90

AIX update

April 2003

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.

Monitoring filesystems

One of the main problems a system administrator faces is the filesystem on servers becoming full – particularly where a large-scale Internet portal is running on the servers (as in my company). The log files or core files can make your filesystems full very quickly, which then causes a lot of problems. In this article, I want to share my filesystem monitoring script (fs.sh) for servers, and two other scripts (logger.sh and rotate_alog.sh) showing how to deal with important log files generated by WebSphere and HTTP servers.

FS.SH

The fs.sh script runs on one machine and issues rsh commands (for execution on remote servers) to the machines to be monitored. When the script detects a filesystem usage that is greater than the given threshold value on any server, it warns the administrator(s) with an Outlook e-mail, which lists the server name, filesystem name, per cent usage, and how much space is left. This is written in the subject field of the Outlook e-mail. This entry is then written in the 'sent' file to make sure that this e-mail is sent only once to the administrator(s). As soon as the script realises that the problematic filesystem has a value less than the threshold value, it sends another mail that shows the situation (server name, filesystem name, percent usage, and how much space is left) and the entry is deleted from the 'sent' file.

We use a user called 'perf' on our servers for these situations. A common user is needed for clients to trust the server where the script runs.

HOME SERVER

The following files are needed on the server (let's say rsc100e0) where the script runs.

/home/perf/servers contains the hostname list of the servers to be monitored (hostnames must be in the */etc/hosts* file):

```
$ cat servers
rsc148e0
rsc149e0
spc329e1
spc361e1
spc497e1
spc745e1
spc581es
spc913es
..............
.............. (new servers can be added easily)
```

/home/perf/emails contains the e-mail address list for Admin(s):

```
$ cat emails
xxxx.yyyyy@turkcell.com
aaa.www@turkcell.com
..............
.............. (new e-mail addresses can be added)
```

The */home/perf/historylog* is created by the script, which adds all warnings that are sent, with the date, for later usage. When there is a problem with a server, it's always possible to check this history file to see whether there was a problem with the filesystem usage at a specific date.

$ cat historylog:

```
..............
..............
Wed Jul 24 08:50:34 CEST 2002 -  rsc148e0 -  /usr/HTTPServer  - file
system is  %98  full. Free space  107 MB
Wed Jul 24 09:00:37 CEST 2002 -  rsc148e0 -  /usr/HTTPServer  - file
system has dropped to  %60. Free space  25 MB
Wed Jul 24 09:40:26 CEST 2002 -  rsc149e0 -  /Weblogs  - file system has
dropped to  %94. Free space  65 MB
Wed Jul 24 11:20:23 CEST 2002 -  rsc149e0 -  /Weblogs  - file system is
%96  full. Free space  49 MB
Wed Jul 24 11:50:27 CEST 2002 -  rsc142e0 -  /Weblogs  - file system has
dropped to  %48. Free space  540 MB
Wed Jul 24 15:40:04 CEST 2002 -  spc329e0 -  /usr/HTTPServer  - file
system is  %97  full. Free space  184 MB
Wed Jul 24 19:30:12 CEST 2002 -  spc745e1 -  /usr  - file system is  %97
full. Free space  26 MB
Wed Jul 24 19:35:15 CEST 2002 -  spc745e1 -  /usr  - file system has
dropped to  %93. Free space  54 MB
```

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

The script is triggered from crontab every 10 minutes:

```
ØØ,1Ø,2Ø,3Ø,4Ø,5Ø * * * * /home/perf/fs.sh > /dev/null 2>&1
```

REMOTE CLIENTS

The server where the monitoring script runs, rsc100e0, must be in */etc/hosts* of the clients. The following file must be in the home directory of 'perf', the in .rhosts file:

```
$ cat /home/perf/.rhosts
rsc1ØØeØ   perf
```

Otherwise, rsh won't work!

FS.SH

```
#!/bin/ksh
# Adnan Akbas, Ø5.Ø2.2ØØ2
# This script monitors file system usage for given remote servers
# and warns the administrator(s) by sending mail when the filesystem
# usage of a server gets more than the given threshold value.
# Getting the list of remote servers in a array
set -A server 'cat ~perf/servers'
# Threshold value for the filesystems
threshold=95
emails=~perf/emails
i=Ø
# Filesystem Alert in Servers
while (( $i < ${#server[*]} )) ; do
  # Executes "df -k" command in each remote server and writes
  # the output in a file.
  rsh ${server[i]} "df -k" |grep -v AFS | grep -v Filesystem | grep -v /
proc | awk '{print $3, $4, $7}' | awk -F% '{print $1, $2}'  >> ~perf/
fs.out
  # Reads every line in fs.out
  while read line
  do
    # Getting the needed data. per: percentage, fs: filesystem name,
    # avail: available space left.
    per='echo $line | awk '{print $2}''
    fs='echo $line | awk '{print $3}''
    avail='echo $line | awk '{print $1}''
    # Checks if %used of the filesystem. If it is greater
    # than Threshold checks the sent file.
```

```
    if [ $per -gt  $threshold ] ; then
      grep "${server[i]} $fs " ~perf/sent > /dev/null 2>&1
      # If it is in the sent file already, then it doesn't
      # send an email again.
      # If not sends the email warning to the Admin(s) with
      # the % of fullness and space left.
      # And the entry is added to the sent file.
      if [[ $? != 0 ]] ; then
        echo "${server[i]} $fs " >> ~perf/sent
        mesaj='echo "${server[i]} -  $fs  - file system is  %$per  full.
Free space  $(( $avail / 1024 )) MB"'
        # Sends the message to every email address in emails file.
        cat $emails |
        while read no
        do
     # The info is in the subject of the outlook mail. (for easy seeing)
         echo "Automatic mail by Akbas" | /usr/bin/mailx -s "$mesaj" -r
FS@info $no > /dev/null 2>&1
        done
        # Logging every sent message to historylog file.
        echo " `date` -  $mesaj " >> ~perf/historylog
      fi
    else
      # Here the %used of the filesystem is below Threshold.
      grep "${server[i]} $fs " ~perf/sent > /dev/null 2>&1
      # Checks if the entry is in sent file already.
      # If yes sends an email with the news that the
      # Filesystem has dropped to %value
      if [[ $? = 0 ]] ; then
        fmesaj='echo "${server[i]} -  $fs  - file system has dropped to
%$per. Free space  $(( $avail / 1024 )) MB"'
        # Sends the message to every email address in emails file.
        cat $emails |
        while read fno ; do
     # The info is in the subject of the outlook mail. (for easy seeing)
          echo "Automatic mail by Akbas" | /usr/bin/mailx -s "$fmesaj" -
r FS@info $fno > /dev/null 2>&1
        done
        # The entry is deleted from the sent file.
        grep -v "${server[i]} $fs " ~perf/sent > ~perf/senttemp
        mv ~perf/senttemp ~perf/sent
       # Logging every sent message to historylog file for later use.
       echo " `date` -  $fmesaj " >> ~perf/historylog
       fi
    fi
done < ~perf/fs.out
rm ~perf/fs.out
i=$i+1
done
```

LOGGER.SH

When you start the WebSphere application on a server, it creates different kinds of log file and performs excessive writes to them. The problem we have faced at our site is that these files (with a time stamp) are still used by WebSphere, sometimes until the next start-up.

It is very important to know whether this file is used by a process before you move/compress the file. If the file is in use and the remove/move command is executed, the file descriptor will be lost and the application (Websphere, HTTP server, etc) will be running without logging. So, this script senses the log files that are not used by processes any more and moves and compresses them automatically, which saves a lot of space for us.

The script is triggered from crontab every hour:

```
ØØ * * * * /usr/local/sbin/logger.sh > /dev/null 2>&1
```

Example log names:

```
servlet.<time stamp>
ecmsg_<hostname>_<time stamp>
……..
```

Here is the code:

```
#!/bin/ksh
# Adnan Akbas , 25.Ø2.2ØØ2
# The script checks whether given log files is in access or not.
# if there is no access, compresses the files and moves to the save
# directory.
# info: all servlet_* and ecmsg_* logs are with time stamp.
logdir=/Weblogs/wcs
servlet_dir=/Weblogs/wcs/save/servlet
ecmsg_dir=/Weblogs/wcs/save/ecmsg
cd $logdir
ls servlet.* |
while read fname ; do
  # Checks if any Process (PID) accesses the log file.
  xxx='fuser ${logdir}/${fname} | awk '{print $1}''
  # If you don't get a PID, then logfile is free for packing.
  if [[ $xxx != +([Ø-9]) ]] ; then
  mv ${logdir}/${fname} ${servlet_dir}/${fname}
  compress ${servlet_dir}/${fname} > /dev/null 2>&1
  fi
```

```
done
ls ecmsg_spcØ49e1_* |
while read fname ; do
  # Checks if any Proccess (PID) accesses the log file.
  xxx='fuser ${logdir}/${fname} | awk '{print $1}''
  # If you don't get a PID, then logfile is free for packing.
  if [[ $xxx != +([Ø-9]) ]] ; then
    mv ${logdir}/${fname} ${ecmsg_dir}/${fname}
    compress ${ecmsg_dir}/${fname} > /dev/null 2>&1
  fi
done
# Remove logs older then 15 days.
find ${ecmsg_dir}/ -name "ecmsg_*.Z" -mtime +15 -exec rm {} \; > /dev/
null 2>&1
find ${servlet_dir}/ -name "servlet.*.Z" -mtime +15 -exec rm {} \; > /
dev/null 2>&1
```

ROTATE_ALOG.SH

There are also logs, like *access_log*, that are always in use by HTTP server and must be handled with special care. Moving/ renaming and then erasing doesn't work because of the file descriptor problem that I mentioned above. The best way to do this is to copy the file with a time stamp and then:

```
# cat /dev/null > access_log
```

This makes access_log size 0 without losing the file descriptor. So, HTTP server keeps on writing logs to this file.

The script is triggered from crontab once everyday:

```
59 23 * * * /usr/local/sbin/rotate_alog.sh > /dev/null 2>&1
```

log name:

```
access_log
```

Here is the code:

```
#!/bin/ksh
# Adnan Akbas ,  Ø7.Ø3.2ØØ2
# The script rotates access_log file which is in access.
tstamp='date +%d%m%y'
logdir=/usr/HTTPServer/logs
# Copies the file with a time stamp
cp ${logdir}/access_log ${logdir}/access_log_${tstamp}
# if successful remove access_log and create an empty access_log file
if [ $? -eq Ø ]
```

```
then
   cat /dev/null > ${logdir}/access_log
fi
# Compress the file
compress ${logdir}/access_log_${tstamp} > /dev/null 2>&1
# Files older than a month will be deleted
find ${logdir}/ -name "access_log_*.Z" -mtime +3Ø -exec rm {} \; > /dev/
null 2>&1
```

Adnan Akbas
System Administrator
Turkcell (Germany)

Sending e-mail attachments from AIX

INTRODUCTION

A regular question posed in the newsgroups and forums is, how can I attach a file to an e-mail within AIX? This has become a popular past-time for many sysadmins who are trying to produce automated monitoring/alerting systems.

Where most people go wrong is in trying to use the Mail User Agent (MUA) software – mail, elm, pine, mh. It's easier just to place a constructed piece of mail directly into the SMTP stream using the Mail Transfer Agent (MTA) software **sendmail**.

ENTERING E-MAIL INTO THE SMTP STREAM

The ordinary **sendmail** command will read a text file using standard redirection and the **-t** flag will direct it to use the e-mail headers in that text file:

```
sendmail -t < email-file
```

CONSTRUCTING AN E-MAIL FILE

At its most simple, an e-mail is nothing more that a plain text file consisting of:

- Routing headers (sender, recipient, CCs, and BCCs)
- Subject field
- MIME boundary information
- Embedded text
- Attachments (as text)
- MIME boundaries.

TRADITIONAL HEADERS

```
From: Sender's Name <senders email address>
To: Recipient1 <recipients email address>
cc: cc-person <carbon-copy email address>
bcc: bcc-person <blind-carbon-copy email address>
Subject: Email Title
```

Note: the *From:* header will not override the actual userid used to execute the **sendmail** command, and any routing decisions made by the MTA will be based on that execution userid.

MIME HEADERS

Every set of MIME headers must be followed by a blank line.

Top level:

```
MIME-Version: 1.0
Content-Type: multipart/mixed;
      boundary="Unique-Character-String"
```

The *Unique-Character-String* must be unique throughout the e-mail (and not replicated in any of the attachments). When creating this string in a script it's a good idea to incorporate the current pid ($$) to make it unique between mailings as well.

For embedded text:

```
--Unique-Character-String
```

```
Content-Type: text/plain; charset=us-ascii
```

For attachments:

```
--Unique-Character-String
Content-Type: text/plain
Content-Disposition: attachment; filename="filename.txt"
```

Different *Content-Type:* headers can be specified for different attachment types. For example:

- Application/MSEXCEL for a file destined to be a spreadsheet.
- Application/X-PGP for an attachment that has been encrypted.

Final boundary:

```
--Unique-Character-String--
```

Be careful to include the final two dashes or this boundary will be misinterpreted.

EXAMPLE FILE

The following file is a constructed e-mail composed of some inline text, a text attachment, and a small spreadsheet:

```
From: Automated DLCMailer <sender@company1.com>
To:  A. N. Other <another@company2.com>
MIME-Version: 1.Ø
Subject:  Multipart MIME File
Content-Type: multipart/mixed;
      boundary="__=--DLC-BOUNDARY-15Ø4Ø2"

--__=--DLC-BOUNDARY-15Ø4Ø2
Content-Type: text/plain; charset=us-ascii

This is the plain text embedded message
It is to show how easy it is to construct MIME based emails


--__=--DLC-BOUNDARY-15Ø4Ø2
Content-Type: text/plain
Content-Disposition: attachment; filename="Chapter1.txt"

1.  The Earthquake
```

```
The train from 'Frisco was very late.  It should have arrived at
Hugson's Siding at midnight, but it was already five o'clock and the
gray dawn was breaking in the east when the little train slowly rumbled
up to the open shed that served for the station-house.  As it came to a
stop the conductor called out in a loud voice:

"Hugson's Siding!"

--__=--DLC-BOUNDARY-150402
Content-Type: application/MS-EXCEL
Content-Disposition: attachment; filename="pies.csv"

Person,Pies Eaten, Pies Left
John,4,5
David,10,3
George,1,1

--__=--DLC-BOUNDARY-150402--
```


Subsystems – not just for IBM

You have just put the finishing touches to that all-important program. It will run as a daemon and monitor your crucial application. You have put an entry into */etc/inittab* and you start it using **nohup**. This may seem like a good idea, but why not use the built-in functions and routines IBM gives you to stop/start/refresh your program?

CONCEPTS

AIX provides a series of commands, C routines, and control structures to manage programs that are designed to run continually (daemons). There is a hierarchical structure, starting with the System Resource Controller (*/usr/sbin/srcmstr*). Using commands such as **startsrc** and **stopsrc**, this daemon controls

the subsystems that are defined to the system. These subsystems can be grouped together as part of a suite of related programs to form subsystem groups and hence can be controlled as one. To complete the hierarchy, a subsystem may have several subservers.

For example, the **inetd** daemon, part of the tcpip subsystem group, has subservers such as ftpd and telnetd.

COMMANDS

I will not attempt to describe the use of the C routines in this article.

There is a collection of commands that are used to control and manipulate subsystems. Below is a very quick guide to them:

- startsrc – start a subsystem or group of subsystems.
- stopsrc – stop a subsystem or group of subsystems.
- lssrc – list the status of a subsystem or group of subsystems.
- refresh – stop and restart a subsystem or group of subsystems.
- traceson – enable tracing (debug information) for a subsystem or group of subsystems.
- tracesoff – disable tracing (debug information) for a subsystem or group of subsystems.
- mkssys – define a subsystem to the operating system.
- rmssys – delete a subsystem from the operating system.
- chssys – alter the definition of a defined subsystem.
- mkserver – define a subserver to the operating system.
- rmserver – delete a subserver from the operating system.
- chserver – alter the definition of a defined subserver.
- odmget – retrieve data stored in the Object Data Manager

database (ODM).

And, of course, there are the usual SMIT panels (from the initial menu choose **Processes & Subsystems** and then **Subsystems**).

I will not go into detail on each of the commands. The manuals cover the options for each command better than I can.

ALTERING AN ENTRY

The **lssrc** command produces output in colon-separated format, which is not the most user friendly. A quick script to format the output is shown below (it takes one argument, which is the subservice you are interested in):

```
#!/usr/bin/ksh
subsrv=$1
j=0;
lssrc -S | head -1 | awk -F":" '{ for ( i=1; i <= NF; i++ ) print $i }'
| while read cols[$j]
do
  let j=$j+1
done
j=0;
lssrc -S -s $subsrv | tail -1 | awk -F":" '{ for ( i=1; i <= NF; i++ )
print $i }' | while read entry[$j]
do
  echo ${cols[$j]} " : " ${entry[$j]}
  let j=$j+1
done
```

Alternatively, you can simply use the **odmget** command to get a formatted version for a subserver, eg:

```
odmget –q"subsysname=qdaemon" SRCsubsys
```

WORKING EXAMPLE

I have had the experience of the TSM client agent dying on some older systems because of ‘Program Memory Exhausted’ errors. This causes the **dsmc** schedule process to die off. Unless all the systems which run TSM schedules are monitored for the presence of the process, it will be the next morning before the problem is

picked up, when the output from the overnight schedules is checked.

There are several ways to sort this out; I will mention two here. First, you can alter the TSM entry in */etc/inittab* to **respawn** rather than **once** and have the init process pick up the problem. However, I favour having more control over the process, and, accordingly, I use an altered inittab entry and a subserver definition. This allows an administrator to stop and start the process as required and also has the system pick up a failed process.

The entry in */etc/inittab* looks like:

```
rctsm:2:wait:/usr/bin/startsrc –sdsmc
```

This can be inserted manually and have init pick it up via the **telinit q** command, or you could use the following command:

```
mkitab –i srcmstr "rctsm:2:wait:/usr/bin/startsrc –sdsmc"
```

The subsystem definition I have used is:

```
mkssys –s dsmc –G tsm –u Ø –p /usr/tivoli/tsm/client/ba/bin/dsmc –a
schedule –o /dev/console \  -e  /var/adm/ras/dsmerror.log –R –Q –S n1 –f
9 –E 38 –w 1Ø
```

Note: the **dsmc** command accepts communication via **signals**. Hence the options **-S -n 1 -f 9**, mean: for normal termination use **kill -1**; to force termination use **kill -9**. I use a priority of 38 (in a range of 0 (highest priority) to 39 (lowest)). The **-R** option tells the **srcmstr** process to **respawn** the **dsmc** process should it fail. Finally, on most machines, only one **dsmc** task will run, hence the **-Q** option. However, there are times when you might want to run two **dsmc** processes; for these machines the **-q** should be used.

CONCLUSION

There are several in-built 'utilities' within AIX, which can be used by system administrators to their own advantage. I have attempted here to highlight the possible advantages of using the in-built 'subsystem/subserver' model. I cannot foresee why an

administrator would want to alter any values for IBM supplied services; however, there is no reason why new ones cannot be added and the supplied mechanisms used to make life easier for administrators.

REFERENCES

AIX Commands Reference Manual.

AIX System Management Concepts: Operating Systems and Devices.

AIX General Programming Concepts: Writing and Debugging Programs.

Phil Pollard
Unix and TSM Administrator (UK)

Good practice in shell programming

This month we conclude the article on good practice in shell programming.

```
   fi
   #
   TOTAL_NO_ARGENTRY='expr ${TOTAL_NO_ARGENTRY} + 1'
done
#
if  [  $ARGC  -ge  ${MAN_NO_ARGENTRY}   ] &&
    [  $ARGC  -le  ${TOTAL_NO_ARGENTRY} ]
then
     :
else
    echo  "Invalid argument count"
    return $FALSE
fi
#
INDEX=1
# echo "Command Line=${ARGL}"
OPTION=""  #  each option and  value  pair
```



```
#
while  [ $INDEX  -le $ARGC  ]
do
   OPTION='echo "${ARGL}" | cut -d' ' -f${INDEX}'
   # extract argument
   ARG='echo "${OPTION}" |  cut -d= -f1'
   # extract value
   VAL='echo "${OPTION}" |  cut -d= -f2'
   #
   # check the argument against being null
   #
   if [ "${ARG}" =  ""  ]
   then
      echo "Must provide an argument using argument=value syntax "
      return $FALSE
   fi
   #
   # check the argument against the list
   #
   VALID_ARG=N
   for  ARGENTRY in ${ARGLIST}
   do
     if [ "${ARG}" =  "`echo "${ARGENTRY}" | cut -d':' -f1`" ]
     then
         VALID_ARG=Y
         break
     fi
  done
  #
  if  [  "${VALID_ARG}" != Y  ]
  then
      echo "Invalid argument ${ARG}"
      return $FALSE
  fi
  #
  # check argument specifier syntax
  #
  if  [ "${OPTION}" !=  "${ARG}=${VAL}"  ]
  then
       echo "Invalid argument specifier ${OPTION}"
       break
  fi
  #
  #  validate argument against expected list
  #
  for  ARGENTRY in ${ARG_PROCESSED}
  do
    if [ "${ARG}" =  "${ARGENTRY}"  ]
    then
```

```
      echo "Duplicate argument ${ARG}"
      return $FALSE
    fi
  done
  #
  # store this argument
  #
  ARG_PROCESSED="${ARG_PROCESSED} ${ARG}"
  #
  #   validate argument against valid values
  #
  if [  "${ARG}"  = "RERUN_FLAG"  ]  &&
     [ "${VAL}"  != "Y"  -a  "${VAL}" !=   "N" ]
  then
      echo "Invalid value for argument ${ARG}"
      return $FALSE
  fi
  #
  if [  "${ARG}"  = "PROCESSING_MONTH"  ]  &&
     [  "${VAL}" != "1"  -a "${VAL}" != "2" -a "${VAL}" != "3"  -a  \
        "${VAL}" != "4"  -a "${VAL}" != "5"  -a "${VAL}" != "6"  -a \
        "${VAL}"  != "7" -a "${VAL}" != "8"  -a "${VAL}" != "9"  -a \
        "${VAL}"  != "1Ø" -a "${VAL}"  != "11" -a "${VAL}" != "12"  ]
  then
      echo "Invalid value for argument ${ARG}"
      return $FALSE
  fi
  #
  INDEX='expr $INDEX + 1'

done
}
#########################################################################
# Name     : main
# Overview : Implements processing structure by invoking appropriate
#            functions in a defined manner.
#
# Notes    :
#########################################################################
main ()
{
ParseCommandLine
#
# other functions
#
}
#
#
FALSE=1
```

```
TRUE=Ø
ARGL="$@"
ARGC=$#
#
# invoke main
#
main
```

MAKING THE SCRIPT STRUCTURED

Making a script structured is all about writing one or more functions to implement the overall functionality of the script, which has one exit and one entry point.

Example 1: unstructured script

```
#
# assign  parameters
#
Var1=$1
Var2=$2
Var3=$3
#
#   process option
#
if  [  "${Var1}"   !=  "A"  -a  "${Var1}"  != "B"   -a  "${Var1}"  !=
"C"  ]
then
          echo "Var1 must be A, B or C"
          exit 1   #  exit point
fi
#
#
if  [  "${Var2}  =  "A"  ]
then
        #
        #    process this logic
        #
        #  check batch size
        #
        if  [  $Var2  -gt    1000 ]
       then
              echo  "Batch size must be less than 1000
              exit  1   # exit point
        fi
       #
       #  successful completion
```

```
       #
       exit Ø   #  exit point
       #
elsif  [  $Var2  =  "B"  ]
then
        #
        #   process this logic
        #
        #  check batch size
        #
        if  [  $Var2  -gt    10000 ]
       then
              echo  "Batch size must be less than 10000
              exit  1    #  exit point
        fi
       #
       #  successful completion
       #
       exit Ø   # exit  point
       #
elsif  [  "${Var2}"   =  "C"  ]
then
       #
       # process this logic
       #
       #  check batch size
       #
       if  [  $Var2  -gt    20000 ]
       then
              echo  "Batch size must be less than 20000
              exit  1   # exit point
        fi
       #
       #  successful completion
       #
       exit Ø   #  exit point
      #
fi
#
#
#    successful completion
#
exit Ø    #  exit point
```

Notes:

1 The script has multiple exit points.

2 The variable names are not self-explanatory.

Example 2: structured script

```
#########################################################################
#  Name     :  InitialiseVariables
#  Overview : The function initializes all variables
#   Notes   :
#########################################################################
InitialiseVariables
{
#
#  exit codes
#
SEC=Ø
FEC=1
#
#  return codes
#
TRUE=Ø
FALSE=1
#
#  expected arguments
#
RUN_OPTION=""
BATCH_SIZE=""
}
#########################################################################
#  Name     : ParseCommandLine
#  Overview : The  function parses the command line
#   Notes   :  1.  The function parses the string held in variable $ARGV
#
#              2.  The function also validates individual arguments
#########################################################################
ParsesCommandLine ( )
{
#  see example above for how to implement this function
}
#########################################################################
#  Name     : ProcessA
#  Overview : The  function  processes  option A   .
#   Notes   : 1.  The function parses the string held in variable $ARGV
#########################################################################
ProcessA ( )
{
return  $TRUE
}
#########################################################################
```

```
#  Name     : ProcessB
#  Overview : The function processes option B.
#   Notes   : 1.  The function parses the string held in variable $ARGV
#
#########################################################################
ProcessB  ( )
{
return  $TRUE
}
#########################################################################
#  Name     :  ProcessC
#  Overview :  The function processes option C .
#   Notes   :  1.  The function parses the string held in variable $ARGV
#########################################################################
ProcessC  ( )
{
return $TRUE
}
#########################################################################
#  Name     : ProcessExit
#  Overview : The function removes any temporary files and
#             makes a graceful exit.
#   Input   : Exit  Code
#   Notes   :
#########################################################################
ProcessExit  ( )
{
EXIT_CODE=$1
exit   ${EXIT_CODE}
}
#########################################################################
#  Name     :  main
#  Overview : The function implements processing control.
#   Notes   :
#########################################################################
main ( )
{
InitialiseVariables
#
# parse  command  line
#
if  !  ParseCommandLine
then
        ProcessExit  $FEC
fi
#
#
if   [   "${RUN_OPTION}"  =  "A"  ]
then
```

```
        if   !  ProcessA
        then
                ProcessExit   $FEC
        fi
#
elsif   [   "${RUN_OPTION}"  =  "B"  ]
then
        if   !  ProcessB
        then
                ProcessExit   $FEC
        fi
#
elsif   [   "${RUN_OPTION}"  =  "C"  ]
then
        if   !  ProcessC
        then
                ProcessExit   $FEC
        fi
fi
#
#
ProcessExit  $SEC
}
#
# package the command line
#
ARGV="$@"
ARGC="$#"
#
# invoke main
#
main
```

Notes:

1 The script has a single exit point via ProcessExit ( ).

2 The variable names are clear and meaningful.

3 The script is easy to read although it is longer.

4 Disabling or enabling a function (equivalent to commenting in and out in unstructured form) is easy. Just place a return statement at the top of the function definition (see below):

```
ProcessA ()
{
#
return  $TRUE
}
```

or:

```
ProcessA ()
{
#
return  $FALSE
}
```

5 Validation for command line parameters is done in ParseCommandLine ( ).

6 There are no exit points from any of the functions.

7 The function main ( ) implements the processing control structure.

8 The function main ( ) must be invoked in order to do anything useful with the script.

9 The $ARGV and $ARGC are assigned before the invocation of main ( ).

Adding debug or tracing features

You can easily add debug or tracing features to your shell script. The idea is that when a script is running with debug or tracing feature on, it will out put a lot of information, either on the screen or into a document, providing a picture of the inner working of the script. This sort of output is not desirable when running a script under normal circumstances because it will add to the execution time. A complex script should have this feature added, which can be turned on in order to understand the inner workings of it, perhaps to address a particular problem with the running of the script.

Now the question is, what information should be traced or debugged? This is entirely dependent on the script and its complexity, but nonetheless a general guideline would be as follows:

- Values of variables before and after assignment.
- Values of iteration.

- Indication of path taken in **if** or **case** statement.

Implementation

To implement the feature, add a command line specification as follows:

```
    myscript.sh        DEBUG=Y
```

Use ParseCommandLine ( ) function mentioned above to process the argument DEBUG.

FUNCTIONS

Functions are a powerful feature that aren't used often enough.

The syntax is:

```
Function-name  (  )
{
    command s
    return  $TRUE
}
```

Notes:

1 Functions can accept arguments like a script.

2 Functions can return a value:

```
add (  )
{
NUM1=$1
NUM2=$2
RESULT=""
RESULT='expr  $NUM1 + $NUM2'
return   $RESULT
}
#
#  main body of script
#
PARAM1=$1
PARAM1=$2
#
#  invoke add ( )
#
TOTAL='add    $PARAM1   PARAM2'
```

3 Functions can return a true or false status:

```
file_exists ( )
{
FILE_TO_CHECK=$1
if  [   -f    "${FILE-TO_CHECK}"  ]
then
        #  file  exists
        return  $TRUE
else
       return  $FALSE
fi
}
#
#  main body of script
#
TRUE=Ø
FALSE=1
#
#
FILE=$1
#
#  invoke file_exists  ( )
#
if    file_exists   "${FILE}2
then
         #  function has returned a TRUE status
         echo   "INFO: File,  ${FILE}  exists "   >&1
else
         echo  "ERROR: File,  ${FILE}  does not exist  "   > &2
fi
```

Auto-loading customized functions

Functions can be written and executed within a script, but what if you want to write functions and execute them as commands; how would you do it?

Function written and executed within a script

Script hello.sh:

```
hello ( )    #   function being defined
{
   echo  "Hello there "
}
#
```

```
#
hello    # function being invoked within the script
```

Running the script in a sub-shell:

```
Hello.sh
```

Running the script in the current shell:

```
.  hello.sh
```

The function hello ( ) has been auto-loaded into the memory by the current shell and therefore it can be invoked as a command as follows:

```
hello
```

Loading functions from a script

Write all the required functions in a script but do not invoke these within the script.

Modify .profile as follows:

```
#
#  auto load functions
#
.  function_library.sh
```

Note: you can write as many library scripts as you like, but in that case all these library scripts must be executed within the current shell as shown above.

Loading functions by placing these in pre-defined locations

Define the shell variable $FPATH to point to a location where scripts with function(s) definition will reside.

Define each function in a single script with the same location as the function and place it in the directory pointed to by $FPATH.

Add the following line to the scripts that need to invoke these functions:

```
    typeset –fu   function1_name
```

```
typeset -fu   function2_name
#
#  invoke these functions
#
function1_name
#
function2_name
```

Usage message

When a script terminates because of improper usage, a proper usage message must be displayed using standard syntax.

Example:

```
USAGE=Usage:myscript.sh  <run_option>  <batch_size>
The script takes mandatory parameter

USAGE=Usage:myscript.sh   <I> |  <B>
The script takes one of the two mandatory parameters

USAGE=Usage:myscript.sh   <run_option>  < batch_size>  <I>  |  <B>
```

Stdin, Stdout, Stderr

Standard input, output, and error are file descriptors 0, 1, and 2. Each has a particular role and should be used accordingly.

Stdin

By default place stdin in the terminal that the script is running on.

For example:

```
echo  "Enter   password:\c"
read  PASSWORD
```

The stdin has been set to point to the terminal and the user is expected to enter the password.

Stdout

All non-error messages should be sent to stdout as follows:

```
echo  "myscript.sh:INFO:Processing  first batch"  >&1
```

Stderr

All error messages should be sent to stderr as follows:

```
echo  "myscript.sh:ERROR:Wrong  argument  count"  >&2
```

Notes:

1 When a particular script is being executed, we can log different messages in different files as follows:

```
ERROR_MSG_FILE=/tmp/f1.err
INFO_MSG_FILE=/tmp/f1.info

myscript.sh   1> ${INFO_MSG_FILE}  2> ${ERROR_MSG_FILE}
```

2 We can ignore non-error messages and log just error messages, as follows:

```
myscript.sh  1>  /dev/null   2 >${ERROR_MSG_FILE}
```

3 We can swap standard input as follows:

```
 PASSWD_FILE=/etc/passwd
  #
  #  at this point stdin is set to point to file descriptor Ø
  #
 exec  3<&Ø   #  this saves the original stdin as FD  3
 #
 cat  ${PASSWD_FILE}  | while  read  LINE
 do
      #  at this point stdin is set to point to ${PASSWD_FILE}
      #  but we wish to accept a reply from the user from the
      #  terminal, how do we do it?
      #  ask the question
      echo   "  do  you   wish  to  delete  this  user  ?\c"
      #
      exec  4<&Ø    #  this saves the current stdin as FD 4
      exec   Ø<&3   #  this restores original stdin to read REPLY
      read  REPLY
      #
      Ø<&4        # this re-connects stdin to file ${PASSWORD_FILE}
done
```

Trapping interrupts

When running a shell script, if you press your interrupt key (like CTRL-c), the shell quits right away. This can be a problem if you

use temporary file(s) in the script because the sudden exit might leave the temporary files there. The trap command lets you tell the shell what to do before it exits.

Example myscript.sh:

```
InitialiseVariable  ()
{
SEC=Ø
FEC=1
TRUE=Ø
FALSE=1
}
ProcessExit  ()
{
EXIT_CODE=$1
 rm –f    ${TEMP_FILE}
exit ${EXIT_CODE}
}
HandleInterrupt  (  )
{
     echo  "Program interrupted; quitting early "
     ProcessExit   $FEC
}
main  ( )
{
   InitialiseVariable
#
#   do other processing
#
}
#
#invoke  main
#  but before invoking main, set the trap
#
trap   "HandleInterrupt"   1  2   15
#
main
```

Notes:

1 The first statement to be interpreted and executed is the trap command, which states that, if signal 1, 2, or 15 is detected, invoke function HandleInterrupt.

2 The function HandleInterrupt echoes a message and then invokes the function ProcessExit, which in turn removes all the temporary files.

3 The meanings of signals are as follows:

1 = SIGHUP – received when session is disconnected

2 = SIGINT – received when a session is interrupted (ctrl-c)

15 = SIGTERM – received when a kill -15 is received

18 = SIGSTP – received when interactive stop (ctrl-z) is used

9 = KILL – received when kill -9 is received.

4 The signal 9 cannot be caught. The signal kill -9 2345 cannot be caught by the process whose id is 1345 and this is why, when a process needs to be killed off, you should issue kill -15 <pid> first and then, if the process still persists, issue kill -9 <pid>. A good script will detect the signal 15 and take any necessary action before exiting.

Arif Zaman
ETL Developer (UK)

It's magic

A strangely named file exists in the */etc* directory. The file is called 'magic', a name that does not give you any indicationof its purpose. Is it really magic or just an imaginitive name for an ordinary file?

Unfortunately, it is just a strange name for a data file that enables the **file** command to do its work. The **file** command attempts to give a user some indication of the contents of a named file, thus avoiding the embarrassing situation of **cat**ting a binary file and having the whole office enjoy the spectacle of someone attempting to stop the **cat** command whilst sweating profusely. For example, the command:

```
/usr/bin/file /usr/bin/file
```

gives the result:

```
/usr/bin/file:   executable (RISC System/6000) or object module
```

However, the file command can get it wrong. Try running the following command on one of your systems:

```
/usr/bin/file /.sh_history
```

and see what is returned. A usual reply I have noticed is something along the lines of:

```
/.sh_history:   Ultrix-11 Stand-Alone or boot executable.
```

But what if you want to help the **file** command along a little? For example, by adding in an entry for Perl rather than having the answer 'shell script' returned?

The */etc/magic* file is an ASCII file that can be edited using your favourite editor. There are instructions in the header as to the layout of the file, which I will not repeat here. As an example, I have added the following line to my system in the '#!' section, which describes shell scripts:

```
>2   string   /usr/bin/perl      - Perl
```

However, the **file** command does not pick up any changes immediately. So, until you can issue a **bosboot** command and a reboot, the only way to get your changes included is to use the **-m** option with the name of the file you have altered. For example:

```
/usr/bin/file -m /etc/magic /usr/local/bin/perlfile.pl
```

Depending on what you entered into the */etc/magic* file for Perl, the answer will look something like:

```
/usr/local/bin/perlfile.pl:  shell script  -  Perl
```

where the *Perl* is taken from the entry made in */etc/magic*.

Take a look through the */etc/magic* file and discover how the **file** command works. The AIX Version 5 magic file does contain more than the one in Version 4. My additions to it work on both versions.

Phil Pollard
Unix and TSM Administrator (UK)

Arithmetic evaluation

Arithmetic evaluation in the Korn shell is a lot more sophisticated than in the Bourne shell, which relied exclusively on the **expr** command with its more complex syntax. **let** and **((...))** are now the preferred commands to use, but arithmetic manipulation in shell scripts would not be complete without a section on **expr**, since you may come across scripts written for the Bourne shell, or by people who know no other way.

ARITHMETIC AND RELATIONAL OPERATORS

The following arithmetic operators are permissible for integer arithmetic evaluation in the Korn shell:

- **+** for addition.

  The assignment operator, **+=**, is permitted with **let**.

- **-** for subtraction.

  The assignment operator, **-=**, is permitted with **let**.

- ***** for multiplication.

  Has to be escaped for **expr**

- **/** for integer division.

- **%** for integer remainder (modulus).

The C language rules on precedence are applicable, but should you have any doubts then expressions can be enclosed in parentheses to ensure that evaluation is performed in the correct order. As you will shortly see, depending on the command used, some operators may have to be escaped to avoid misinterpretation by the shell.

In addition to the arithmetic operators, there are the following relational operators:

- **=** or **==** for equality.
- **!=** for inequality.
- > for greater than.
- **<** for less than.
- **>=** for greater than or equal to. Not available with **expr>**.
- **<=** for less than or equal to. Not available with **expr>**.

When these operators are used for comparing integers, they produce the value 1 for true and 0 for false, which may be displayed, depending on the command used. Do not confuse these with command return values which relate to the successful execution (or otherwise) of the command, not the result of the comparison.

THE EXPR COMMAND

For completeness, we include string manipulation by **expr** in this article, which, despite its complex syntax, on occasion has no equal!

ARITHMETIC EVALUATION USING EXPR

The format for arithmetic evaluation is:

```
expr expression1 operator expression2
```

Parameter terms must be separated by blanks, and characters special to the shell must be preceded by a backslash. Examples are:

```
expr $x + $y
expr $x \* $y
expr $x - $y % 3
expr \( $x \* 7 \) / \( $y - 3 \)
```

The **expr** command returns the following exit values for arithmetic operators:

- If the expression is neither null nor 0, **expr 3 / 2** will display the value 1, and return an exit value 0.

- If the expression is null or 0, **expr 2 - 2** will display the value 0, and return an exit value 1.
- If the expression is invalid, **expr 2 * 2** will display a syntax error, and return an exit value 2.

The most common usage of **expr** is to modify a shell variable. For example:

```
count=$(expr $count + 1)
```

This adds 1 to the variable count. This type of statement is commonly used within the for, while, and until loops, where a variable is incremented and then tested to see if it meets a certain condition, whereupon the loop terminates.

The **expr** command returns the following exit values for relational operators:

- If the expression is true, **expr 3 != 2** will display the value 1, meaning true, and return an exit value 0.
- If the expression is false, **expr 4 = 5** will display the value 0, meaning false, and return an exit value 1.

COMPARING AND RETURNING STRINGS

In addition to being used for arithmetic evaluation, the **expr** command can also be used to return different string expressions. For example:

```
expr expression1 \| expression2
```

will return the string *expression1* if it is neither null nor 0; otherwise it returns *expression2*. Similarly:

```
expr expression1 \& expression2
```

will return the string *expression1* if neither *expression1* nor *expression2* is null or 0; otherwise it returns 0.

Examples are:

- **expr 2 + 3 \| 4 / 5** returns the value 5.
- **expr 2 / 3 \| 4 + 5** returns the value 9.

- **expr 2 / 3 \& 4 + 5** returns the value 0.
- **expr 2 + 3 \& 4 / 5** returns the value 0.
- **expr 2 + 3 \& 4 + 5** returns the value 5.

Very few shell scripts contain expressions such as those shown above since most programmers use them so infrequently that they either forget the syntax or cannot remember the values returned.

The most frequently used string operation is the **:** operator, which is used to compare two arguments. The first must be a string, and the second a regular expression. For example:

```
expr string1 : "expression2"
```

will normally return the number of characters matched, providing *expression2* is a pattern. The regular expressions or patterns can look very strange and can often be quite difficult to read, and for this reason alone they are rarely used in scripts, usually as a last resort when another command is not available or cannot be remembered.

Let us consider an example whereby we want to extract four-digit sequence numbers contained within filenames which have a format filename.nnnn.ext, where *nnnn* is the sequence number, but we want to exclude any files that start with a dot. If the file name is contained within the variable **$file**, then the syntax would look something like the following:

```
expr $file : "[^.].*\.\([ø-9]*\)\..*"
```

The **[^.]** pattern means ‘do not match the character contained within the square brackets’, which in this case is the dot, and since this is the very first part of our pattern then it will exclude files which start with a dot. In the pattern **.***, the dot is not contained in brackets and so becomes a wildcard, which will match any character; **.*** will thus match zero or more occurrences of any character. The pattern **\.** is an escaped dot because we want **expr** to recognize the character **.** itself, rather than just any character.

So far, **[^.].*\.** matches every character up to and including the first dot before the extension containing the sequence number, ie *filename.*. To extract the filename's sequence number we use the expression **\([0-9]*\)**, which matches zero or more occurrences of a digit; this tells **expr** to remember all the characters (**[0-9]***) it finds in the escaped parentheses **\(....\)**. Still with it?

Finally, we must exclude all characters after the digits, which we do with the pattern **\..*** following the escaped parentheses; this expressly matches a dot followed by any number of characters. We need the **\.** to identify the exact point that **expr** should stop remembering the characters to be extracted. If you had used just **.***, then the sequence number would not be extracted.

The goal in writing the regular expression is to identify all the characters in the string and to surround the part of the regular expression that recognizes the characters in which we are interested with the symbols **\(\)**. You will often find that much experimentation is required to extract the characters you want since **expr** often appears to use its own obscure rules!

The above form of **expr** is rarely used in shell scripts because of its complexity, and you can often (but not always) achieve the same result by using commands such as **awk**, which will be covered in a future article.

Another use of **expr** is to find the length of a shell variable:

```
length=$(expr $STR : ".*")
```

This sets the **length** to the value given by the **:** (colon) operator. Since the second string does not contain the **\(\)** characters, the pattern **".*"** matches any string from beginning to end and returns the length of **$STR** as the number of characters matched. Note that **".*"** must be in quotes to prevent the shell from treating the ***** as a pattern-matching character. The quotes themselves do not form part of the pattern.

If **$STR** is a null string, an error message is displayed since the

shell does not normally pass null strings to commands. This problem can be fixed by enclosing **$STR** in double quotes:

```
length=$(expr "$STR" : ".*")
```

Now, if **$STR** is null, **length** is set to zero.

You must also be careful with **expr** when you want to test, for example, whether a string has the value **=**. If you used the following expression:

```
expr "$STR" = "="
```

then after the shell processes the command, **expr** sees the expression:

```
= = =
```

and interprets this as three **=** operators in a row and displays a syntax error message. This happens whenever the value of a shell variable is the same as one of the expression operators, or the string (**$STR**) has a null value and you want to test that it is null.

When the shell variable has the same value as an operator, or if you are ever in doubt as to the way in which the shell may interpret the comparison, the problem can be avoided by using:

```
expr "x$STR" = "x="
```

Should the variable have a null value, for example,

```
expr "x$STR" = "x"
```

will allow the test for a null value to be performed. The above comments do not apply to comparisons using the **test** command and, because of this, and the more complicated syntax, **expr** is rarely used for string comparisons of this nature.

USING TYPESET

The **typeset** command provides the fastest method for arithmetic evaluation, although this will not usually be noticed unless your script is doing lots of calculations. Once a variable has been

defined as an integer, standard arithmetic expressions can be used. Floating point and decimal operations are not available using **typeset**, and variables are defined as integers by using the **-i** option, which by default uses base 10.

To set a variable as an integer, you use commands such as:

```
typeset -i intvar
```

or:

```
typeset -i intvar=20
```

which both set **intvar** as an integer, and the second form of the command initializes it to 20. Once a variable has been defined as an integer you can use standard arithmetic operations:

```
intvar=intvar+5
intvar=intvar/10
intvar=\(80-intvar\)/2
```

You must be careful not to leave any spaces in these expressions since the shell will then look for other commands, rather than perform variable arithmetic. A single **typeset** command can also be used to set and initialize any number of variables:

```
typeset -i int1=5 int2=23 int3 int4=0 . . . .
```

The optional **n** can be specified with **-i** to indicate the base of an integer. For example:

```
typeset -i2 bin=59
print $bin
2#111011
```

or for an octal number:

```
typeset -i8 oct=928
print $oct
8#1640
```

You should be aware that if you have defined a variable to be an integer, you cannot at some later stage give the variable a non-integer value. For example:

```
typeset -i int
int=aaa
```

will give an error message similar to:

```
aaa: The specified number is not valid for this command
```

Similarly, you cannot set the integer equal to a variable that is not itself an integer.

If a variable has been set to a base other than 10, and you want to reset it to base 10, enter:

```
typeset -i10 var
```

not:

```
typeset -i var
```

USING LET AND ((...))

The built-in shell command **let** can also be used to perform integer arithmetic. All calculations are performed as long integers, and, because **let** evaluates each argument separately, any expressions that contain spaces, tabs, or brackets to ensure that the operations are performed in the correct order must be quoted. Using **let** you do not have to predefine integers to be used in the expression and you can have a mixture of predefined integers and other variables. Also, you do not need to precede variable names with a **$**. For example:

```
$ x=3
$ typeset -i y=5
$ let z=x*y+2 or let "z = x * y + 2" or let "z=(x*y)+2"
$ print $z
17
```

There is an alternative to **let** that is used to define integer variables and assign values to them. The construction uses the **((...))** format and you can use spaces inside in much the same way as you can for quoted **let** expressions. The above commands could have been written:

```
$ x=3
$ typeset -i y=5
$ ((z = x * y + 2))
$ print $z
17
```

You can also combine expressions within the double parentheses:

```
$ ((z = y / x + x * y))
$ print $z
16
```

Note that **let** and **((...))** themselves do not display any output since they contain assignment operators. If you want to use arithmetic evaluation in a conditional statement within a script, you must use a construction without an assignment operator:

```
if [[ $((a / 3)) -eq 4 ]]
then . .
```

In much the same way that **[[..]]** evaluates string comparisons, etc, **((..))** can be used to evaluate arithmetic relational operators. Note that the following example is a slightly different version from the one above, which does not use the **$** in front of the parentheses:

```
if (((a / 3) >= 4))
then . .
```

The usual rules of precedence between operators are applicable; multiplication, division, and remainder operators have equal precedence over addition and subtraction. If you are unsure of the order in which the shell will interpret the operators, it is best to enclose them in brackets since the commands in brackets are executed first; brackets also make complex expressions easier to read, even though they may not be essential to ensuring the correct order of precedence.

You should also be aware that when multiplying and dividing by numbers you must be particularly careful where you actually do the dividing since divisions always exclude the remainder to produce an integer value. For example, consider the difference in the output between the following two commands:

```
$ print $((10*3/4))
7

$ print $((10/4*3))
6
```

The first command multiplies 10 by 3 first to produce 30, and then

divides by 4, to give 7 (excluding the remainder 2). The second command divides 10 by 4 first to produce 2 (again excluding the remainder 2) and then multiplies by 3 to give 6. In general, you should multiply before dividing to give the more accurate results; the same comments apply when using the modulus (**%**) operator to produce remainders.

EXAMPLE OF THE USE OF LET

Consider the following simple example, which is used to calculate (to three decimal places) the result of the division of two integers; the script ignores rounding up of the third decimal place. Arithmetic evaluation by the shell is in integer format and the only way you can get output in floating point format is by using **awk**, which will be discussed in a future article. In the meantime, let us soldier on.

To test this example, create a shell script, **div2**:

```
$ vi div2

#!/bin/ksh
# test whether second number has been
# entered and is non-zero

if [[ -z $2 ]] || [[ $2 -eq Ø ]]
then
print "Second number cannot be zero"
exit 1
fi

d=Ø
let a=$1/$2
let b=$1%$2      # remainder
for i in 1 2 3
do
[[ $b -eq Ø ]] && break      # remainder equals zero?
let "c=(b*1Ø) / $2)"
let "b=(b*1Ø) % $2)"
let "d=(d*1Ø) + c)"
done
print "Answer equals ${a}.$d"
```

The above script is not complete, since it does not allow for negative numbers; I'm sure you can see other flaws also! Before

reading the next example, try to amend the script to allow for the possibility of negative numbers so that the answer also contains the correct sign.

To remove the flaw, you can amend **div2** as follows:

```
#!/bin/ksh
# first two commands remove leading - signs
x=$(expr "$1" : "-*\(.*\)")
y=$(expr "$2" : "-*\(.*\)")

# test whether second number has been
# entered and is non-zero
if [[ -z $y ]] || [[ $y -eq Ø ]]
then
print "Second number cannot be zero"
exit 1
fi

# extract the negative signs, if any
sign1=$(expr "$1" : "\(-*\).*"| tr -s '-')
sign2=$(expr "$2" : "\(-*\).*"| tr -s '-')

[[ $sign1 != $sign2 ]] && sign=-

d=Ø
let a=$x/$y
let b=$x%$y      # remainder
for i in 1 2 3
do
[[ $b -eq Ø ]] && break      # remainder equals zero?
let "c=(b*1Ø) / $y)"
let "b=(b*1Ø) % $y)"
let "d=(d*1Ø) + c)"
done
print "Answer equals ${sign}${a}.$d"
```

You will note that in the two statements extracting the negative signs, the output from the **expr** commands has been piped into **tr -s '-'**. This is done to allow for the possibility that multiple negative signs have been entered (we are talking about seriously sad people out there!), since the **-s** option to **tr** squeezes out multiple occurrences of the specified character.

MODIFYING THE LVMAN SCRIPT

You will recall that our **lvman** script uses **getopts**, which had

certain limitations when it came to recognizing command line arguments when we entered by mistake, for example, **lvman rootvg**, or **lvman -v rootvg myvg**. To overcome these problems we have modified the script to perform further checks on the arguments before attempting processing.

```
#!/bin/ksh
# Script name: lvman
# Usage: lvman {[-v VG1name -v VG2name..| -v all]|
#            [-p PV1name -p PV2name..| -p all]}
#########################################################
# Version History
# Version      Date      Remarks
# 1.Ø                Original Version
# 1.1                Function to check VG and PV names
# 2.Ø                Modified case statements in Main to use if command
# 3.Ø                Added while statement to loop through args
#                    Modified to allow multiple VGs and PVs
# 3.1                Added f_chk_args function to further check
#                   arguments
#########################################################
#-----------------------------------------------
# Function: f_chk_args
# Checks number of command line args and options
#-----------------------------------------------
f_chk_args()
{
      if [[ $(($#%2)) -eq 1 ]]
      then
            f_dsp_usage      # odd number of args
            exit 3
      fi
      ((numpairs=$#/2))
      count=Ø
      for i in $@
      do
            if [[ $(echo "$i" | grep -c "-") -eq 1 ]]
            then
                  let count=count+1
            fi
      done
      if [[ $numpairs -ne $count ]]
      then
            f_dsp_usage      # not enough - args
            exit 3
      fi
}
#-----------------------------------------------
```

```
# Function: f_dsp_usage
# Displays usage messages
#-----------------------------------------------
f_dsp_usage()
{
    print "Usage: $(basename $Ø) {[-v VG1name -v VG2name..| -v all] |"
    print "            [-p PV1name -p PV2name..| -p all]}"
    print "Where:"
    print "\t-v VG1name -v VG2name.. specifies volume group list"
    print "\t-v all specifies all volume groups"
    print "\t-p PV1name -p PV2name.. specifies physical volume list"
    print "\t-p all specifies all physical volumes"
}
#-----------------------------------------------
# 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
    [[ $(echo "$DEV" | grep -c "-") -ne Ø ]] && return 2
    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 $VG | tail +3 | wc -l | tr -d " ")
    eval NUMLVS='$'${VG}_LVNUM
    #
    # Print output
    #
    if [[ $FIRST -ne 1 ]]
    then
      printf "\n%-20s %-15s %-15s %-15s\n" \
          "Volume Group" "Total Size" "Free Space" "Number LVs"
    fi
    printf "%-20s %-15s %-15s %-15s\n" \
          $VG "$TOTAL MB" "$FREE MB" $NUMLVS
}
#-----------------------------------------------
# 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 $PV | tail +3 | wc -l | tr -d " ")
    eval NUMLVS='$'${PV}_LVNUM
    #
    # Print output
    #
    if [[ $FIRST -ne 1 ]]
    then
      printf "\n%-20s %-15s %-15s %-15s\n" \
          "Physical Volume" "Total Size" "Free Space" "Number LVs"
    fi
    printf "%-20s %-15s %-15s %-15s\n" \
          $PV "$TOTAL MB" "$FREE MB" $NUMLVS
}
################################################
# Main section
################################################
f_chk_args $@                  # check numbers of args
while getopts :v:p: opt
do
    case $opt in
    v)
```

```
  if [[ $OPTARG = all ]]
  then
      VGS=$(lsvg -o | sort)      # all volume groups
  else
      VGS=$OPTARG
  fi
  FIRST=Ø
  for VG in $VGS
  do
      f_chk_valid $VG
      case $? in
      Ø)
        f_get_vg_space $VG
        FIRST=1 ;;
      1)
        print $VG is not a valid volume group
        exit 1 ;;
      2)
        f_dsp_usage ;;
      esac
  done ;;
p)
  if [[ $OPTARG = all ]]
  then
      pvs=$(lsdev -Cc disk -r name)      # all physical volumes
      for pv in $pvs
      do
        if [[ $(lsdev -Cl $pv | grep -c Available) -eq 1 ]]
        then
            PVS=$PVS" $pv"            # only want Available disks
        fi
      done
  else
      PVS=$OPTARG
  fi

  FIRST=Ø
  for PV in $PVS
  do
      f_chk_valid $PV
      case $? in
      Ø)
        f_get_pv_space $PV
        FIRST=1 ;;
      1)
        print $PV is not a valid physical volume
        exit 2
        ;;
      2)
```

```
            f_dsp_usage ;;
          esac
      done ;;
    *)
      f_dsp_usage
      exit 3 ;;
    esac
done
```

You will note that we have added a further function, **f_chk_args**, to take advantage of our new-found skills in manipulating arithmetic variables, and which we call at the start of the main section.

We know that we should have an equal number of command line arguments and so the function first checks whether we have an odd number, using the test **[[$(($#%2)) -eq 1]]**. This divides the number of arguments by 2, extracts the remainder, and compares it with 1; if they are equal then there is an odd number of arguments and the usage message is displayed.

If we have an equal number of arguments, the next part of the function checks how many pair combinations we have, using **((numpairs=$#/2))**, and then runs the **for** loop with a **count** variable to determine the number of minus signs we have in our argument list.

The remainder of the function compares the number of minus signs with the number of pair combinations since we assume that there should be one **-v** or **-p** option for each argument pair.

This function will allow error-checking for perhaps most of the possible argument combinations that may be entered in error, but it is still by no means perfect. In fact this has now become the classic example of how *not* to develop a script. We started by using **getopts**, a command whose limitations did not become obvious until after extensive use and testing. We continued to add error checking bits to the script until we have now got to the stage where we have a rather clumsy script which still does not do all the checks we would like.

A more ideal solution would be to scrap **getopts** all together and instead use a **case** statement, combined with a number of **shift**

commands, to perform more comprehensive checks on each and every argument instead of using the kludge we currently have. Perhaps you would care to rewrite the main section to perform its error-checking in this way.

Tonto Kowalski
Guru (UAE)

If you have ever experienced any difficulties with AIX, or made an interesting discovery, you could receive a cash payment, a free subscription to any of our *Updates*, or a credit against any of Xephon's wide range of products and services, simply by telling us all about it.

More information about contributing an article to a Xephon *Update*, and an explanation of the terms and conditions under which we publish articles, can be found at http://www.xephon.com/nfc. Alternatively, please write to the editor, Trevor Eddolls, at any of the addresses shown on page 2, or e-mail him at trevore@xephon.com

Veritas has announced availability of Veritas Database Edition for DB2 for the first time on AIX, integrating core volume management and file system tools specifically optimized, says the vendor, to make DB2 databases perform up to two times faster, more highly available, and easier to manage.

The software provides raw partition performance and data integrity for databases stored in file systems, which means administrators don't have to manage raw partition databases.

Using the product's cached Quick I/O feature, DB2 performance is said to be up to two times faster than the same database stored on a standard Unix file system, and is comparable to or faster than raw partition database performance.

Downtime is reduced by enabling administrators to reconfigure storage and data while the database is online, without disrupting user access. Using the FlashSnap Option, administrators can create point-in-time copies of the data. The HA version integrates Cluster Server to help further reduce downtime and increase database availability.

For further information contact:
Veritas, 350 Ellis Street, Mountain View, CA 94043, USA.
Tel: (650) 527 8000.
URL: http://www.veritas.com/db2guided/db2_2.html.

* * *

Reconda International has launched its QN-StatWatch, designed to collect statistics at the channel, queue, and message level. The browser-based WebSphere MQ and WMQI support application provides MQ administrators, system architects, and managers with the data needed to facilitate accurate charge-back, SLA compliance, and resource capacity planning.

It supports every platform on which WebSphere MQ Server or Client runs, and runs on AIX, Solaris, and Windows NT/2000.

For further information contact:
Reconda, 15 East Putnam Avenue, Suite 306, Greenwich, CT 06830, USA.
Tel: (203) 299 4000.
URL: http://www.reconda.com/productsFrame_QN-StatWatch.html.

* * *

IBM has announced its 7205 Model 550 160GB external digital linear tape drive, a stand-alone, SCSI streaming device that attaches externally to pSeries and RS/6000 servers.

The drive writes data to tape using a laser-guided recording technique, providing a media capacity of up to 160GB (320GB with 2:1 compression) data storage per cartridge. It has a sustained data transfer rate of up to 32MB per second with compression, which provides for tape storage back-up at the rate of 115GB per hour.

It's positioned to provide a migration path from the 7205 Model 440 DLT8000 tape drives, and the increased storage capacity, up to 160GB (320GB with compression), is four times the capacity of the 7205 Model 440.

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