Example – if NR is 1, then print the report header by making a function call:

```
NR == 1  {  report_header()  }
```

Execution of statement using Boolean operators

The Boolean operators are:

|| Logical OR

&& Logical AND

The syntax is:

Example – if NR is 5 and NF is 2 then print the field:

```
NR == 5 && NF == 2 { print $2 }
```

Example – if field1 is null or field2 is null then print error missing:

```
$1 == "" || $2 == "" { printf("Mandatory Fields missing") }
```

AWK BUILT-IN FUNCTIONS

Integer functions

The int( ) function truncates a numeric value by removing digits to the right of the decimal point.

Example:

```
print  100/3      == >  33.333
print  int ( 100 / 3)   ==> 33
```

The int () function simply truncates; it does not round up or down.

String functions:

- Index (s,t) – returns position of substring *t* in string *s* or zero if not present.
- Length (s) – returns length of string *s*.
- Split (s, a, sep) – parses string *s* into elements of array *a* using Field separator *sep*; returns number of elements. If *sep* is not supplied, FS is used.
- Sprintf("fmt",expr) – it uses the same format specification as printf( ), but the format specification is applied on a string.
- Substr(s,p,n) – returns substring of string *s* at beginning position

p up to a maximum length of *n*. If *n* is not supplied, the rest of the string form *p* is used.

Nawk functions:

- Gsub(r,s,t) – globally substitute *s* for each match of the regular expression *r* in the string *t*. Returns the number of substitutions. If *t* is not supplied, it defaults to $0 (current record).
- Match(s,r) – returns either the position in *s* where the regular expression *r* begins, or 0 if no occurrences are found. It sets the value for RSTART and RLENGTH.
- Sub(r,s,t) – substitutes *s* for the first match of the regular expression in the string *t*. It returns 1 if successful, 0 otherwise. If *t* is not supplied it defaults to $0.

Example of index ( ):

```
pos  =  index ("Catwalk", "walk" )
```

The value of pos is 4:

```
var1="Strongman"

pos =  index (var1,"man")
```

The value of pos is 7.

```
pos  =  index("Bigworld", "is" )
```

The value of pos is 0.

Example of length( ):

```
curreclen =  length($0)
```

The value of curreclen will be set to the length of the current record.

```
field2len  =  length ( $2)
```

The value of field2len will be set to the length of field two from the current record.

Editor's note: this article will be concluded next month.

Arif Zaman
ETL Developer (UK)

Understanding the cp, mv, and rm commands

cp, mv, AND rm COMMAND BASICS

The **cp**, **mv**, and **rm** commands are some of the most basic commands used by AIX users and administrators. You have probably used these commands (which can copy, move, rename, and remove files) as a regular part of your activities. The intention of this article is to describe some of the flags used with these commands to ensure that you are getting as much as possible from them.

The **cp** command can copy files and directory structures. **mv** can move files and directory structures. You can also use the **mv** command to rename a file, which, in essence, is telling the command processor, "Move this file to the same directory, only call it this". The **rm** command can remove files and directory structures.

The basic syntax of the **cp**, **mv**, and **rm** commands is shown below:

```
cp    flags   sourcepath  targetpath
mv    flags   sourcepath  targetpath
rm    flags   path
```

where:

- *flags* is an optional flag or flags used to enhance the operation.
- *sourcepath* is the path of the files to be copied, moved, or renamed.
- *targetpath* is the destination of the files to be copied, moved, or renamed.
- *path* is the path of the files to be removed.

FLAGS FOR THE cp, mv, AND rm COMMANDS

Now let's look at the flags for the commands.

cp command flags:

- *-f* – forces a copy despite mode restrictions that would prohibit the operation. For example, if the target file exists already and you have

no write access to it, the default behaviour of the **cp** command would be to issue an error message. However, **cp** will force the overwrite of the file if the *-f* flag is used.

- *-h* – copies a symbolic link. The default behaviour of the **cp** command is to copy the contents of a linked file if a symbolic link is encountered. Using the *-h* flag causes the **cp** command to actually copy the LINK rather than the linked file. This would be useful if you want multiple pointers to the same file.
- *-i* – prompts before overwriting a file. If a file to be copied exists already in the target directory, specifying the *-i* flag will get you a prompt before the file is overwritten.

  This could be useful to detect the unexpected existence of files for diagnostic purposes. For example, let's say you had a directory with 300 files in it. You also have a dozen files you want to add. Your goal is to determine whether any of the twelve files are already in the directory. If you were to specify the *-i* flag on the **cp** command, each of any files existing already in the directory would be indicated by a prompt. If no prompt appears, then none of the files pre-existed in the directory.
- *-R* – recursive copy. The *-R* (uppercase R) flag allows the **cp** command to copy entire directory structures to the target location.

mv command flags:

- *-f* – forces a move or rename despite mode restrictions that would prohibit the operation. Similar to the **cp** command.
- *-i* – prompts before overwriting file. If a file to be moved or renamed exists already in the target directory, specifying the *-i* flag will get you a prompt before the file is overwritten.

rm command flags:

- *-e* – displays a message naming each file after it has been removed from the directory. This could be useful if the **rm** command is writing to a log so you can track successful file removal for diagnostic purposes.
- *-f* – forces removal despite mode restrictions that would prohibit

the operation. The *-f* flag suppresses the prompt asking whether you want to remove a read-only file. This would be useful if you were issuing **rm** commands from automated scripts.

- *-i* – prompts before removing each specified file. This is useful when using wildcard characters in the file specification, specially when attempting to remove a specific subset of files from a directory containing many files. For example, suppose you had hundreds of files generally named as follows:

```
ABC20000.TXT
ABC20010.TXT
ABC20020.TXT
ABC20030.TXT
ABC20040.TXT
ABC20050.TXT
ABC20060.TXT
...
ABC90090.TXT
```

  If you were to enter a command such as **rm *200*.TXT**, you may find yourself inadvertently removing a file such as ABC42000.TXT which you may have wanted to keep. However, the command **rm -i *200*.TXT** would resolve the wildcard characters and prompt you before removing each file, thus allowing you to selectively verify each of the subset of files found matching the specification.

- *-R* – recursive removal. The *-R* (uppercase R) flag allows the **rm** command to remove entire directory structures and all their contents. Be careful with this command! Unlike other operating systems that require confirmation before deleting complete directory structures, AIX will allow you to do so and give you nothing more than your cursor back after it has completed.

SOME EXERCISES

Here are some exercises to test your knowledge of the commands.

Exercise set-up

1 From a directory to which you have write access, enter the following commands:

```
mkdir cp_mv_rm    (to make a new directory for these exercises)
cd    cp_mv_rm    (cd into the directory)
mkdir recurse (to create a directory structure for recursive steps)
```

2 Create a test file called exer1 by copying an existing small file or by entering the command **ls -l > exer1**.

3 Populate the exercise directory structure by entering the following **cp** commands:

```
cp exer1 exer2
cp exer1 recurse/exer3
cp exer1 recurse/exer4
cp exer1 recurse/exer5
```

This will give you five copies of your test file to start the steps of the exercise.

Exercise

Set up back-up and recovery:

- Step 1 – enter **cd ..** to put you in the parent directory of cp_mv_rm.

  Enter **cp -R cp_mv_rm exersave**. This will copy the exercise directory structure into a new directory called exersave in case you need to start over.

- Step 2 – in the event you need to recover the exercise directory structure, **cd** to the parent of the exercise structure and enter **rm -fR cp_mv_rm** to force deletion of the contents of the directory and the directory itself. Then enter **cp -R exersave cp_mv_rm** to recreate the original directory structure and start the exercise over. (Note: the steps build upon each other so be sure to start from Step 3 or you may not have the necessary files in place to continue.)

cp -f flag:

- Step 3 – **cd** back into cp_mv_rm to continue the exercise. Enter **chmod 444 exer2** to set the file to read-only. Enter **ls -l** to verify the write mode if you desire.

- Step 4 – enter **cp exer1 exer2** and note the file access permissions error displayed.

- Step 5 – enter **cp -f exer1 exer2** and note the return of your cursor. **ls -l** will verify that the file copy has been forced by the *-f* flag and that exer2 now has the same write permissions as exer1. You have used the *-f* flag to force a copy over read-only restrictions.

cp -h flag:

- Step 6 – enter **ln -s exer1 exer6** to create a symbolic link to exer1. Use **ls -l** to verify that exer6 is just a pointer to exer1.
- Step 7 – enter **cp exer6 exer7** and use **ls -l** to see that exer7 is the same size as exer1. It should be, because the **cp** command has copied the contents of the linked file (exer1) rather than the link.
- Step 8 – enter **cp -h exer6 exer8** and use **ls -l** to see that now you have two links to exer1, namely exer6 and exer8. You have used the *-h* flag to copy a symbolic link.

cp -i flag:

- Step 9 – enter **cp -i exer1 recurse/exer3** and note the overwrite prompt displayed. Enter *y* at the prompt to overwrite the exercise file. **ls -l** recurse will verify that it has been overwritten because the timestamp will be later than when you originally created the file. You have used the *-i* flag to prompt before overlaying a file being copied.

mv -f flag:

- Step 10 – enter **chmod 444 exer7** to set the file to read only. Enter **mv exer2 exer7** to attempt to rename a file to a name that already exists. Note that the overwrite prompt appears. Enter *n* at the prompt to return your cursor.
- Step 11 – enter **mv -f exer2 exer7**. **ls -l** will verify that the **mv** command has renamed exer2 to exer7. You have used the *-f* flag to force a rename over read-only restrictions.

mv -i flag:

- Step 12 – enter **mv exer7 exer3** to rename the file.
- Step 13 – enter **mv -i exer3** recurse and note that the overwrite

prompt appears. Enter *y* at the prompt to overwrite the file. Use **ls -l** to verify the overwrite. You have used the *-i* flag to prompt before overlaying a file being moved.

rm -f flag:

- Step 14 – enter **chmod 444 recurse/exer**3 to set the file to read-only.
- Step 15 – enter **rm recurse/exer3** and answer *n* at the overwrite prompt.
- Step 16 – enter **rm -f recurse/exer3**. Enter **ls -l** recurse and note that you have used the *-f* flag to force removal of a file with read-only access.

rm -i flag:

- Step 17 – enter pwd to ensure you are in the cp_mv_rm directory. Enter **rm -i exer*** and note you get a prompt for each file in the directory. Answer *n* for each file. You have used the *-i* flag to prompt before the removal of files.

rm -e and **-R** flags:

- Step 18 – enter **cd ..** to put you in the parent directory of cp_mv_rm. Enter **pwd** to verify.
- Step 19 – this step will remove all the files you have created in this exercise! Enter **rm -eR cp_mv_rm** and note that, as each file and directory is removed, a message to that effect is displayed. You have used the *-e* flag to indicate each object removed and the *-R* flag to remove an entire directory structure. **ls -l** will show that the only data left is exersave.
- Step 20 – if you are certain you are satisfied with your progress in this exercise, you may remove the exersave directory structure by entering the command **rm -R exersave**.

David Chakmakian
Software Engineer (USA)

AIX news

IBM has announced new versions of its C, C++, and FORTRAN languages for AIX. IBM C for AIX V6.0 supports the latest C99 standard, 32-bit and 64-bit application development, partial GNU C portability support, generation of highly optimized code for all RS/6000 processors, and new compiler options and pragmas.

It also includes the IBM Distributed Debugger, to provide visual debugging of programs running locally, remotely, or in a client/server environment.

Meanwhile, VisualAge C++ Professional for AIX V6.0 gets improved portability through support in C++ for the OpenMP specification, support for 32-bit and 64-bit application development, and enhanced template handling for faster compilations and the generation of smaller objects.

There's inclusion of the C for AIX compiler at the latest C99 standard, support in C++ for the latest approved clarifications of the ISO 1998 C++ Standard, partial GNU C/C++ portability support, generation of highly optimized code for all RS/6000 processors, new compiler options and pragmas, and the Distributed Debugger.

Finally, XL FORTRAN (XLF) for AIX V8.1 features full functionality of V7.1.1, support for the OpenMP FORTRAN API V2.0, and support for select features of FORTRAN 2000, including allocatable components, IEEE Floating Point Exception Handling, and pointer with INTENT attribute.

There's also support for the full FORTRAN 95 standard and SMP programming, support for all RS/6000 processors, new performance enhancements, and enhanced porting features.

For further information contact your local IBM representative.
URL: http://www.ibm.com/software/ad.

* * *

IBM has announced Version 4.5 of its HACMP high-availability software with better usability and performance, easier configuration, and additional hardware support for Cluster 1600, pSeries, and RS/6000 users.

The software is designed to detect system failures and handle failover to a recovery node gracefully, providing continuous application availability.

New functions include reduced failover time, streamlined configuration process, automated configuration discovery, improved security for cluster administration, persistent IP address support, and enhanced WAN and X.25 support.

Among the enhanced scalability features, there's easier configuration with AIX Enhanced Concurrent Mode, a new application availability analysis tool, tighter integration with GPFS V1.5 Cluster file system, monitoring and recovery from loss of volume group quorum, support for multiple applications on each network adapter, and 64-bit-capable APIs.

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