96

AIX

October 2003

In this issue

update

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.

Editor

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 £100 ($160) per 1000 words and £50 ($80) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £20 ($32) per 100 lines. 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.

Handling command line parameters using getopts

When creating robust shell scripts, remember that they should ideally be capable of handling command line parameters or options, checking user input, and trapping and recovering from errors. In this article I will look at handling multiple command line arguments using **getopts**. **Getopts** is an in-built shell function that takes away from the system administrator or user the tedious tasks of checking correctly-parsed parameters.

When parsing parameters to a script, the most standard form is to prefix the parameter name with a hyphen (-), followed by the actual value or argument. For instance to **tar** up the directory */usr/local* with a verbose option is:

```
$ tar -xvf /usr/local/ /dev/rmtØ
```

Where f is the name of the files or directories.

Of course **tar** is a binary file, but, by using **getopts**, your scripts can handle these types of argument.

The basic format of **getopts** is:

```
getopts [options] opstring name
```

Getopts is used within a while loop, usually with a few case statements thrown in, depending on the complexity of the actual script. A very simple script, sampleopt.sh, using **getopts** is shown below:

```
#!/bin/bash
while getopts pvf opt
do
    case "$opt" in
        p) PRINT="$OPTARG"
        echo "PRINT is selected"
        ;;
      v)  VERBOSE="$OPTARG"
              echo "VERBOSE is selected"
;;
      f)  FILENAME="$OPTARG"
          echo "FILENAME is selected"
;;
```

```
      \?) echo "Invalid Option"
                    exit 1;;
     *) echo "Unknown Options"
     exit 1
;;
    esac
done
```

The script takes the arguments p, v, and f. These values are the opstring, and the name of the variable is $opt. When values are parsed to the sampleopt.sh script from the command line, the valid options are printed.

Running the above script with different options could produce the following:

```
$ sampleopt -v -p
VERBOSE is selected
PRINT is selected
$ sampleopt -v -c
VERBOSE is selected
sample1: illegal option -- c
Invalid Option
```

UNDERSTANDING GETOPTS

Looking more closely at the listing, we see that **getopts** reads the opstring and knows these are valid options to be used in the script. It will look for all arguments that start with a hyphen. When an option that contains a value is read in, it is assigned to the special variable $OPTARG. If a match is found, then the **getopts** variable $OPTARG is set to $opt. This process is repeated using a while loop until there are no more options left. When **getopts** has finished processing all the arguments, it returns a non-zero status, which means that all arguments were parsed successfully. The special variable $OPTIND holds the next argument that is to be processed. When **getopts** is initially run it is set to 1. We will discuss $OPTFIND shortly.

There is a lot that can be done to the script to make it handle different arguments. For instance, the print option should be set to **off** by default; only when the user parses **-p** do we want to print. If the operation is to be verbose, the user should be able to

choose whether it is **on** or **off**; this option should be parsed to the script. If no verbose option is parsed then a default value should be set. When a filename is parsed we need to check that the file does exist; indeed, the script should be able to handle more than one file.

When using **getopts** you can specify what options require a parameter by using the colon. A colon precedes an option that must have a parameter or value. So the following **getopts** snippet:

```
getopts pv:f: opt
```

would mean that the options **v** and **f** require a value parsed with them and are compulsory, but the **p** option is optional. When **getopts** encounters an error it will print a rather unfriendly message, as in the previous example (illegal option -c). To suppress these messages put a colon before any of the options, like so:

```
getopts :pv:f: opt
```

This colon will also parse any unmatched options to the variable '?', allowing a case statement to trap these errors. Generally the ? character is used to produce a usage statement, but you are not forced to do so. You can use the standard **-help** option if you wish by putting a pattern match in a case statement.

CHECKING WITHIN GETOPTS

When dealing with **getopts** errors, it is best to throw up a usage statement to the user, showing all the available parameters. Not only is this the standard, it is considered good form. The usage statement should be enclosed as a function, thus enabling you to call it from anywhere in your script. Because this is an error call, the script should exit with a status 1. A simple usage function for our script would be:

```
usage()
{
echo -e "usage: `basename $Ø` [ -p to print] [-v on or off]  -f
filename"
echo " -p print file defaults to no"
```

```
echo " -v on for verbose, off for silent, defaults to on"
echo " -f filename"
exit 1
}
```

Users who do not know the different parameters or arguments of a script sometimes just type the script name and hope to see a usage statement – we must not disappoint these users. Using a test condition, we can tell whether the shell variable **$#** is empty. If it is, then we print a usage statement.

The **$#** variable holds the total number of parameters being parsed to the current script:

```
if [ $# = Ø ]; then
    echo "No parameters found"
    usage
fi
```

Handling default values is simple enough, just put these default values in your script before any main input processing is started.

```
PRINT=Ø
VERBOSE=on
```

If the user wants to have verbose on, then we need to check whether the user has passed the correct value. Using a simple case statement will accomplish this quite nicely – notice the use of different ways of writing **on** and **off** so as to trap most if not all possible ways. If the value passed does not match our case pattern match, then the script throws up a usage statement and exits:

```
   case "$VERBOSE" in
           on|On|ON) ;;
          off|Off|OFF) ;;
           *) echo "Verbose option must be 'on' or 'off'"
             usage;;
          esac
```

Checking for a valid filename passed is quite easy – all we need to do is check that the file actually exists and has a size greater than zero bytes. If none of these conditions is met using the test operator **-s**, then we exit with a usage statement:

```
           if ! [ -s "$FILENAME" ]; then
```

```
            echo "File '$FILENAME' does not exist"
            usage
           fi
```

To handle multiple files, simply put a ‘for’ loop around the file option parameter:

```
     for FILENAME in $FILES
             do
# file exist & size > Ø bytes
           if ! [ -s "$FILENAME" ]; then
            echo "File '$FILENAME' does not exist"
            usage
           fi
          done
```

One shortfall that **getopts** suffers from is that arguments cannot have spaces in them. At best it will treat this as an undefined option, at worst it will ignore it completely. When parsing more than one value as an option enclose it with quotes, like so:

```
-f "/home/dxtans/report.txt /home/dxtans/monthly.txt"
```

To correctly process all the options using **getopts** we must shift to the next value being processed. As the variable $OPTIND holds the next current argument, all we need is a simple shift statement directly after the ‘while’ loop, so we are dealing with the current argument being processed:

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

Now with the above topics discussed we can put together a more robust script, which handles command line options quite nicely. The complete script is shown below:

```
$ sampelopt -?
usage: sampelopt [ -p to print] [-v on or off]  -f filename
 -p print file defaults to no
 -v on for verbose, off for silent, defaults to on
 -f filename
Processing three files with verbose on
$ sampelopt -f "/home/dxtans/file1.txt /home/dxtans/file2.txt /home/
dxtans/file3.txt"  -v off -p
processing ...
PRINT is 1
VERBOSE is off
FILENAME is /tmp/file1.txt
=========
```

```
processing ...
PRINT is 1
VERBOSE is off
FILENAME is /tmp/file2.txt
=========
processing ...
PRINT is 1
VERBOSE is off
FILENAME is getdt2
=========
processing ...
PRINT is 1
VERBOSE is off
FILENAME is dx
=========
Another typical use could be;
$ sampleopt.sh  -p  -v on  -f "/tmp/dt1 /tmp/dt3"
processing ...
PRINT is 1
VERBOSE is on
FILENAME is /tmp/dt1
=========
processing ...
PRINT is 1
VERBOSE is on
FILENAME is /tmp/dt3
=========
Listing 2. sampleopt.sh
#!/bin/bash
# sampleopt
usage()
{
echo "usage: `basename $Ø` [ -p to print] [-v on or off]  -f filename"
echo " -p print file defaults to no"
echo " -v on for verbose, off for silent, defaults to on"
echo " -f filename"
exit 1
}
if [ -z $# ]; then
    echo "No parameters found"
    usage
fi
#default values
PRINT=Ø
VERBOSE=on
while getopts :pv:f: opt
do
    case "$opt" in
      p)  PRINT=1
     ;;
```

```
      v)  VERBOSE="$OPTARG"
          case "$VERBOSE" in
           on|On|ON) ;;
          off|Off|OFF) ;;
           *) echo "Verbose option must be 'on' or 'off'"
             usage;;
          esac
;;
      f)  FILES="$OPTARG"
            for FILENAME in $FILES
             do
# file exist & size > Ø bytes
           if ! [ -s "$FILENAME" ]; then
            echo "File '$FILENAME' does not exist"
            usage
           fi
          done
;;
      \?)
          usage ;;
     *) usage ;;
    esac
done
shift 'expr $OPTIND - 1'
for FILENAME in $FILES
do
echo "processing ...\nPRINT is $PRINT\nVERBOSE is $VERBOSE\nFILENAME is
$FILENAME  "
echo "========="
# your actual processing statements go here.
done
```

Running this with a non-existent file produces:

```
$ sampelopt -f "file3.txt" -v off -p
File 'file3.txt' does not exist
usage: sampelopt [ -p to print] [-v on or off]  -f filename
 -p print file defaults to no
 -v on for verbose, off for silent, defaults to on
 -f filename
```

I have not covered all the options **getopts** has to offer, but hopefully enough to show you that you can handle command line options in your scripts with renewed confidence and vigour.

David Tansley
Global Production Support (IBM-p series) (UK)

Automated mechanism for changing passwords – part 2

This month we conclude the article on changing passwords.

PASSCH.SH

```
#!/bin/ksh
###############################################
# This script changes the password for a given user by changing the
# lastupdate & password fields in /etc/security/passwd file.
###############################################
# This function replaces encrypted_passowrd & lastupdate in /etc/
security/passwd
function replace_pass {
lsuser $usr > /dev/null 2>&1
if [[ $? = 0 ]] ;then
chk='cat $pwd_file | grep ${usr}: | wc -l'
if [ chk -eq 1 ] ; then
   if [[ ${#epass} = 13 ]] ; then
      if [[ $lupdate = +([0-9]) ]] ; then
         let lno='nl -ba $pwd_file | grep ${usr}: | awk '{print $1}''
         twolines='sed -n "$((${lno}+1))','$((${lno}+2))'p' $pwd_file |
grep -E "password|lastupdate" | wc -l'
         if [ $twolines -eq 2 ] ; then
            # taking out encrypted password field
            encrypted_pw='sed -n "$((${lno}+1))','$((${lno}+2))'p'
$pwd_file | grep password | awk -F= '{print $2}''
            # taking out the lastupdate field
            lastupdate='sed -n "$((${lno}+1))','$((${lno}+2))'p'
$pwd_file | grep lastupdate | awk -F= '{print $2}''
            # getting out the spaces from the strings
            epass_old='print $encrypted_pw | tr -d '\0''
            lupdate_old='print $lastupdate | tr -d '\0''
            # Backup the passwd file
            cp -p $pwd_file $bck_pwd_file > /dev/null 2>&1
            if [[ $? = 0 ]] ; then
               # replacing new encrypted password
               sed s#$epass_old#$epass# $pwd_file > $new_pwd_file
               # replacing new lastupdate field
               sed s#$lupdate_old#$lupdate# $new_pwd_file > $pwd_file
               rm $new_pwd_file
               if [[ $usr != root ]] ; then
                  pwdadm -c $usr
               fi
```

```
            else
               print "$retstr205"
               exit 205 # backing up the passwd file was not successful
            fi
         else
            print "$retstr204"
            exit 204 # sed could not get the correct lines in /etc/
security/passwd
         fi
      else
         print "$retstr203"
         exit 203 # lastupdate is not in correct format
      fi
   else
      print "$retstr202"
      exit 202 # encrypted passowrd is not in correct format
   fi
else
   print "$retstr201"
   exit 201 # cannot find user in /etc/security/passwd file or string
douple found
fi
else
   print "$retstr207"
   exit 207 # user does not exist
fi
}
################################################
# variables
scr_name=$0
usage="$scr_name <user> <encrypted_passowrd> <lastupdate>"
pwd_file=/etc/security/passwd
bck_pwd_file=/etc/security/passwd.old
new_pwd_file=/etc/security/passwd.new
################################################
# error codes
retstr201="cannot find $usr in /etc/security/passwd file or string
douple found! PASSWORD NOT CHANGED !!!"
retstr202="encrypted password is not in correct format! PASSWORD NOT
CHANGED !!!"
retstr203="lastupdate is not in correct format! PASSWORD NOT CHANGED
!!!"
retstr204="sed could not get the correct lines in /etc/security/passwd!
PASSWORD NOT CHANGED !!!"
retstr205="backing up the passwd file was not successful! PASSWORD NOT
CHANGED !!!"
retstr206="Wrong number of parameters are given! PASSWORD NOT CHANGED
!!!"
retstr207="User $1 does not exist! PASSWORD NOT CHANGED !!!"
# MAIN #########################################
```

```
if [[ $# = 3 ]] ; then
   usr='print $1 | tr -d '\Ø''
   epass='print $2 | tr -d '\Ø''
   lupdate='print $3 | tr -d '\Ø''
   replace_pass
else
   print "$retstr2Ø6"
   exit 2Ø6  # Wrong number of parameters are given
fi
```

STAGE 3/UPDATER.SH

The script works on the main system if the password for the given user is successfully changed on the target system. In this stage, the files are updated on the main system. For non-root password changes, these files are not updated, but the new password for the user of the system is logged in a history file. As I explained above, every group in the company has a directory and it is identified using the file *.which_group_needs_which_pw*. The new password files are then updated in the appropriate group directory.

We have five groups:

- flgrp – first level support
- slgrp – second level support
- sapgrp – SAP
- dbagrp – DBA
- webgrp – Web.

For all groups under */pwl/* there exists a directory on the main system:

- flgrp – first_level
- slgrp – second_level
- sapgrp – sap
- dbagrp – dba
- webgrp – web.

All directories belong to root, and every group has its own group permissions. The permissions for all directories are **rwxr-s---**. Additionally, ACL is activated so that the group **system** is authorized to have access to the directories.

So for example under the directory */pwl/web*, which only the Web group can access, there is a list of systems that they need to know the root passwords for:

```
tcell002.turkcel.com.tr
tcell002.turkcel.com.tr.old
tcell005.turkcel.com.tr
tcell005.turkcel.com.tr.old
tcell007.turkcel.com.tr
tcell007.turkcel.com.tr.old
.........................................
.........................................
```

The format of each file is as follows:

```
<hostname with domain> <ip> <user> <8-char-passwd>
```

eg:

```
tcell005.turkcell.com.tr 10.55.203.30 root H?%,EjSV
```

For:

```
/usr/local/sbin/updater.sh
```

The script has the following steps:

1 If the password for the given user is successfully changed on the target system then this script is executed by the host.

2 It unzips **goner.gz** and takes the information.

3 If the given user is root then all groups are identified as needing the password. It moves the password files to *.old* and creates the new file with the new information under the directories.

4 It deletes the transfer file **goner**.

5 It makes a tar file of the */pwl* directory with its subdirectories, where all the password files are sent/ftp to another server in another location/city in case of disaster.

6 If ftp is successful, it deletes the compressed tar file on the main system.

7 Int performs an incremental back-up with Tivoli System Management.

8 A history log (date, server, user, and password) of manual and automatic executions of the password mechanism is kept in a file:

```
.............................................
 Mon Mar 24 13:03:44 CET 2003    tcell088    root    P41qb07G
 Mon Mar 24 14:03:09 CET 2003    tcell002    root    inz8lErk
 Mon Mar 24 15:03:46 CET 2003    tcell044    root    Cpzz,,J6
 Mon Mar 24 16:03:59 CET 2003    tcell055    root    Es,/UeN)
 Mon Mar 24 17:03:19 CET 2003    tcell002    root    fu:?fa4H
 Mon Mar 24 17:22:22 CET 2003    tcell042    tradnan    12345678
 Mon Mar 24 18:03:36 CET 2003    tcell077    root    adnanakb
 Mon Mar 24 18:16:43 CET 2003    tcell078    root    xdnanakb
 Mon Mar 24 19:03:06 CET 2003    tcell065    root    CyIcB7MD
 Mon Mar 24 19:39:16 CET 2003    tcell012    trahmet    adnanakb
 Mon Mar 24 20:01:50 CET 2003    tcell033    root    akbasadn
 Mon Mar 24 21:03:14 CET 2003    tcell099    root    W9y:cTp(
 Tue Mar 24 22:03:17 CET 2003    tcell002    root    yVhcW5t1
 Tue Mar 24 23:03:23 CET 2003    tcell055    root    H885;K2b
 Tue Mar 25 00:03:26 CET 2003    tcell007    root    OqV;HeV/
 Tue Mar 25 01:03:56 CET 2003    tcell045    root    tqMeM3/U
 Tue Mar 25 02:03:56 CET 2003    tcell018    root    Au:ai_?_
 Tue Mar 25 03:03:29 CET 2003    tcell002    root    HPc0z)E)
 Tue Mar 25 04:03:05 CET 2003    tcell033    root    uPohFB?9

.............................................
.............................................
```

9 The */pwl/.work/running* file is removed so it can be created at the next time of execution.

Each step has an exit code in case of error.

UPDATER.SH

```
#!/bin/ksh
###############################################
# This script updates the files belonging to the groups in the company.
# This way every group member can see the new passwords they need to
# know.
###############################################
```

```
# varibales
goner_zipfile=/pwl/.work/goner.gz
goner_file=/pwl/.work/goner
grps="first_level second_level sap dba web"
fname=/pwl/.which_group_needs_which_pw
grps_dir=/pwl
pwdlog=${grps_dir}/.work/history_passwd
local_dir=/pwl/.work
run_file=${local_dir}/running
tar_file=${local_dir}/pwds.tar
tar_send_to=spdcws
tar_target_dir=/home/ftpuser
logfile=${local_dir}/ftp_${tar_send_to}.log
tar_user=ftpuser
tar_pwd=xxxxxx
###############################################
# error codes
retstr151="ERROR: $goner_zipfile could not be uncompressed! FILES NOT
UPDATED !!!"
retstr152="ERROR: $goner_zipfile does not exist or empty! FILES NOT
UPDATED !!!"
retstr153="ERROR: Could not tar the directory structure! TAR FILE NOT
SENT !!!"
retstr154="ERROR: Tar file could not be compressed! TAR FILE NOT SENT
!!!"
retstr155="ERROR: Cannot change to ${local_dir}! TAR FILE NOT SENT !!!"
retstr156="ERROR: Cannot ping ${tar_send_to}! TAR FILE NOT SENT !!!"
retstr157="ERROR: Cannot connect to $tar_send_to ! TAR FILE NOT SENT
!!!"
retstr158="ERROR: Login failed, check user and password! TAR FILE NOT
SENT !!!"
retstr159="ERROR: Unknown failure! TAR FILE NOT SENT !!!"
retstr160="ERROR: TSM backup not successful!!!"
###############################################
# unzipping the goner file
if [ -s $goner_zipfile ] ; then
   /usr/bin/gzip -d $goner_zipfile > /dev/null 2>&1
   if [ "$?" != "0" ] ; then
      print "$retstr151"
      exit 151 # file not uncompressed
   fi
else
   print "$retstr152"
   exit 152 # file does not exist or empty
fi
# getting the info from goner file
user='cat $goner_file | awk '{print $2}''
chosen_host='cat $goner_file | awk '{print $1}''
de_passwd='cat $goner_file | awk '{print $3}''
if [[ $user = "root" ]] ; then
```

```
#lastupdate='cat $goner_file | awk '{print $4}''
#en_passwd='cat $goner_file | awk '{print $5}''
# updating files for the groups
random_line='grep "$chosen_host" $fname'
chosen_ip='echo $random_line | awk '{print $2}''
chosen_domain='echo $random_line | awk '{print $1}''
chosen_host_with_domain="$chosen_host.${chosen_domain#*.}"
for grp_name in $grps
do
  print $random_line | awk '{print $4}' | grep -q $grp_name
  if [ $? -eq Ø ]
  then
     if [ -s ${grps_dir}/${grp_name}/${chosen_host_with_domain} ] ; then
        mv ${grps_dir}/${grp_name}/${chosen_host_with_domain}
${grps_dir}/${grp_name}/${chosen_host_with_domain}.old
        print $chosen_host_with_domain $chosen_ip $user $de_passwd >
${grps_dir}/${grp_name}/${chosen_host_with_domain}
     else
        print $chosen_host_with_domain $chosen_ip $user $de_passwd >
${grps_dir}/${grp_name}/${chosen_host_with_domain}
     fi
  fi
done
# Making a tar file from the directory structure
tar -cf $tar_file $grps_dir > /dev/null 2>&1
if [ $? -eq Ø ] ; then
   chmod 4ØØ $tar_file > /dev/null 2>&1
   compress -f $tar_file > /dev/null 2>&1
   if [[ $? != Ø ]] ; then
     print "$retstr154"
     exit 154 # tar file not compressed
   fi
else
   print "$retstr153"
   exit 153 # could not tar the directory structure
fi
# FTP the compressed tar file to another host for backup
# Check whether we are in the right directory in local server
cd ${local_dir} > $logfile
if [[ $(pwd) != ${local_dir} ]]
then
   print "$retstr155"
   exit 155 # cannot change to the local directory
fi
# Check whether target is pingable
ping -c 2 ${tar_send_to} 1 > /dev/null 2>&1
if [[ $? != Ø ]]
then
   print "$retstr156"
   exit 156 # cannot ping Host
```

```
fi
# Start FTP Job
ftp -v -n ${tar_send_to} << ! >> $logfile
user $tar_user $tar_pwd
prompt
bin
cd ${tar_target_dir}
put pwds.tar.Z
bye
!
# Check whether successfully connected to Host
grep -q "Connected to spd999eØ" $logfile
if [ $? != Ø ]
then
   print "$retstr157"
   exit 157 # Cannot connect to Host
fi
# Checking whether the user could log in.
grep -q "Login failed" $logfile
if [ $? -eq Ø ]
then
   print "$retstr158"
   exit 158 # Login failed check user/passwd
fi
# Checking whether ftp is successful
grep -q "bytes sent in" $logfile
if [[ $? != Ø ]]
then
   print "$retstr159"
   exit 159 # Unknown failure
else
   rm ${tar_file}.Z > /dev/null 2>&1
fi
# Backing up with TSM
print >> $logfile
print « ---------------------------------------- « >> $logfile
print >> $logfile
dsmc inc $grps_dir  -subdir=yes >> $logfile 2>&1
grep -q "Successful incremental backup of" $logfile
if [ $? != Ø ]
then
   print "$retstr16Ø"
   exit 16Ø # TSM backup not successful
fi
fi
# history logging
print " `date`    $chosen_host    $user    $de_passwd" >> $pwdlog
chmod 4ØØ $pwdlog
# deleting the goner (transfer) file
rm $goner_file > /dev/null 2>&1
```

```
# deleting the indicator file
rm $run_file  > /dev/null 2>&1
```

Adnan Akbas
System Administrator
TURKCELL (Germany)

Complex sed operations

Under normal circumstances, **sed** reads a line to be processed and then performs any operations relating to the line, either from subcommands in a source file or directly from the command line, one after the other. When all of these have been applied, it then reads in the next line, performs operations on this line, and so on. In this article we will discuss how we can break up the normal flow control so that our lines can be processed in non-standard ways.

USING THE EDIT BUFFERS

Two work spaces, or temporary buffers, are used by **sed** for holding the line being modified:

- The **pattern** space
- The **hold** space.

The pattern space

When **sed** processes an input file, it performs the following operations:

- It first reads each input line into its pattern space edit buffer.
- All the **sed** subcommands which select that line are then applied in sequence.
- The modified line is then written to standard output.
- The pattern space is cleared and the process is repeated for each succeeding line in the input file.

The advantage of reading one line at a time into the pattern space is that large files can be processed without having to read the whole file into memory, and thus use up valuable memory resources. Although this is not quite so important in modern systems, which tend to have much greater amounts of memory, it does speed up processing since the system does not have to wait for the whole file to be loaded into memory.

When there are multiple operations to be performed on each line, at some stage the current pattern space version of the line may have changed out of all recognition from its original state. If subsequent operations are dependent on the original version of the line, then these operations may not be executed. This problem can usually be rectified by changing the order in which the various operations in your script, source file or command line are performed.

As an example, let us assume that we want to change the following line in our **stars** file:

```
Brooklyn Becks     102
```

so that **Becks** becomes **Spicer**, and the contestant number is changed to **305**.

If we use the command:

```
sed -e s/Becks/Spicer/ -e /Becks/s/102/305/ stars
```

then this will not work since the first operation changes **Becks** to **Spicer** in the pattern space, and the second operation now looks in the pattern space for the string **Becks**, which no longer exists, and so the second operation is never performed. To get round this we must reverse the operations. This example shows that you must give some thought to the order in which you want to execute your **sed** subcommands.

The hold space

The hold space is analogous to the edit buffers used in **vi** to contain deleted or yanked text from the main edit buffer. You can use the hold space to keep a copy of something that was in the

pattern space, and then achieve cut and paste type operations similar to yank and put operations in interactive editors. The contents of both the pattern and hold spaces can be copied to each other.

There are a number of operators that allow you to move data between the pattern and hold spaces:

- **h** or **H**

  Copy (**h**) or append (**H**) the contents of the pattern space to the **h**old space. The copy overwrites, whereas the append puts a newline after the contents of the hold space and then adds the contents of the pattern space.

- **g** or **G**

  Copy (**g**) or append (**G**) the contents of the hold space to the pattern space. The copy again overwrites, and the append puts a newline after the pattern space contents and then adds the contents of the hold space. Think of the pattern space as the primary space from which all operations are performed and then this becomes a **g**et operation.

- **x**

  Exchange the contents of the pattern and hold spaces.

EXAMPLES USING PATTERN AND HOLD SPACES

Manipulation of the pattern and hold spaces can take some time to get used to, and you will probably find that you need several attempts with your subcommands before you get things right. The following examples may help you towards an easier understanding of how these two edit buffers work within **sed**.

Example 1

Since the hold space is initially empty, an easy way to make a file double-spaced is to use the **G** operation to append the hold space after each line in the pattern space:

```
sed G stars
```

As each line is read into the pattern space, a newline is appended, followed by a blank line from the hold space; this combination produces the double spacing. If, instead, you use the **g** operation, which overwrites the pattern space with the hold space, then the output is a series of blank lines, as many as there are in the **stars** file.

Example 2

We can exchange the first and second line in the **stars** file by creating the following source file:

```
$ vi source

1{
     h
     d
}
2G
```

and then running:

```
sed -f source stars
```

This first line is read into the pattern space and then copied to the hold space (**h**). After this has been completed it is then deleted from the pattern space with the **d** operator. Since the pattern space is now empty, nothing is written to standard output.

The second line is read into the pattern space, which now consists of this single line, and the only operation to be performed on line 2 is to append the contents of the hold space (**2G**). When this has been done, the contents of the pattern space are written to standard output.

The remaining lines of the file have no operations performed against them and they are displayed without alteration as each is read into the pattern space.

Example 3

Suppose we wanted to move to the end of the file all lines in the **stars** file whose contestant numbers start with the number **1**. This can be achieved with:

```
sed -e "/1../H" -e "/1../d" -e '$G' stars
```

As each line is read into the pattern space, **sed** tests whether there is a match for the number **1** followed by 2 characters. If there is, then the line is appended (**H**) to the hold space, after which it is deleted from the pattern space (**d**), and since the pattern space is now empty, nothing is written to standard output.

If a line does not match our test criteria after being read into the pattern space, then there is no operation to be performed on it and it is written directly to standard output. Eventually, the hold space will contain all lines with the pattern **1..**.

When the last line is read into the pattern space, **sed** determines that there is an operation to be performed on it (**$G**). The contents of the hold space are thus appended to this last line, and the pattern space is then written to standard output.

Notice that there is a blank line in the output after the current last line. This is because we are appending to the hold space and an append operation puts a newline after the current contents of the hold space before adding the pattern space. Since initially the hold space contains nothing, this results in nothing followed by a newline, which gives us a blank line at the start of the hold space. You can remove the blank line from the output with:

```
sed -e "/1../H" -e "/1../d" -e '$G' stars | sed "/^$/"d
```

Notice also that in order to get the lines copied after the last line in the file, we enclose the **$G** in single quotes. This is a case where double quotes cannot be used since the shell would then try to substitute a value for the shell variable named **G**, which is not what is intended.

MULTIPLE PATTERN SPACE LINES

We have already created multiple lines in the pattern space by using the **G** operator to append the contents of the hold space, but let us now consider how we can read two or more lines from our input straight into the pattern space and then perform our edit operations directly on these. Before we can do this we must first understand the next operator.

Reading the next line

There are two next operators, **n** and **N**, which are used in **sed** source files and which operate in slightly different ways. The first of these, **n**, will empty the contents of the pattern space to standard output (unless it has been suppressed of course, in which case it just empties the pattern space), and then read in the next line of input without returning control to the top of the source file script.

As a simple example, assume that in error we have duplicated all lines containing the name **Smith** in our **stars** file, and that the duplicates are situated beneath the originals (if not, you can always sort the file). We can delete these extra lines by creating a **sed** source file containing the following lines:

```
/Smith/{
     n
     /Smith/d
}
```

When you run this script, as each line is read into the pattern space, **sed** checks to see whether there is a match for the string **Smith**. If there is, then it starts to run all operations contained in the braces, but immediately finds an **n** operator, which tells it to ignore the remainder of the operations in the braces and instead read the next line into the pattern space. If that line also contains the string **Smith**, then it is deleted.

When using source scripts similar to the one above, any operations prior to the **n** will obviously not be applied to the new input line. In our example above we were only interested in deleting the next input line so there was no requirement to perform further operations on it, but in other circumstances you may wish to carry out some other edit operation, rather than just delete the line. Similarly, operations performed prior to the **n** will not be applied to the new line.

The next operator we have used so far merely replaces the current line in the pattern space with the next input line, but the **N** operator allows us to read additional lines into the pattern space so that we can perform edit operations on multiple lines

separated by newline characters, which we must specifically match with **\n** if we want to perform some substitute command across the lines.

You should be aware that when there are multiple lines in the pattern space, **^** matches the start of the first line, and **$** matches the end of the last line.

To see how the **N** operator works, let us assume that in our **stars** file we want to exchange the contestant numbers on any pair of lines throughout the whole file (let's not worry about why!), so that the first the two lines, for example, would now look like:

```
Vicky Spicer       201
Serena Smith       102
```

and each succeeding pair of lines will also have the numbers swapped. One of the ways we can do this is with a source file with the complicated-looking substitute command (usually easier to construct than to read):

```
N
s/\([0-9]*\)\n\(.*[ ^I]\)\([0-9]*\)/\3\
\2\1//
```

So how does this work? When **sed** reads a line into the pattern space, it immediately appends the next line (**N**) to the pattern space. In the first part of the substitution command, which is enclosed between the first pair of forward slashes, we have three groups of expressions enclosed by escaped parentheses. We have met constructions like this before when we used the **expr** command, and each one instructs **sed** to remember the characters extracted by the enclosed expression.

The pattern matching works on the multi-line pattern space and says "match zero or more digits (remember this as string **1**), followed by a newline (**\n**), followed by zero or more characters in turn followed by a single tab or space (remember this as string **2**), followed by zero or more digits (remember this as string **3**)".

If you consider the first two lines of the **stars** file, then the remembered strings are:

```
string 1  "101"
```

```
string 2 "Serena Smith      "
string 3 "2Ø1"
```

The start of our pattern space also contains:

```
"Vicky Spicer      "
```

but we do not need to remember these characters since we do not intend to do anything other than print them.

The second part of the substitution, which as you will see is split over two lines, says “substitute the pattern space text extracted by the regular expressions from the first part of the command, with the third remembered string, followed by a newline, followed by the second remembered string, followed by the first remembered string”. This exchanges the contestant numbers, sends the pattern space to standard output, and then continues with the next pair of lines.

The remembered strings are shown as **\1**, **\2**, and **\3**, which are shorthand notations for the first, second, and third strings respectively. These are distinct from the other character used for remembered patterns, **&**, which remembers all the characters matched between the first two forward slashes (***/.../***), whereas the escaped numbers only remember those matched from the expressions enclosed in escaped parentheses (**\(...\)**).

To insert a newline into the second part of the substitution command we cannot, unfortunately, use **\n** again and instead we must resort to an escaped newline as shown. This ensures that a newline is inserted between the two lines in the pattern space.

There is a major bug in our **sed** script since it assumes that there is an even number of lines within our text file. If there is an odd number, then the final line will have no further line to read into the pattern space and this results in the last line not being printed. To get round this we ensure that the first line of the source script is **$p**. With an even number of lines we will never satisfy this match since our lines are read in pairs, but if there is an odd number then this will print the last line and then exit.

It can be quite frustrating using these complex pattern matching

expressions with **sed**. For example, we could not use either of the following as the first part of the substitute expression to swap our contestant numbers:

```
s/\([0-9]*\)\n\(.*\)\(.*\)/
```

or:

```
s/\([0-9]*\)\n\(.*\)\([0-9]*\)/
```

since **sed** can match neither the **\(.*\)** nor the **\([0-9]\)** unless it is preceded by a character that it can definitely identify, such as a specific space or tab, not the catch all **.***. Extensive experimentation may be required to achieve your goal.

Input and output loops

Using a combination of the **N**, **D**, and **P** operators, it is possible to construct input/output loops to allow operations to be performed on multi-line pattern spaces. You are already familiar with the first of these operators, but not the other two.

The **D** operator is similar to its lower-case cousin, but instead of deleting the whole of the pattern space, it deletes everything up to the first newline. It does not cause a new line of input to be read, but instead it returns control to the top of the script so that the subcommands can then be applied to the text that still remains in the pattern space.

The **P** operator usually follows the **N** operator and prints out everything up to the first newline in the pattern space. The lower case **p**, by comparison, prints out the whole of the pattern space when the script has finished all its operations on the current line, and then reads in the next line. **P** allows you to output part of the multi-line pattern space, and then continue operations on what is left in the pattern space. It does not return control to the top of the script.

To understand how these work, suppose we have a file containing the following text:

```
The company is called Atlas
```

```
Widgets. For many years, Atlas
Widgets was the market leader in Atlas
and Widgets production.
```

What we want to do is change every occurrence of **Atlas Widgets**, to **Atlas Universal Widgets**. It is relatively simple to construct a **sed** script, which changes every occurrence where both **Atlas** and **Widgets** are on the same line, but in our text file above they are on different lines and so we must give a great deal more thought as to how we are going to achieve this.

Consider the following **sed** script, and what happens when it is executed:

```
/Atlas$/{
     N
     /\nWidgets/{
          s// Universal&/
          P
          D
     }
}
```

When this script is executed against our text file, it reads the first line into the pattern space, finds a match for **Atlas** at the end of the pattern space, and then reads in the second line (**N**); remember that **$** matches the end of the last line in a multi-line pattern space, not the end of any line, although in this case it is not relevant since initially we only have a single line in the pattern space when we test for the match. The pattern space will now contain:

```
The company is called Atlas\nWidgets. For many years, Atlas
```

The next line of the **sed** script now tries to match a newline in the pattern space followed by **Widgets**. If this is successful, then the substitute command replaces the remembered pattern, **\nWidgets** (represented by the double empty slashes, ***//***) with **<space>Universal\nWidgets**; the **&** also signifies the remembered pattern, but **sed** does not allow us to use it in the first part of the substitution. The pattern space now contains:

```
The company is called Atlas Universal\nWidgets. For many years, Atlas
```

The **P** operator now prints out ‘The company is called Atlas Universal’, which is all the text up to the **\n**, and the **D** operator deletes this text and then passes control to the top of the script to continue processing with what still remains in the pattern space. **sed** again finds a match for **Atlas$** in the pattern space, reads in the third line and repeats the above process, eventually printing out ‘Widgets. For many years, Atlas Universal’.

When control is again passed to the top of the script after the **D** operator has completed, **sed** finds a match for **Atlas** at the end of the third line, which is all that remains in the pattern space. It then reads the fourth line into the pattern space, but no match is found for **\nWidgets** since **Widgets** is not at the start of its line and is preceded by a space. The remaining script subcommands are not executed and **sed** prints out the third line without alteration. The script finishes executing, and exits for this line.

Finally, the fourth line is read into the pattern space and the script is then executed for this line, which results in the last line being printed without alteration since **sed** can find no text matches. Our final output looks like:

```
The company is called Atlas Universal
Widgets. For many years, Atlas Universal
Widgets was the market leader in Atlas
and Widgets production.
```

LABELS AND BRANCHING

Let us now look at two commands that allow you to jump to specified locations in a **sed** script. The branch command, **b**, allows you to unconditionally transfer execution control to labels located elsewhere in the script. The second, the test command, **t**, is a conditional transfer dependent on a substitute command changing the current line; it also branches to a label.

A label appears on a line of its own and can consist of up to seven characters preceded by a colon. No spaces are permitted between the colon and the label name, and you must be careful not to put a space or tab at the end of the line since they will be considered to be part of the label name.

When you jump to a label using either the branch or test commands, you must put a space between the command and the label name. You must also be careful not to put a space at the end of the line.

The branch command

The **branch** subcommand has the format:

```
[address]b[label]
```

If the label does not exist, then control is transferred to the end of the script and the contents of the pattern space will be displayed. To understand how branching works, create the following text file:

```
This example shows \
how to join lines \
ending in a backslash.
No backslash, then\
no join.
```

What we would like to do is create a **sed** script to join each line in our text file that ends in a backslash with the line that follows it. Consider the following script:

```
s/\\[ ^I]*$/\\/
: join
/\\$/{
     s/[a-z],.!?]\\/& /
     s/\\ / \\/
     N
     s/\\\n//
     b join
}
```

The first substitute command, **s/\\[^I]*$/\\/**, selects a line that ends with a **** followed by zero or more spaces or tabs, and then changes this pattern to a single backslash; the **** combination is used to escape a single ****. This allows us to get rid of spaces and tabs that appear after the backslash since these would not normally be visible when we look at the text file, and they are not required.

The **/\\$/** pattern now selects all lines that end in a backslash, and the commands in the braces that follow are all executed against lines now matching this pattern.

The first and second substitute commands in the braces are together used to match lines that end in a backslash not preceded by a space, such as the penultimate line in our text file, and then insert the space. The first, **s/[a-z],.!?]\\/& /**, adds a space after the backslash, and the second, **s/\\ / \\/**, exchanges the space and backslash so that we have a standard pattern against which we can now use the third substitution command.

The **N** subcommand now appends the next line, embedding a new-line character directly after the ****, and the **s/\\\n//** command then deletes the **** and embedded new-line (**\n**). The first **** in the pattern again escapes the second ****, but it is not necessary to escape the third **** since it, together with the **n**, indicates the new-line character to the shell.

Finally, the **b join** command branches back to the label, **: join**, to continue checking for a **** at the end of the newly joined line, which is still in the pattern space. Without this branch, **sed** would write the joined line directly to output before checking for a further ****, and, if necessary, reading in the next line. The branch will thus allow all the lines with backslashes to be joined together.

Our **sed** source file currently has a bug in it. Remember that the **N** subcommand causes **sed** to stop immediately if there are no more lines of input. It does not copy the pattern space to standard output before stopping, which means that, if the last line of the input ends with a ****, it is not copied to the output. We can get round this by changing our source file to:

```
s/\\[ ^I]*$/\\/
: join
/\\$/{
     s/[a-z],.!?]\\/& /
     s/\\ / \\/
     $b end
     N
     s/\\\n//
     b join
```

```
}
:end
s/\\$//
```

We have now introduced a second branch command before the **N** operator to check whether we are on the last line of the file. If so, then we branch to the label, **:end**, so that the **N** operator is not executed. The script then continues and runs the final substitution, which deletes the backslash at the end of our last line. This construction ensures that our final line is printed.

Note that the above substitution commands will have to be modified if any of your lines contain backslashes that are not at the end of the line.

The test command

The **test** subcommand is identical to the branch subcommand with its syntax:

```
[address]t[label]
```

It will only branch to a label if there has been a successful substitution on the currently addressed line; if there is no label, it will branch to the end of the file.

To see how this subcommand works, assume we have a text file containing the following lines, where the second directory name in the path has been erroneously interchanged with the final file name:

```
/home/file1/user1
/home/file2/bin/user1
/home/file3/user2
/home/file4/lib/user2
```

What we want to do is swap these back so that we have the correct pathname. We can do this by using the following **sed** source file:

```
s-/\(.*\)/\(.*\)/\(.*\)/\(.*\)-/\1/\4/\3/\2-
t
s-/\(.*\)/\(.*\)/\(.*\)-/\1/\3/\2-
```

We have used the minus sign with the substitution command to

surround the pattern match and replacement parts of the command in order to avoid escaping the forward slashes in the pathname and so make the commands easier to read (!).

Both of the substitutions are used to extract the characters following each slash (**\(.*\)** and remember them. From earlier in the article you will remember that the escaped digits in the second part of the command represent the remembered strings from the first part in the order they have been processed from left to right. We must use two substitution commands in our script since we have to exchange different strings for three and two directory pathnames; for three directories we must exchange the second and fourth strings, and for two directories it must be the second and third strings.

Our first substitution is performed on any pathname matching three directories, and the second is performed on pathnames with two directories. When we have made our substitution on a three directory pathname, we then want to continue with the next line of the file. The **t** command, which follows the first line, checks that there has been a successful substitution for a three directory pathname and then jumps to the end of our source file, thus allowing processing to start with the next line. If the substitution was not successful, the input line contains only two directories in the pathname and so the second substitution command is run against this line.

When you make substitutions on lines containing multiple identical matches you must always perform them on the line which gives you the greatest number of matches first, before jumping to the end of your source file with the **t** command. If you do it in the reverse order, ie least first, then your first substitution will always be successful and you will then jump to the end of the source file and thus miss out the appropriate substitution command for the line.

Tonto Kowalski
Guru (UAE)

Implementing I/O multipathing using HP Auto Path XP product

I/O multipathing is an essential feature of modern operating systems. AIX 5L Version 5.2 provides a new feature called Multipath I/O (MPIO), which allows for a single device (disk, lun) to have multiple paths through different adapters. These paths must reside within a single machine or logical partition of a machine. Multiple machines connected to the same device are considered as clustering and not as MPIO.

Unfortunately, earlier versions of AIX that are still in use do not include this feature and MPIO is presently supported only for SCSI devices.

So what solution can be used to implement multipathing for configurations that use modern fibre channel-based storage? Unfortunately no single answer can be found. For instance EMC Storage arrays have as a prerequisite usage of PowerPath EMC software, IBM ESS Shark users must use SDD (Subsystem Device Driver), while IBM FASstTXXX users use a totally different and specific implementation included in the devices.fcp.disk.array.rte fileset.

This article will discuss the HP Auto Path XP V04-01 product, which provides multipathing support for computers running AIX and connected to HP Disk Array XP128, XP256, and XP1024. These storage devices are OEM versions of the following storage devices produced by Hitachi:

- Hitachi Lightning 9900 V Series (9900V)
- Hitachi Lightning 9900
- Hitachi Freedom Storage 7700E
- Hitachi Thunder 9200
- Hitachi Freedom Storage 5800
- Hitachi Freedom Storage 5700E.

Not surprisingly, the software itself has been produced by Hitachi as well.

AUTO PATH FEATURES

The major features of Auto Path are:

- Load balancing – when multiple paths connect a host and storage, Auto Path distributes the load across multiple paths to prevent a heavily-loaded path from affecting processing speed.
- Path failover – when multiple paths connect a host and storage, Auto Path switches to another path if there is a failure in the current path being used. This allows processing to continue without interruption because of path failure.
- Fallback – when a path recovers from an error, Auto Path places the path online. This enables the maximum number of paths to be online, allowing Auto Path to ensure that the number of paths to which it can distribute loads is the maximum possible in that situation.
- Path health checking – Auto Path can automatically check the status of the paths at user-specified time intervals. This eliminates the need for repeatedly performing manual checks of the path status.

INSTALLATION PREREQUISITES

System prerequisites for using the Auto Path XP for AIX are:

- Operating system:
  - AIX 4.3.3 with maintenance level of 09 or later.
  - AIX 5.1 with maintenance level of 02 or later and patch IY37437.
  - AIX 5.2 and patch IY39860.
- Host-based adapters:

  - IBM Fibre Channel Adapter FC6227:
    - Adapter firmware level should be 3.22A0 or later.
    - AIX 4.3.3 driver (df1000f7) level should be 4.3.3.75 or later.
    - AIX 5.1 driver (df1000f7) level should be 5.1.0.10 or later (AIX 5.1 Fiber Driver patch is required; it is called devices.pci.df1000f7.com.5.1.0.36.bff and is found in IY37437).
    - AIX 5.2 driver (df1000f7) level should be 5.2.0.0 or later.
  - IBM Fibre Channel Adapter FC6228:
    - Adapter firmware level should be 3.82A1 or later.
    - AIX 4.3.3 driver (df1000f9) level should be 4.3.3.75 or later.
    - AIX 5.1 driver (df1000f9) level should be 5.1.0.10 or later (AIX 5.1 Fiber Driver patch is required; it is called devices.pci.df1000f7.com.5.1.0.36.bff and is found in IY37437 patch).
    - AIX 5.2 driver (df1000f9) level should be 5.2.0.0 or later.
- HP XP disk arrays models and minimum firmware levels:
  - XP256 52-42-52-00/00
  - XP512 01-10-00-00/13
  - XP1024 21-01-24-00/00.
- VisualAge C++ Runtime 5.0.0.0 or later. When using VisualAge C++ Runtime 6.0.0.0 in AIX 5.1 be sure to install the IY33524 patch.

INSTALLATION AND CONFIGURATION PROCESS

The following description of the installation process assumes

that all storage and SAN hardware has been already configured and connected and that the storage has not yet been defined.

The Auto Path product is installed from CD-ROM. Hitachi Network Objectplaza Trace Library is installed together with Auto Path.

Note: you must uninstall previous versions of Auto Path before upgrading to the current version.

To install Auto Path:

1 Log on as a user with root permission.

2 Insert the CD-ROM into the drive. You do not need to mount the medium.

3 Create the */var/DLM* directory, and then create a licence key file (dlm.lic_key) in the */var/DLM* directory.

The following example shows how to create the licence key 123456789ABCDEF:

```
# mkdir /var/DLM
# echo 123456789ABCDEF>/var/DLM/dlm_lice.key
```

Make sure to store the licence key in a safe place – at present it is generated by HP, based on the serial number of the XP array to which your server is connected. After successful completion of the installation process, the file containing the licence key will be deleted.

4 Install the software from the CD by executing the following command:

```
installp –aXgd /dev/cd0 all
```

5 Check that the software has been properly installed by executing the following command:

```
#  lslpp -L|grep -i auto
  AutoPath.rte            4.0.102.1    C     F    Auto Path
```

The installation status of the fileset should be **C** for commit.

6 Add directory */usr/DynamicLinkManager/bin* to the PATH environment variable:

```
export PATH=$PATH: /usr/DynamicLinkManager/bin
```

7 Set up the configuration file.

Note: if you would like all the disks to be recognized by the DLM driver, this step is unnecessary. Continue to the next one.

There are two ways to limit the DLM drivers managed by Auto Path:

- Define the disks (hdisk) that you would like the DLM driver to recognize in the */usr/DynamicLinkManager/drv/dlmfdrv.conf* file.
- Define the disks that you would *not* like the DLM driver to recognize in the */usr/DynamicLinkManager/drv/dlmfdrv.unconf* file.

A specification in the *dlmfdrv.unconf* file has priority over a specification in the *dlmfdrv.conf* file. Therefore, if the same disk is defined in both the *dlmfdrv.conf* and *dlmfdrv.unconf* files, the DLM driver will not recognize the defined disk.

When including the disk specification in the configuration file, remember to make sure that *all* disks representing the same logical unit of storage device are included in the same configuration file.

There are two methods to verify the lun number of a particular disk. The first one is to use the following command:

```
#    lsattr -El hdisk27|grep lun_id|awk '{print $2}'
0x270000000000000
```

The second one uses the HP **xpinfo** utility:

```
root@rsh7001:/home/root:  xpinfo -d|awk -F,
'{print $1," ",$3}'|sort –A –b +1

/dev/rhdisk29   01
/dev/rhdisk31   01
/dev/rhdisk30   02
/dev/rhdisk32   02
/dev/rhdisk1    03
/dev/rhdisk15   03
```

```
/dev/rhdisk16    Øb
/dev/rhdisk2     Øb
/dev/rhdisk17   1Ø
/dev/rhdisk3    1Ø
/dev/rhdisk33   11
/dev/rhdisk35   11
/dev/rhdisk18   12
/dev/rhdisk4    12
```

8 Execute the following command to start the DLM configuration manager:

```
# dlmcfgmgr
```

This command will load the DLM drivers and the DLM alert driver into the kernel and create the DLM, a device pointing to the luns available to the system and not masked out in the configuration files.

9 Execute the following command to verify that DLM drivers have been successfully loaded into the kernel:

```
# lsdev –C | grep dlm
```

You should expect output similar to the following:

```
#  lsdev -C|grep -i dlm
dlmadrv        Available              DLM Alert Driver
dlmfdrv1       Available              DLM Driver
dlmfdrv        Available              DLM Driver
dlmfdrv2       Available              DLM Driver
```

This listing shows that a single instance of a DLM Alert Driver (dlmadrv) as well as a single instance of non-lun-specific DLM driver (dlmfdrv) are available as well as two lun-specific (dlmfdrv1 and dlmfdrv2) drivers.

You should check the output from the previous command to verify that all lun-specific drivers have been defined according to the physical connectivity configured in your SAN.

Check the file */usr/DynamicLinkManager/log/dlmcfgmgr1.log* for error messages.

10 At this stage you are ready to use DLM devices for storage allocation. You can use the special SMIT menus located under **System Storage management/Logical Volume**

manager/DLM Volume Groups.

Or you can use the utilities located in */usr/DynamicLinkManager/bin*. These include: dlmmkvg, dlmsavevg, dlmrestvg, dlmexportvg, dlmrecreatevg, dlmsyncvg, dlmextendvg, dlmreducevg, dlmunmirrorvg, dlmimportvg, dlmreleasevg, dlmvaryoffvg, dlmreorgvg, dlmvaryonvg, dlmrmdev, dlmlsvg, dlmrestorevgfiles, dlmcfgmgr, dlmmgr, dlmchvg, and dlmmirrorvg.

Please note that although you are directing these commands to operate on a single instance of a lun-specific driver, the underlying DLM operations will use all available I/O paths.

COMMANDS AND OPERATIONS

The basic format of the Auto Path command is:

```
dlnkmgr operation-name [parameter [parameter-value]
```

There are six types of operation – **clear**, **help**, **offline**, **online**, **set**, and **view**.

Clear

The **clear** command is used to clear management information from the Auto Path internal buffers. The general format of the command is:

```
dnkmgr clear –pdst  [-s]
```

The single optional **-s** parameter enables execution of the command without requesting user confirmation.

Help

The **help** command is used to display the format of various Auto Path commands. The general format of the command is:

```
dnkmgr help [operation-name]
```

The single optional **-s** parameter enables execution of the command without requesting user confirmation.

Offline

The **offline** command is used to transfer an online path to an offline state. The general format of the command is:

```
dnkmgr offline [-path] –pathid AutoPATH_ID [-s]
```

A number of precautions must be taken before executing this command:

- The last path accessing a device cannot be placed offline.
- Placing too many paths offline may prevent path switching if an error occurs.

The **-path** parameter is optional.

The **-pathid** AutoPATH_ID specifies the unique AutoPATH_ID that Auto Path has assigned to the path during system start-up. Leading zeros in the AutoPATH_ID can be omitted.

The optional **-s** parameter enables execution of the command without requesting user confirmation.

Online

The **online** command is used to transfer an offline path to an online state. The general format of the command is:

```
dnkmgr online [-path] –pathid AutoPATH_ID [-s]
```

The **-path** parameter is optional.

The **-pathid** AutoPATH_ID specifies the unique AutoPATH_ID that Auto Path has assigned to the path during system start-up. Leading zeros in the AutoPATH_ID can be omitted. If no **-pathid** parameter is specified all offline paths are placed in an online state.

The optional **-s** parameter enables execution of the command without requesting user confirmation.

Set

The **set** command is used to alter different characteristics of Auto Path. The general format of the command is:

```
dlnkmgr set [-lb {on|off}]
[-ellv log-level]
[–elfs log-size]
[-systflv trace-level]
[-pchk {on [-intvl execution-interval] | off]
[-afb {on [-intvl execution-interval] | off]
[-s]
```

The **-lb** parameter enables or disables the load balancing function of Auto Path. The default value is on.

The **-ellv** parameter specifies the level of information collected into the error log. The possible values are:

0 Do not collect an error log.

1 Collect error information for the Error level or higher.

2 Collect error information for the Warning level or higher.

3 Collect error information for the Information level or higher (ie all levels).

The default value is 3.

The **-elfs** parameter specifies in KB the size of the error log file. The supported range is between 100 and 9900. The default is 1000.

If the size of the error log file reaches the specified value, Auto Path switches to a new file and outputs the data to it. All old files except for the preceding one are deleted.

The **-systflv** parameter specifies the level of trace output. The possible values are:

0 Do not output any trace.

1 Only output error information.

2 Output a summary of program operation.

3 Output details of program operation.

4 Output all information.

The default trace level is 0.

Larger trace level values cause a greater number of logs to be output. To accommodate a greater number of logs, data wrapping is applied by Auto Path to reduce the time required for deleting old logs.

The history and results of user-issued commands are output to the trace files regardless of the trace level.

The **-pchk** parameter enables or disables path health checking. The default value is off. The path health checking function checks paths that are online as well as offline (E). Path health checking does not check the status of paths in the offline(C) status because of the execution of the offline operation. Possible values are:

- on: Enabled – enables error path checking in the specified time interval. Automatic fallback places the status of the recovered path online.
- off: Disabled – disables error path automatic fallback (default).
- intvl execution-interval – specifies the path health checking interval with a value immediately following on. When you omit the interval, path health checking defaults to every 30 minutes.

  Specify the path health checking interval in minutes between 1 and 1440. The default value is 30 minutes.

  The interval setting does not take effect immediately. When this parameter is changed, the setting is stored, and the setting takes effect after the OS or the Auto Path manager is restarted. When the OS or the Auto Path manager is restarted, path health checking is executed according to the most-recently-specified execution interval.

  When you change the execution interval, the new setting takes effect only after path health checking is executed once, using the interval specified previously. When you want to change the execution interval immediately, execute the **dlnkmgr set -pchk off** command to disable path health checking, wait for more than five seconds, and then execute

the **dlnkmgr set -pchk on -intvl** execution-interval command. This will change the path health checking execution interval.

The **-afb** parameter enables or disables automatic path fallback. The default value is off.

Possible values are:

- on: Enabled – automatic failback of the failed paths is performed at a specified interval. The recovered path is returned online.
- off: Disabled – automatic failback is not performed (default).
- intvl execution-interval – specifies the path health checking interval with a value immediately following on. When you omit the interval, path health checking defaults to every 1 minute.

When automatic fallback is newly turned on, or the Auto Path manager is activated, the first automatic fallback is performed after the duration of the specified interval.

Settings on automatic fallback remain valid after the system is turned off and restarted. The interval value is maintained even after the automatic fallback is turned off. The stored value is used when automatic fallback is set on again and no value is set for the interval at that time.

Only the paths containing an error are subject to the automatic fallback operation.

The paths having an error when the DLM manager is activated are also included in the automatic fallback operation.

With the addition of the auto fallback, only normal paths are subject to health check.

The optional **-s** parameter enables execution of the command without requesting user confirmation.

View

The **view** command enables the display of Auto Path settings

and I/O path status.

The general format of the command to display system information is:

```
dlnkmgr view -sys [-sfuc | -msrv | -adrv | -pdrv]
```

This form of the command enables the display of the following specific information about Auto Path path settings:

- **-sfunc** flag displays information about the Auto Path function settings.
- **-msrv** flag displays information about the Auto Path manager.
- **-adrv** flag displays information about the Auto Path alert driver.
- **-pdrv** flag displays information about the Auto Path driver.

If you do not specify the value for this parameter, the command displays all program information.

Auto Path function settings displayed by the **view** command are:

- Auto Path version – the Auto Path version number consisting of four to six characters.
- Load balance – set the load balancing function: **on** for enabled, and **off** for disabled.
- Support cluster – set the cluster support function and the type of cluster server (see note 1 below). Cluster support function (see note 2 below) settings are on for enabled and off for disabled. The type of cluster server is HACMP.
- Elog level – error logging level:

  0 Do not collect an error log.

  1 Collect error information for the Error level or higher.

  2 Collect error information for the Warning level or higher.

  3 Collect error information for the Information level or

higher (ie all levels).

- Elog file size (KB) – size of the error log file in KB.
- Trace level – trace output level:

  0 Do not output any trace.

  1 Only output error information.

  2 Output a summary of program operation.

  3 Output details of program operation.

  4 Output all information.

- Path health checking – this function's settings are **on** for enabled and **off** for disabled. When path health checking is **on**, the execution interval (in minutes) is displayed in parentheses.

Note 1: Auto Path automatically recognizes the cluster configuration and sets the cluster support function. The setting cannot be changed manually.

Note 2: when you use Auto Path for AIX systems in a cluster configuration, off is displayed in the Support Cluster field; however, the cluster support function operates normally.

The general format of the command to display I/O path information is:

```
dlnkmgr view –path [-hdev host-device-name] [-t]
```

The **-path** parameter specifies the path for which you want to display information. You can also specify a host device that the path accesses. If you do not specify the subsequent parameter, the command displays information for all paths.

The **-t** disables the display of titles for each information item.

Host Device path information displayed by the **view** command includes:

- Paths – sum of the number of displayed paths, indicated by up to five decimal numbers.

- OnlinePaths – number of available paths in the displayed paths, indicated by up to five decimal numbers.
- PathStatus – status of the displayed paths. **Online**: all paths are available. **Reduced**: some of the paths are available. **Offline**: no paths are available.
- IO-Counts – total I/O count for the displayed paths, indicated by up to ten decimal numbers.
- IO-Errors – total I/O error count for the displayed paths, indicated by up to ten decimal numbers.
- PathID – AutoPATH_ID, indicated by up to six decimal numbers.
- PathName – path name, indicated by a string of up to 19 one-byte characters. A path name consists of the port number for the host bus adapter, bus number, target ID, and host LU number, each of which is separated by a period (full stop).

- DskName – storage subsystem name, indicated by a string of up to 38 one-byte characters. A storage subsystem name consists of the vendor ID, product ID, and serial number, each of which is separated by a period (full stop).
- iLU – LU number of the storage subsystem, indicated by a string of up to four one-byte characters.
- ChaPort – port number of the channel adapter, indicated by a string of two one-byte characters.
- Status *– status of the path. **Online**, online; **offline(C)**, offline by a command; **offline(E)**, offline because of an error.
- Type – attribute of the path. **Own**: owner path, **non**: non-owner path.
- IO-Counts – total I/O count for the path, indicated by up to ten decimal numbers.
- IO-Errors – total I/O error count for the path, indicated by up to ten decimal numbers.

- DNum – device number, indicated by up to three decimal numbers.
- HDevName – host device name.

In AIX systems, the hdisk is displayed in this field. When the file system is not mounted, a hyphen (-) is displayed.

Note: a path that has the same PathName as a path whose status is offline(E) may be in an error status even though its status is online.

REFERENCES

1 *HP StorageWorks Auto Path XP for AIX Installation Guide*, Fourth Edition, April 2003 (located in file *installation_guide.pdf* on Installation CD).

2 *Hitachi Dynamic Link Manager (AUTO PATH) User's Guide for IBM AIX Systems*, Revision MK-92DLM111-1, October 2002 (located at *\Auto Pathhelp\en\index.htm* on the Installation CD).

3 *HP StorageWorks Disk Array XP128: Owner's Guide*.

4 *HP StorageWorks Disk Array XP1024: Owner's Guide*.

5 *Hitachi Lightning 9900 V Series User and Reference Guide*, MK-92RD100.

6 *Hitachi Lightning 9900 V Series IBM AIX Configuration Guide*, MP-92RD119.

7 *Hitachi Lightning 9900 User and Reference Guide*, MK-90RD008.

8 *Hitachi Lightning 9900 IBM AIX Configuration Guide*, MK-90RD014.

9 *Hitachi Thunder 9200 User and Reference Guide*, MK-90DF504.

10 *Hitachi Thunder 9200 IBM AIX Host Installation Guide*, MK-91DF544.

11 *Hitachi Freedom Storage 5000 Series Software Configuration User's Guide*, BO-98DF376.

Alex Polyak
System Engineer
APS (Israel)

You don't have to lose your subscription to *AIX Update* when you move to another location – let us know your new address, and the name of your successor at your current address, and we will send *AIX Update* to both of you, for the duration of your subscription. There is no charge for the additional copies.

November 2000 – October 2003 index

Items below are references to articles that have appeared in *AIX Update* since Issue 61, November 2000. References show the issue number followed by the page number(s). Back-issues of *AIX Update* are available back to issue 49 (November 1999). See page 2 for details.

AIX news

Group 1 Software has announced the release of its new Universal Coder, the first product built on Group 1's new LESLIE architecture. LESLIE enables organizations to integrate Group 1's data quality services seamlessly across the enterprise. The Universal Coder incorporates Group 1's address validation, correction, and standardization technologies for over 220 countries and dependencies worldwide.

LESLIE is a platform-independent enterprise data quality architecture. Group 1's new Universal Coder provides flexibility by letting organizations implement global address data quality in their own enterprises, or even access this functionality via hosted services using the same API. The Universal Coder can process global address data in a single pass - mixed data from any number of countries can be submitted and processed simultaneously. The Universal Coder supports Java, COM, C, and C++ interfaces and can be easily integrated into a .NET environment.

The Universal Coder is currently available on AIX, Sun Solaris, HP UX, Linux, and Windows NT.

For further information contact:
Group 1 Software, 4200 Parliament Place, Suite 600, Lanham, MD 20706-1844, USA.
Tel: (888) 413 6763.
URL: http://www.g1.com/News/disppr.asp?PR_ID=235.

* * *

IBM has announced Lotus Domino Collaboration Express and Lotus Domino Utility Server Express. The products offer the performance, security, and dependability of IBM Lotus Notes and Domino 6 to small and mid-size businesses, IBM says. The messaging and collaboration server provides e-mail, group scheduling, discussion forums, team workplaces, and Domino's custom application capabilities. Customers have multiple client access choices, including Lotus Notes or Domino Web Access (iNotes), Domino Access for Microsoft Outlook, or Domino Web Mail.

Both packages work on a variety of operating systems, including AIX, Intel-based Linux, Microsoft's Windows, and Sun Microsystems' Solaris.

IBM likes to think of the products as cheap, easy-to-use, and simple to install and manage.

For further information contact your local IBM representative.
URL: http://www.lotus.com/products/product4.nsf/wdocs/dominoexpress.

* * *