89

AIX update

March 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

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.

AIX 5L performance analysis tools enhancements

Among many areas affected by the introduction of AIX 5L are performance analysis tools. The tools are members of the following filesets:

- bos.sysmgt.trace
- bos.perf.perfstat
- perfagent.tools.

The perfagent.tools filseset is a pre-requisite for the installation of the Performance Tool Box (PTX) product and is dependent on the installation of the two former filesets.

The following tools have been withdrawn from AIX 5L: **bf**, **bfprt**, **lockstat**, **stem**, and **syscalls**. Some of the functionality of these tools is supported by **svmon**, **locktrace**, and **truss**.

The **truss** command provides the ability to trace the execution of system calls performed by application programs.

The **alstat** command reports computer instruction alignment statistics.

The following commands have been carried on from earlier versions of AIX: **genname**, **iostat**, **vmstat**, **sar**, **prof**, **tprof**, **gprof**, **emstat**, **filemon**, **fileplace**, **netpmon**, **pprof**, **rmss**, **svmon**, and **topas.**

PERFORMANCE ANALYSIS LIBRARIES

AIX 5L introduces APIs that enable convenient access to system performance data.

The performance Monitor API provides access to Performance Monitor counters for the following processor types: Power PC 604, Power PC 604e, POWER3, POWER3-II, RS64-II, RS64-III, and RS64-IV. This API can serve as a foundation for applications looking to optimize computationally-intensive programs.

The APIs included in the Perfstat library have a different purpose – their goal is to facilitate the writing of system performance and monitoring programs in a portable way, without the need to analyse the */dev/kmem* and avoiding dependencies on kernel data structures, which can change between the releases of the operating system. The following summarizes the available APIs:

- perfstat_cpu() – retrieves individual CPU usage statistics.
- perfstat_cpu_total() – retrieves global CPU usage statistics.
- perfstat_disk() – retrieves individual disk usage statistics.
- perfstat_disk_total() – retrieves global disk usage statistics.
- perfstat_memory_total() – retrieves global memory usage statistics.
- perfstat_netinterface() – retrieves individual network interface usage statistics.
- perfstat_netinterface_total() – retrieves global network interface usage statistics.

The perfstat library is included in the bos.perf.libperfstat fileset.

The directory */usr/samples/libperfstat* contains a single example file – perfstat.c.

The following is the output produced on my server:

```
>-- Statistics regarding the network interface : en1
> Description of the network interface         : Standard Ethernet
Network Interface
> Type the interface                           : 6
> Network frame size                           : 1500
> Packets received on interface                : 18082325
> Input bytes on interface                     : 0
> Input errors on interface                    : 1505300160
> Packets sent on interface                    : 177271
> Output bytes on interface                    : 82947182
> Output errors on interface                   : 0
```

```
> Collisions on csma interface                  : Ø
        Press return key to proceed...
>-- Statistics regarding the network interface : loØ
> Description of the network interface         : Loopback Network
Interface
> Type the interface                           : 24
> Network frame size                           : 16896
> Packets received on interface                : 2586Ø6
> Input bytes on interface                     : Ø
> Input errors on interface                    : 31273899
> Packets sent on interface                    : 259111
> Output bytes on interface                    : 31288Ø39
> Output errors on interface                   : Ø
> Collisions on csma interface                 : Ø
        Press return key to proceed...
> Number of disks                    : 4
> Sum of the size of the disks       : 6444 MB
> Sum of the free space of the disks : 436 MB
> Average xfer rate capability       : Ø kbytes/sec
> Total transfers to/from disks      : 37635Ø
> Blocks written to all disks        : 323Ø851
> Blocks read from all disks         : 6974128
> Amount of time disks are active    : 345483
        Press return key to proceed...
>-- Statistics regarding the CPU : cpuØ
> User time used                 : 839833 ticks
> System time used               : 184168Ø ticks
> Idle time used                 : 411ØØ161 ticks
> Wait time used                 : 2Ø3477 ticks
> Number of process switch       : 394349Ø1
> Number of syscalls             : 3Ø2145814
> Number of system read          : 1568769
> Number of system write         : 4Ø8Ø97
> Number of forks                : 1Ø373
> Number of execs                : 1Ø992
> Number of read characters      : 1332736383
> Number of written characters   : 353397285
        Press return key to proceed...
```

TRUSS – PROCESS SYSTEM CALLS TRACING UTILITY

The ability to trace execution of system calls invoked by a specific process is a very handy performance and debugging tool. In previous versions of AIX it was supported by **trace** and **trcrpt** commands. AIX 5L introduces a **truss** command that works similarly to the identically-named command in Solaris or the **tusc** command of HP-UX. **Truss** can attach to a process specified by a pid or it can invoke a process specified as one of its command line parameters.

One of the important flags of this command is **-c**, which generates a profile summary of the command being **truss**ed:

```
# truss -c grep lp /etc/passwd
lpd:!:9:4294967294::/:
lp:*:11:11::/var/spool/lp:/bin/false
syscall              seconds   calls  errors
execve                   .ØØ       1
__loadx                  .Ø2      12
_exit                    .ØØ       1
close                    .ØØ       2
kwrite                   .ØØ       2
kread                    .ØØ       2
_getpid                  .ØØ       1
getuidx                  .ØØ       3
kioctl                   .ØØ       2      1
open                     .ØØ       1
getgidx                  .ØØ       3
sbrk                     .ØØ       3
access                   .ØØ       1
kfcntl                   .ØØ       2
                        ----      --    ---
sys totals:              .Ø4      36      1
usr time:                .ØØ
elapsed:                 .Ø4
```

The **-e** flag directs **truss** to display the environment variables present in the environment of the **truss**ed program. The **-a** flag displays the parameter strings that are passed in each executed system call.

EMSTAT AND ALSTAT – PROCESSOR INSTRUCTIONS EMULATION AND ALIGNMENT DETECTION UTILITIES

In addition to the existing **emstat** tool, which reports the amount of processor instructions that have to be emulated by the available computer hardware, the new **alstat** command displays the number of alignment interrupts that occur during the execution of applications.

Both commands feature the **-v** flag, which reports the statistics per CPU in SMP systems.

TPROF – CPU USAGE BREAKDOWN AT THE SYSTEM AND INDIVIDUAL PROGRAM LEVELS

The **tprof** command produces reports that detail the CPU usage caused by individual components (function, class, method) of individual programs. This utility has been enhanced to include the profiling of Java applications.

LOCKTRACE – SYSTEM KERNEL LOCKS TRACING UTILITY

The **locktrace** command is a replacement for the **lockstat** command. It produces reports detailing the statistics that describe the locking activity that occurs in the system.

This command is affected by another system utility, **bosboot.** If the system has been rebooted after the command **bosboot –L** was executed, the **locktrace** command will be able to report locking for individual lock classes (as well as for specific lock classes). If the **bosboot –L** command has not been executed, the **locktrace** will be able to trace all classes of lock but will display only partial information.

VMSTAT – SYSTEM THREADS STATISTICS REPORTING UTILITY

The **vmstat** command is used to report statistics about kernel threads in the run and wait queues, memory, paging, disks, interrupts, system calls, context switches, and CPU activity.

The **vmstat** command has two new flags introduced in AIX 5L; these flags extend the available reports.

The **-I** flag adds two new columns for number of file pages paged in (fi) and out (fo). When this flag is specified the columns re and cy are not displayed. A new p column displays the number of threads waiting for physical I/O operations:

```
# vmstat -I 1 3
  kthr      memory              page              faults        cpu
-------- ---------- ----------------------- ------------ -----------
 r  b  p   avm   fre  fi  fo  pi  po  fr  sr   in    sy  cs us sy id wa
 1  1  Ø 62391   125   Ø   Ø   Ø   Ø   Ø    1 19Ø 78465 89  2  4 93  Ø
 Ø  Ø  Ø 62395   123   Ø   Ø   Ø   2   Ø    Ø 183  2398 83  2  7 91  Ø
 Ø  Ø  Ø 62395   123   Ø   Ø   Ø   Ø   Ø    Ø 175  2367 87  2  7 91  Ø
```

The **-t** flag shows a time stamp at the end of each line reported by vmstat:

```
# vmstat -t 1 3
kthr     memory             page              faults        cpu      time
---- ---------- --------------------- ----------- --------- -------
r  b   avm fre  re  pi  po  fr   sr cy  in   sy   cs us sy id wa hr mi se
1  1 62391 469   Ø   Ø   Ø   Ø    1  Ø 19Ø 78452  89  2  4 93  Ø 22:1Ø:48
Ø  Ø 62397 463   Ø   Ø   Ø   Ø    Ø  Ø 18Ø  24Ø7  83  2  5 93  Ø 22:1Ø:49
1  Ø 62397 463   Ø   Ø   Ø   Ø    Ø  Ø 197  246Ø  87  1  6 93  Ø 22:1Ø:5Ø
```

IOSTAT – DISK I/O STATISTICS REPORTING UTILITY

The **iostat** command is used to report statistics about CPU and I/O activity for TTY devices, disks, and CD-ROMs. The reports produced by **iostat** can be used in order to perform fine-tuning of storage allocation to improve the input/output load distribution between physical disks.

The **iostat** command has two new flags introduced in AIX 5L; these flags extend the available reports.

The **-s** flag adds a new line to the header line of each set of statistics that reports the sum of all activity on the system:

```
# iostat -s 1 1

tty:      tin     tout   avg-cpu:  % user    % sys     % idle    % iowait
          Ø.Ø      2.4              1.9      4.2       93.4       Ø.5

System: rsc2Ø4
                           Kbps       tps    Kb_read   Kb_wrtn
                          11.8        Ø.9    3575365   1653453

Disks:        % tm_act     Kbps       tps    Kb_read   Kb_wrtn
hdisk2           Ø.Ø       Ø.5        Ø.Ø      563Ø6    179674
hdisk1           Ø.2       1.Ø        Ø.2      75327    379795
hdiskØ           Ø.6      1Ø.3        Ø.7    3443732   1Ø93984
cdØ              Ø.Ø       Ø.Ø        Ø.Ø          Ø         Ø
#
```

The **-a** flag specifies the generation of a report that details adapter-based statistics of I/O activities. After the display of adapter statistics, the statistics for disks connected to a specific adapter are displayed:

```
# iostat -a 1 1

tty:      tin     tout   avg-cpu:  % user   % sys     % idle    % iowait
          Ø.Ø      2.4              1.9      4.2       93.4       Ø.5

Adapter:                  Kbps      tps    Kb_read   Kb_wrtn
ascsiØ                   11.8       Ø.9    3575541   1653558

Disks:        % tm_act     Kbps      tps    Kb_read   Kb_wrtn
hdisk2           Ø.Ø       Ø.5       Ø.Ø      563Ø6    179674
hdisk1           Ø.2       1.Ø       Ø.2      75451    379823
hdiskØ           Ø.6      1Ø.3       Ø.7    3443784   1Ø94Ø61

Adapter:                  Kbps      tps    Kb_read   Kb_wrtn
scsiØ                     Ø.Ø       Ø.Ø          Ø         Ø

Disks:        % tm_act     Kbps      tps    Kb_read   Kb_wrtn
cdØ              Ø.Ø       Ø.Ø       Ø.Ø          Ø         Ø
```

GENNAMES – COLLECT INFORMATION NEEDED FOR OFF-LINE SYSTEM TRACING

The **gennames** command is used to collect information needed for off-line execution of the **tprof**, **filemon**, **netpmon**, or **pprof** commands.

NETPMON – PROCESS DATA OFF-LINE

The **netpmon** command, which monitors network I/O and network-related CPU activity and reports usage statistics, has been enhanced to enable off-line batch processing of normal trace report files.

Below is a typical sequence that demonstrates usage of this feature:

- Step 1: generate unformatted system trace file:

```
 # trace
-> trcon
-> trcstop
-> trcoff
-> q
# ls -l /var/adm/ras/trcfile
-rw-rw-rw-  1 root   system    1344488 Dec 2Ø 22:38 /var/adm/ras/trcfile
```

- Step 2 – immediately following the collection of trace information, execute **gennames** command and save its output:

```
# gennames > /tmp/gennames.out
```

- Step 3 – format the collected trace file using the **trcrpt** command:

```
# trcrpt –r /var/adm/ras/trcfile > /tmp/trcrpt.out
```

- Step 4 – generate the **netpmon** report at your convenience using the -i and **-n** flags:

```
# netpmon –i /tmp/trcrpt.out –n /tmp/gennames.out
```

FILEMON – PROCESS DATA OFF-LINE

The **filemon** command, which monitors the performance of the file system and reports the I/O activity on behalf of logical files, virtual memory segments, logical volumes, and physical volumes, has been extended in a fashion similar to **netpmon**. Step 4 in the above sequence should be changed to:

- Step 4 – generate the **netpmon** report at your convenience using the **-i** and **-n** flags:

```
# filemon –i /tmp/trcrpt.out –n /tmp/gennames.out –O all
```

SVMON – CAPTURE AND ANALYSIS OF VIRTUAL MEMORY STATISTICS

The **svmon** command**,** which monitors the performance of the virtual memory, has been enhanced to provide reports on various elements of Workload manager system such as superclasses, subclasses, and tiers.

The **-W** flag directs the command to collect statistics for a specific superclass.

The **-e** flag directs the command to collect statistics for a specific subclass of a superclass.

The **-T** flag directs the command to collect statistics for classes of a specific WLM tier.

TOPAS – REPEATEDLY DISPLAY SYSTEM STATISTICS IN TERMINAL EMULATOR WINDOW

The **topas** command, which was introduced in AIX 4.3, has been enhanced with new capabilities. It is now possible to display the individual CPU usage statistics by typing the **c** (lower case) command into the tool's screen.

The default screen will include information about the two busiest WLM classes. The display of this information can be toggled by typing the **w** (lower case) command into the tool's screen. The **W** (upper case) command will select the display of the workload manager classes monitoring screen.

PMAPI – HARDWARE PERFORMANCE MONITOR API

A new set of APIs is provided to allow access to hardware performance counters available on PowerPC 604, PowerPC 604e, POWER3, POWER3-II, RS644-II, RS64-II, and RS64-IV processors.

The directory */usr/samples/pmapi* contains sample programs demonstrating the potential of this library. One nice utility using it is the **pmcycles** command, which reports the processor clock speed of your computer.

SUMMARY

In this article I have described only a portion of performance tools enhancements introduced in AIX 5L. I will provide additional information in future articles.

REFERENCES

1 *AIX 5L Version 5.2 Commands Reference* Volume 1 to 6, SG24-2014-01, IBM Corporation.

2 *AIX 5L Version 5.2 Performance Management Guide*, SG24-2014-01, IBM Corporation.

3 *AIX 5L Version 5.2 Performance Tools Guide and Reference*, SG24-2014-01, IBM Corporation.

4 *AIX 5L Performance Tools Handbook*, SG24-6039, IBM Corporation.

5 *AIX 5L Differences Guide Version 5.2 Edition*, SG24-5765-02, IBM Corporation.

Alex Polak
System Engineer
APS (Israel)

Good practice in shell programming

INTRODUCTION

The importance of using programming standards cannot be over-emphasized. The proper use of good programming standard ensures that:

- Proven programming practices are used
- Programs are easier to read and have a more professional look
- Programs are easier to maintain
- Each unit of program has a similar look and feel
- Better quality programs are produced.

PORTABILITY

Portability is crucial if we are to move shell scripts off AIX to other flavours of Unix.

This can be implemented by:

```
OS='uname'
If  [  "${OS}"  =   "AIX" ]
then
          COMMAND=<command  specification>
elsif   [ "${OS}"  =  "SunOs"  ]
Then
         COMMAND=<command specification>
fi
#
#issue   command
#
$COMMAND
```

ROBUST SHELL PROGRAMMING

Remember with error checking:

- Check every function call for error return, unless you know that you wish to ignore errors.
- To give debugger and support staff a chance, wherever possible error messages should at least be in the form:

```
Source-file-name:lineno;message
```

- You may also want to write your error message with the following format, with non-interactive programs:

```
Program;source-file-name:line:message
```

- If you have no relevant source file, use the following format:

```
program: message
```

- You may also want to output the column number. Do this with the following

```
Format:
Program:source-file-nanme:lineno:column: message
```

  Line numbers start at 1 and column numbers start at 1

- Start error messages with a capital, but, to avoid any possible confusion with meaningful shell programming usage, do not use full stops.

INTERFACE STANDARDS

Don’t make your script reliant on its own filename, or the name of any scripts that have either called it or it calls. The names of the scripts should not make any difference to the way they run because this gives us greatly enhanced portability.

Don’t let your script rely on the input and output characteristics of the operating system you happen to be on at the moment. For instance, NT may add a special ^M character to the end of some input file lines. Don’t rely on this. Always make sure your program will behave in the same way, on another operating system, which doesn’t have this special behaviour.

OTHER ROBUSTNESS TIPS

Try and avoid low-level interfaces to obscure programs (which may or may not exist on other systems). For instance, if you are making special use of **awk**, make sure this exists on all the systems that the shell script may be ported to. Even better, try and avoid using this in the first place.

When writing temporary files, check the TMPDIR environmental variable and use this. It should be defined.

Similarly, always use generic environmental variables, whenever given the opportunity (HOME, PWD, etc).

Be aware of memory usage. If using cat or tail operations, these can sometimes use up large swathes of memory. Be prepared for this.

FORMATTING SHELL SCRIPTS

Indentation

Use three spaces indenting to give clear indentation which does not rely on what the tabspace key is currently set to, and which does not cramp the right-hand side of the file too quickly.

For instance:

```
If  condition
Then
   Do_something
Else
   If  another_condition
   Then
      Do_something
   fi
fi
```

Alternatively:

```
case   $COPY_FLAG  in

   a)   action  add_script
        ;;

   b)   action  blank_script
        ;;

   c)   action  cut_script
        ;;

   *)   action  default_script
        ;;
esac
```

VARIABLE NAMING

To reiterate, make the names meaningful, to get self-documentation commentary going.

As is traditional, all environment variables are in uppercase and all shell variables are in lower case.

When naming local variables, to avoid over-writing global ones, make the local name as locally specific as possible. For instance, don't use $HOME for a local variable (which may be perfectly rational within the context of the program, but use something like:

```
$HOME_FOR_RATED_CUSTOMERS
```

Declare each new variable on a separate line, to aid readability and clarity. Don't declare a list of variables separated by semi-colons, on a single long line.

Separate elements of a variable name with an underscore in the traditional C programming style, and don't use the Java programming style, which avoids underscores. For instance, use:

```
$HOME_ON_THE_RANGE
```

rather than:

```
$homeOnTheRange
```

Be careful with abbreviations. For instance, does $HOME_STAT mean 'Home Static' or 'Home Status'? You may know right now, but six months down the line it may not be so clear. Where any ambiguities arise, clarify them there and then, for example, use $HOME_STATUS.

When naming soft-coded constants, try to make this clear in the name. For example:

```
$CLOSED_STATUS_CONSTANT
```

COMMENTS

General notes on comments

We are trying to do as much self-documentation as possible within these coding guidelines. To explicitly supplement this approach, the following guidelines may be useful:

- Write straightforward code and avoid clever tricks.
- Wherever there's any doubt, comment.
- Use filename, function, and variable names which make sense in the real world (for instance close_input_file as a function name, rather than c1).
- Always employ a clean consistent layout with as much self-commenting as possible, because of its obvious indentation, use of white space, and clearly sign-posted variable names.

The use of literals and constants

Use named constants rather than literal lines ($AGREED_CUTOFF_FIGURE rather than 72).

Remove all hard-coded literals, within reason, from your code and replace with soft-coded environmental variables to aid future portability and ease of maintenance. Should ‘72’ become set to ‘102’ in the future, with soft coding, the value will have to be updated in only a single location.

To follow this up, allow the value of that literal to be set in only one place within your code (where it can be most easily maintained), preferably within a sourced file, including all of the exported soft-coded environmental variables your program will require.

If you are using these environmental variables only locally, put them all at the top of script where possible, so that they can all be referenced in one place.

It will be easier to maintain these scripts in the future, the less hard-coding there is within them.

Comment as you code

When you’re writing code, you know what you’re trying to do, and it hardly takes any time to make a conmment as you’re thinking through the coding problem.

However, when you finish a large project, you’ve often forgotten what you’ve done in the 300 files you’ve updated and created. You would have to go back and comment them all. Plus you’ve probably got no time available anyway.

Therefore, comment as you code.

How to comment

What do you think of the following commented code?

```
# Setting  flag  to  'C'
FLAG1="C"
```

How about this alternative (where $COMPLETED has been set to ‘C’ beforehand):

```
#Make sure the monthly sales flag is set to 'C' to indicate
# completed status, at the end of each month
MONTHLY_SALES_FLAG=$COMPLETED
```

Hopefully you’ll agree the new comment makes it clearer what the code is attempting to do.

This is because it explains why you’re doing rather than how, which the code is telling you anyway. Always think why with a comment, rather than how.

Easy maintenance

What do you think of the following comment ?

```
#=======================================
#  I'm  a  lovely  comment
#    far lovelier  than,
#  a  cloud  or  daffodil
#========================================
```

This may look like a neatly formatted comment, but it’s a real pain to change, especially if the indentation changes, or more testing is required, for whatever reason – especially if you’ve got hundreds of lines to do. Therefore, avoid too much beautification of comments. We recommend the following for longer comments, as to lengthen or differently indent the comment is no real problem at all:

```
#  I'm  a lovely  comment
#   far lovelier  than,
# a cloud  or  a  daffodil
#
```

Comment indentation

To avoid confusion, always indent your comment directly with the code you’re commenting on, eg:

```
# I'm commenting  on the "if"  statement
 if    some_condition
then
```



```
        # now  I'm commenting  on  the  action
        some_action
fi
```

Commenting declarations

Once again, to make sure we always consider the poor chap six months down the line from us, who has to maintain our carefully crafted code, provide a comment for each and every variable declaration

No matter how well we name our variables, there's always room for a more in-depth description, which the latter will remind us of, rather than provide the entire explanation for. Use the following style to comment on declarations:

```
# The following variable is used to indicate the sales ratings
# for each customer.
CUST_SALES_RATING=Ø
```

File and functional headers and comments

This section attempts to define a standard header that will be included for all script files. The method encouraged in this document is to use the source control system to automatically generate this header. This will ensure that the header will include owner information, history of script, modifications, etc. There will also be minimal effort on the part of the programmer to keep the header up to date.

To achieve this, add the following to the top of your program:

```
#$Author$
#$Date$
#$Header$
#$Id$
#$Locker$
#$Log$
#$Name$
#$Rcsfilele$
#$Revision$
#$Source$
#$State$
```

The script header at the top of the file should also contain at least the following comment template:

```
# Author ;  A  Body
#
# Overview:  This script demonstrates the use of script commenting.
#
```

Each function should also contain a description comment. If the function returns a value, it should describe what it returns (it may also be useful to say it returns void, if it doesn't return anything):

```
#
# Function purpose: This function demonstrates comment
#
# Arguments:
#     $1   The number of comments in a file
#     $2   The number of variables in the file
#
#
#  Returns;
#  $?  The number of letters in all the comments in the  file
#
```

Within the main body of your shell program, whenever in doubt, make a comment. There should be at least one comment for every single logical operation. Use white space effectively to separate these logical operations, so as to tie up the relevant comment to the relevant piece of code. The meaning of your code should be completely clear to someone in a year's time, who has to open it up to maintain it. If you suspect it won't be, improve it with comments, variable naming, and white space until it will.

#! /bin/ksh shell directive

On the first line of a script, following the '#!', is the name of the program that should be used to interpret the contents of the file. For instance, if the first line contains '#! /bin/ksh', the contents of the file will be executed using ksh.

You can get away without this, but you shouldn't. All good scripts state the interpreter explicitly. Long ago there was just one (the Bourne shell) but these days there are many interpreters – csh, ksh, bash, and others.

Search path

A PATH specification is recommended – often a script will fail for some people because it has a different or incomplete search path.

Usually, all the standard locations will be included in the PATH specification that should exist in */etc/environment* and/or */etc/profile*. In this case, you can customize PATH specifications as follows:

```
PATH=${PATH}:/u1/users/azaman/bin    ;   export   PATH
```

where $PATH is the standard path specification.

But if in doubt about the standard PATH specification, specify it fully as follows:

```
PATH=/etc:/etc/bin:/etc/sbin:${ORACLE_HOME}/bin:/u1/users/azaman/bin:.
;  export   PATH
```

EXIT STATUS

All scripts should return a meaningful exit status. Usually, a script would return 0 for successful completion or 1 for abnormal termination. On many occasions, an abnormal termination may be caused by a variety of reasons and, in those cases, a pre-defined exit status for each abnormal termination of the script would give support personnel a head start in trying to establish the reason for the script failure. Therefore, it pays to define exit statuses to cater for different scenarios under which the script will fail and document these in the header of the script.

COMMAND LINE PARAMETER SPECIFICATION

The command line parameters for a shell script can be specified in one of two ways:

- By value specification

- By option and value pair specification

Value specification looks like:

```
my_script.sh      Y    1    2ØØ
```

Notes:

1 The shell script takes three parameters and the specified values are Y, 1, and 200.

  It is not clear from the command line what these parameters mean and it is therefore necessary to examine variables that are assigned these values to understand the meaning of the parameters.

2 Segments of code assigning parameters will look like this:

```
PARAM1=$1
PARAM2=$2
PARAM3=$3
```

  These variable names are not very helpful and therefore should be renamed – for example as:

```
RERUN_FLAG=$1
PROCESSING_MOTH=$2
BATCH_SIZE=$3
```

Argument and value pair specification looks like:

```
my_script.sh   RERUN_FLAG=Y   PROCESSING_MONTH=1   BATCH_SIZE=2ØØ
```

Notes:

1 The shell script takes three parameters and these are clearly understood from the command line.

2 A function such as ParseCommandLine should be included to process the command line:

```
######################################################################
# Name     : ParseCommandLine
# Overview : The function parses command line options which are
#            specified using argument=value syntax.
# Notes    : 1. In order to use this function, change the following:
```

```
#                   - define  $ARGLIST
#                   - plug in appropriate validation for expected
#                     arguments
######################################################################
ParseCommandLine ()
{
#
# make a list of expected arguments along with its optionality flag
#
ARGLIST="RERUN_FLAG:Y       \
         PROCESSING_MONTH:Y \
         BATCH_SIZE:Y       \
         DEBUG:N              "
#
# make an empty list for holding arguments to be provided
#
ARG_PROCESSED=""
#
# count argument entries
#
MAN_NO_ARGENTRY=Ø
TOTAL_NO_ARGENTRY=Ø
for  ARGENTRY in ${ARGLIST}
do
   if [ "`echo "${ARGENTRY}" | cut -d':' -f2`" = "Y"  ]
   then
       MAN_NO_ARGENTRY='expr ${MAN_NO_ARGENTRY} + 1'
```

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

Arif Zaman
ETL Developer (UK)

Controlling signals and processes

In this article we will discuss some of the Korn shell features used for handling processes, such as trapping and ignoring signals, interprocess communication and coroutines, and how these features can be used in shell programming.

It will be assumed that you are familiar with standard Unix features such as process IDs, job control, and running commands in the foreground and background.

CATCHING AND IGNORING SIGNALS

A process can be interrupted many times throughout its life. Sometimes the interruptions are caused by the process itself, such as when initiating an I/O operation, or they can be caused by external events entirely unrelated to the process's execution.

For example, if you are running the **mailto** script we created in *Input for shell scripts*, *AIX Update*, Issue 81, July 2002, and you press **Ctrl C**, an interrupt signal will be sent to **mailto**. Normally such signals kill the process to which they are sent and in this case we may be left with temporary files which we no longer need. We shall see later how we can remove these files when the script is interrupted.

Generally speaking, a signal is simply a message that one process sends to communicate with another; the message may tell the receiving process of the occurrence of an unusual event, in which case the receiving process may choose to ignore it or do something else.

Programs can be written so that they catch signals, or ignore them:

- To catch a signal means to execute one or more commands when the signal is received.
- To ignore one means proceed as if nothing happened.

There are several different types of signal used to notify processes of possible error conditions, or unusual occurrences. Sometimes a programmer will decide that a certain signal does not really indicate an error, and that the signal should be ignored. At other times, he may decide that although the program should die when it receives a particular signal, it should perform some special action, such as removing temporary files, as we discussed above, before it finishes.

SIGNALS AVAILABLE

Signals have numbers and names, and you can get a list of all the signals generated by the operating system by running **kill -l**. Signal names tend to be standard across different Unix operating systems and it is advisable to use names if you are considering the portability of your scripts; signal names can be either lower or upper case, and some signals have alternative names, such as **SIGHUP** for **HUP**.

The following is a list of some of the operating system signals (and their names) commonly used in shell programming. The signal number is followed by its name in brackets.

- **1** (**SIGHUP** or **HUP**) **hangup**

  This signal is generated when you log out, shut off your terminal, or hang up when using a dialled remote connection. The signal is also sent to all background processes associated with your terminal (or window) when **Ctrl D** is pressed, or you enter the **exit** command.

  If you have a job running in the background and you try to log off before it completes by entering **Ctrl D**, a hangup signal will be sent and you will be reminded that there are background jobs. On entering **Ctrl D** the second time, the process will be killed and you will be logged off.

- **2** (**INT**) **interrupt**

  This signal is sent to processes associated with a terminal, or window, which are currently running in the foreground, and the user presses **Ctrl C**. Processes that run in the background *automatically ignore* the interrupt signal.

- **3** (**QUIT**) **qui**

  This signal is generated by the terminal device driver for the quit key combination and will normally cause a core dump and create a **core** file in your current directory. On many terminals the quit signal is **Ctrl **; you can confirm the combination by running **stty -a**.

  If **Ctrl C** fails to kill a process, then the quit key combination will most likely do so.

- **9** (**KILL**) **kill**

  This is a special signal not associated with a **Ctrl+*key*** combination that cannot be caught or ignored, and will kill the process to which it was sent; it cannot be sent to your current process in the same way that the interrupt and quit signals can.

- **15** (**TERM**) **terminate**

  This is the signal that the **kill** command sends by default and it usually allows a graceful shutdown of the process, giving it time to clean up.

The integers and names associated with each type of signal can be used as arguments to both the **kill** and **trap** commands. When you send any of these signals to a particular process by using the **kill** command, the process will be killed unless the process catches or ignores the signal.

The syntax for the **kill** command is:

```
kill -signal_number PID
```

or:

```
kill -signal_name PID
```

where *signal_number* and *signal_name* are the number and name associated with the type of signal that is to be sent, and *PID* is the process ID of the process the signal is to be sent to.

In addition to the signals generated by the operating system, a further three signals are generated by the shell itself that can be used in **trap** statements – you can use their names only. They are used extensively for debugging and will be discussed in detail later. They are:

- **exit** (**EXIT**)

  This signal is sent to the system when the function or script within which it was set exits.

- **error** (**ERR**)

This signal is generated whenever a command in the surrounding script or function exits with a non-zero exit status.

- **debug** (**DEBUG**)

  This signal causes the **trap** *command* (see below) to be run after every statement in the surrounding script, or function, has finished executing.

RULES FOR CATCHING AND IGNORING SIGNALS

A child process that is started by a parent which ignores a particular signal will also ignore the same signal. This is fairly logical since if we run a script containing code to ignore a particular signal, we would expect that all subsequent commands in the script would also ignore the same signal, otherwise there wouldn't be much point in trapping it in the first place!

The same does not apply to processes that catch signals. If a child process is started by a parent that catches a given signal, the child process will not automatically catch the same signal.

For example, suppose you have a script that catches the interrupt signal, and the script also starts a **sort** process. If you run the script in the foreground and press **Ctrl C**, the **sort** process will be killed, even though the signal has been caught by the parent process. On the other hand, if the script starts a **chdev** command, it also will catch the signal since it is considered important that **chdev** is not interrupted during its execution.

It is not always obvious what the results will be within scripts when you are catching and ignoring signals and it is often necessary to experiment before achieving the desired result.

THE TRAP COMMAND

You can use the built-in **trap** command to specify what a shell script should do when it receives a particular signal. It is frequently used for clean-up situations when large scripts are subjected to abnormal events.

Whether you actually need to use traps in a script is really determined by what might happen should an unusual event occur. If you are running a script that needs to continue should you log out, then most likely a trap to ignore **HUP** is justified. If it means that your script terminates without removing a temporary file should it receive the interrupt signal, this is less likely to be quite as earth-shatteringly important; nice to clean up, yes, but probably not essential.

The **trap** command with no arguments prints a list of commands associated with each signal number. In a script, it will only print out the traps that have been interpreted prior to the command itself; any subsequent traps will not be listed.

CATCHING SIGNALS

The syntax of the command to catch signals is:

```
trap 'command' signal1 signal2 . . .
```

where *signal1*, *signal2* etc, are the signal numbers or names of the signals that are to be caught, and *command* is the command to be executed when one of the specified signals is caught – this may be a single command with or without arguments, a series of commands separated by semi-colons, another shell script (use the full path name for safety), or a function.

The command to be executed is normally enclosed in single quotes. If the command contains no arguments, the quotes are not necessary; if it contains arguments separated by spaces, or multiple commands separated by semi-colons, then the quotes are essential. Double quotes are also permissible, but, as you will see later in this article, you have to be particularly careful with the syntax otherwise you might not execute the command you would like.

After the *command* has finished, the script will normally resume execution at the point at which it received the signal, although what it actually does is dependent both on the *command* itself, which may cause the script to exit, and on any commands running at the time the signal was received, which may or may not themselves abort.

You can have any number of **trap** commands in a script, but you should be aware that, if you have two or more traps for the same signal in the body of the script, then only the last one will be executed. For example, if you have the following two traps in your script:

```
trap 'rm tmpfile; exit' 2
.
.
trap 'print tmpfile removed' INT
```

then only the second of these commands will be executed and **tmpfile** itself will not be removed.

You should also be aware that, if you have traps in your script for both the **INT** and **ERR** signals, and the script receives an interrupt signal, then the commands associated with both of the traps will be executed.

IGNORING SIGNALS

There will be times when your scripts receive particular signals and you want to ignore them.

For example, you may have written a script to gather performance statistics over a 24-hour period, but you want to be able to log off after starting the script in the background. Under normal circumstances logging off will send the **HUP** signal to your script and kill it. You could run the script either using the **nohup** command, which would continue running when you logged off and place all output in the **nohup.out** file, unless otherwise redirected from within the script, or you could use a **trap** statement to get your script to ignore the **HUP** signal.

To get your script to ignore a signal, you use a null string, **''** or **" "**, with the **trap** command. In the example above you would use:

```
trap ' ' HUP
```

Ignoring the interrupt signal may cause you problems should your script contain errors and never finish. If this happens you can suspend the script with **Ctrl Z** and then kill off the suspended job.

TRAPPING SIGNALS IN FUNCTIONS

In much the same way that functions have arguments, which may be unknown to the surrounding code (**$1** for the script may be different to **$1** for a function), **trap** statements can be local to a function and unknown outside of it. This can allow you to control a function's behaviour separately from the main body of your code.

If you have a script which contains improbable code such as:

```
f_trap()
{
      trap 'print Ctrl C caught; return' INT
      sleep 1Ø
}

trap 'Now exiting' INT
f_trap
print Continuing ...
sleep 1Ø
```

then on first entering **Ctrl C** your script will print the function **trap** message, abort the first **sleep** command, return to the main body of the program, and print *Continuing ...*. The next time you enter **Ctrl** C it will print the *Now exiting* message, abort the last **sleep** command, and then exit from the script.

Within our **f_trap** function we have added a **return** statement to the trap to ensure that we return to the body of the script immediately. In this particular case the command was not absolutely essential since we would have exited the function after the **sleep 10** had completed, but if you have some endless looping construct in a function you must have a way of returning to the body of the script (or exiting completely) when **Ctrl C** is pressed.

RESETTING TRAPS

Many scripts have insufficient complexity to justify catching and ignoring signals, let alone resetting them. Small scripts rarely have a requirement for traps, which are most often reserved for large scripts, often run by many users, and which need to be made as resilient as possible.

When traps are used in scripts, they are quite often used to ensure that a particular piece of code runs without being interrupted. Once this usually small section of the script has completed, the traps can be reset so that the signal action reverts to its default.

To reset a signal to its default action we use a dash (**-**) as the command argument. You can try the following to see how it works:

```
$ vi traptest

trap '' 1 2
print 1st sleep
sleep 1Ø

trap '-' 1 2
print 2nd sleep
sleep 1Ø
```

If you press **Ctrl C** after the first message it will be ignored. When you press it after the second message, the trap has already been reset and the interrupt signal is now sent to the shell process that is running the script, and to the **sleep** process; the **sleep** process is killed by the signal and the script immediately exits.

MODIFYING THE MAILTO SCRIPT

When you use **trap** to catch a signal, you will usually want your script to perform some kind of clean-up activity and then die. People expect programs to die when they press **Ctrl C**, and the usual reason for catching a signal is to allow the program to do something before it completely finishes; the **mailto** program is no exception since we would like to remove the temporary file that we created to contain our message before exiting.

You could add a **trap** statement to **mailto** with the following line:

```
trap 'rm -f $TMPDIR/$MEMOFILE; exit' 1 2 15
```

We can now see the significance of using single quotes to surround the commands to be executed. If a **trap** statement contains variable names, like those shown above, we want to be sure that the string isn't evaluated until it needs to be run. The single quotes ensure that any variable will be expanded only at the time of execution so that we can be certain it contains the correct value.

For example, if the command to be executed contained the **$PWD** string and we had used double quotes, then the variable would have been expanded immediately and the current directory would have been inserted into our command string. If during our script we changed to another directory, then the **trap** command, when finally executed, would contain the wrong **PWD** value.

Many scripts call a function to perform their clean-up activities, and this is preferable if there are many commands to run before exiting. A function has the advantage that you can add further commands to it easily, it looks neater, and you don't have to quote it, whereas placing multiple commands within the quotes can be cumbersome and less easy to follow.

The preferred method of modifying **mailto** is to use the function approach so that the script now looks like the following (comment lines have been removed and changes are shown in italics):

```
$ vi mailto

#!/bin/ksh
RECIPIENT=$1
LOG=logfile_$RECIPIENT
LOGDIR=/usr/local/log
TMPDIR=/var/tmp
MEMOFILE=memo_$$

f_cleanup()
{
rm -f $TMPDIR/$MEMOFILE
exit 1
}
```

```
trap f_cleanup 1 2 15
print 'Subject: \c'
read subject

print "Subject: $subject" > $TMPDIR/$MEMOFILE
print 'Enter message and end with ^D on a blank line'
cat >> $TMPDIR/$MEMOFILE

date >> $LOGDIR/$LOG
cat $TMPDIR/$MEMOFILE >> $LOGDIR/$LOG
print '\n' >> $LOGDIR/$LOG

mail $RECIPIENT < $TMPDIR/$MEMOFILE
rm $TMPDIR/$MEMOFILE
```

Instead of being immediately killed by the **hangup**, **interrupt**, or **terminate** signal, **mailto** will now execute the **f_cleanup** function and remove the temporary file it has created. The **-f** option keeps **rm** from printing an error message if the temporary file has not yet been created.

COPROCESSES AND COROUTINES

When two or more processes are programmed to be executed simultaneously they are called coprocesses or coroutines – they may communicate with each other, or they may run independently. A pipeline is one example of a coprocess, but let us now consider coprocesses started from within shell scripts.

Within a script you can start one or more commands in the background, which under most circumstances would run completely independently of each other, and also of the script itself. There may be performance advantages in running multiple background commands, particularly when they use different resources. For example, one may be I/O intensive, and a second CPU intensive, or you may have two I/O intensive programs which access different disks. Normally your script would continue with its own processing, irrespective of what the background commands were doing.

However, each time you start a background process, you can never be certain when it is going to finish, and if the successful completion of the calling script is dependent upon the processing that this other command is doing, you should not run the background command in this manner; you will later see how we can partially get round this using the **wait** command.

There may be occasions when you want a background command to communicate with its calling script after it has completed whatever it is doing. To do so we must define it as a coprocess by placing the **|&** operator after the command in the calling script. This ensures that standard input of the coprocess is received from the calling script and its standard output is piped to it. Coprocesses can be other scripts, or they can be functions called from within the script.

A coprocess must satisfy the following criteria:

- There must be a newline character at the end of each output message.
- Each output message must be sent to standard output; commands within the coprocess can send their output to files if required. The standard output must be cleared after each message.

The following simple example shows how input can be passed to, and returned from, a coprocess called from within a script:

```
$ vi callcoproc

print "The calling script"
coproc |&
read -p a1 b1 c1 d1
print "Reading from the coprocess: $a1 $b1 $c1 $d1"

print -p "Passed to the coprocess"
read -p a2 b2 c2 d2
print "Passed back from the coprocess: $a2 $b2 $c2 $d2"

$ vi coproc

print "The coprocess is running"
read a b c d
print $a $b $c $d
```

When you run the **callcoproc** script the following output will be displayed:


```
The calling script
Reading from the coprocess: The coprocess is running
Passed back from the coprocess: Passed to the coprocess
```

The **print -p** command lets you send output to a coprocess and the corresponding **read -p** gets input from one. In the above example, this is what happens when you execute **callcoproc**:

1 The message, ‘The calling script’, is sent to the standard output of **callcoproc** and is printed on the screen.

2 It then runs **coproc |&** to start the coprocess script, executes the command **read -p a1 b1 c1 d1**, and awaits input from **coproc**.

3 On starting, **coproc** immediately sends the ‘The coprocess is running’ message to its standard output, and **callcoproc** assigns the words from the message to the variables **a1**, **b1**, **c1** and **d1**.

   You should be aware that if a coprocess sends its output to standard output and there is no corresponding **read -p** in the calling script, then this output is effectively lost.

4 **callcoproc** then prints the ‘Reading from the coprocess: The coprocess is running’ message on the screen.

5 In the meantime **coproc** has executed the **read a b c d** command and is waiting for standard input.

6 **callcoproc** now sends the message, ‘Passed to the coprocess’, to **coproc** via the **print -p** command, then executes the **read -p a2 b2 c2 d2** command and waits for standard input to be sent back to it from **coproc**.

7 **coproc** reads the message sent by **callcoproc** and assigns the words of ‘Passed to the coprocess’ to the variables **a**, **b**, **c** and **d**.

8 **coproc** then prints the variables using **print $a $b $c $d** and **callcoproc** reads the words from this message into the variables **a2**, **b2**, **c2** and **d2**.

9 Finally, **callcoproc** prints the message ‘Passed back from the coprocess: Passed to the coprocess’.

The above script is not particularly useful, but merely shows how you can get scripts to interact with each other. A more realistic situation would be to pass some parameters to a coprocess to enable it to perform a number of tasks and then pass back a message to the calling process when these had been completed.

THE WAIT COMMAND

As we mentioned earlier, there may be occasions when running multiple commands in the background from within a script can result in significant performance improvements. Consider a script which starts off multiple applications which are independent of each other, which often happens in an HACMP environment. The code within our script may look something like:

```
start_app1 &
start_app2 &
start_app3 &
```

If any further processing within our script is independent of the starting of the applications, then the above approach is OK, but you must be aware that if your script finishes before all the applications have started you will get zombie processes. But what if further script processing requires all applications to be started before it can continue? We could try to get round this by using:

```
start_app1 &
start_app2 &
start_app3
```

which would be perfectly satisfactory provided that we knew that **start_app3** finished after the other applications had started. But what if the start times for the three applications were all similar and we could never be sure which would finish first, which could easily happen where databases are involved and a database clean-up was required?

The solution is to use the **wait** command, which waits until all background jobs have been completed before continuing with the remaining commands in the script. The code would look like:

```
start_app1 &
start_app2 &
start_app3 &
wait
```

The **wait** command can take one or more PIDs as arguments so that you can wait for the completion of specific processes; but without arguments it waits until all processes known to the invoking shell have completed. If one of the processes currently invoked does not complete because of some error, then no further processing within your script will continue until you kill off the rogue process.

Tonto Kowalski
Guru (UAE)

Source code control system – part 2

This month we conclude the code for a Source Code Control System (SCCS).

```
######################################################################
#  Name     : CheckOutSpecificSourceFileForUpdate  ( )
#  Overview : The function checks out a specific version of a source
#             file for update.
#  Notes    :
######################################################################
CheckOutSpecificSourceFileForUpdate ()
{
#
if ! GetSourceFileName "CO"
then
    return $FALSE
fi
#
if ! GetDirectoryName
then
    return $FALSE
```

```
fi
# get version number
while  true
do
   clear
   echo "Enter the version number( l for list of values )"
   echo "(a to abandon):\c"
   read  RELEASE_ID
   case  $RELEASE_ID in
      "" ) DisplayMessage E "${INVALID_ENTRY}" ;;
        a) return  $FALSE  ;;
        l) DisplayListOfValues "V"  ;
           if [ "${SELECTED_VALUE}" = ""  ]
           then
                 : ;
           else
                RELEASE_ID="${SELECTED_VALUE}" ;
                break ;
           fi ;;
       * ) break    ;;
   esac
done
# remove the file to be checked out from target directory
rm  -f ${DIR_NAME}/${SOURCE_FILE_NAME}
#
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} edit -r${RELEASE_ID} \
    -p ${SOURCE_FILE_NAME} 1> ${DIR_NAME}/${SOURCE_FILE_NAME} 2>
${TEMP_FILE_1}
#
if  [  $?  -ne  Ø  ]
then
   DisplayMessage E "${EDIT_CHKOUT_FAILED}" N
   ERR_MSG=`cat  ${TEMP_FILE_1}`
   DisplayMessage E "${OS_ERROR}"
   #
   if [ "${DEBUG}"  = "${TRUE}"   ]
   then
        view ${TEMP_FILE_1}
   fi
   #
   return $FALSE
else
   return  $TRUE
fi
}
#####################################################################
#  Name     : RemoveLatestDelta
#  Overview : The function removes the latest delta for a specific
#             source.
#  Notes    :
```



```
##########################################################################
RemoveLatestDelta ()
{
#
if ! GetSourceFileName "CO"
then
    return $FALSE
fi
# get the latest release id
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} prt -y ${SOURCE_FILE_NAME}  \
                                                  > ${TEMP_FILE_1} 2>&1
if  [  $?  -ne  Ø  ]
then
   DisplayMessage E "${SID_NOT_RETRIEVED}" N
   ERR_MSG=`cat  ${TEMP_FILE_1}`
   DisplayMessage E "${OS_ERROR}"
   #
   if [ "${DEBUG}"  = "${TRUE}"   ]
   then
        view ${TEMP_FILE_1}
   fi
   #
   return $FALSE
fi
#
RELEASE_ID='cat ${TEMP_FILE_1} | awk {'print $3'}'
#
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} rmdel -r${RELEASE_ID} \
                                   ${SOURCE_FILE_NAME} > ${TEMP_FILE_1}
2>&1
#
if  [  $?  -ne  Ø  ]
then
   DisplayMessage E "${DELTA_NOT_REMOVED}" N
   ERR_MSG=`cat  ${TEMP_FILE_1}`
   DisplayMessage E "${OS_ERROR}"
   #
   if [ "${DEBUG}"  = "${TRUE}"   ]
   then
        view ${TEMP_FILE_1}
   fi
   #
   return $FALSE
else
   return $TRUE
fi
}
##########################################################################
#  Name     : GetReadOnlyLatestSourceFile
#  Overview : The function checks out a read-only copy of the latest
```

```
#             source file.
#  Notes    :
######################################################################
GetReadOnlyLatestSourceFile  ()
{
if ! GetSourceFileName "CO"
then
    return $FALSE
fi
#
if ! GetDirectoryName
then
    return $FALSE
fi
# checkout the source
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} get -p  ${SOURCE_FILE_NAME} \
                         1> ${DIR_NAME}/${SOURCE_FILE_NAME} 2>
${TEMP_FILE_1}
if  [  $?  -ne  Ø  ]
then
   DisplayMessage E "${READ_CHKOUT_FAILED}" N
   ERR_MSG='cat  ${TEMP_FILE_1}'
   DisplayMessage E "${OS_ERROR}"
   #
   if [ "${DEBUG}"  = "${TRUE}"   ]
   then
        view ${TEMP_FILE_1}
   fi
   #
   return $FALSE
else
   return $TRUE
fi
#
}
######################################################################
#  Name     : GetReadOnlySpecificVersionOfSourceFile
#  Overview : The function checks out a read-only copy of a specific
#             version of the source file.
#  Notes    :
######################################################################
GetReadOnlySpecificVersionOfSourceFile  ( )
{
#
if ! GetSourceFileName "CO"
then
    return $FALSE
fi
#
if ! GetDirectoryName
```

```
then
    return $FALSE
fi
#  get version number
while  true
do
   clear
   echo "Enter the version number( l for list of values )"
   echo "(a to abandon):\c"
   read  RELEASE_ID
   case  $RELEASE_ID in
      "" ) DisplayMessage E "${INVALID_ENTRY}" ;;
        a) return  $FALSE  ;;
        l) DisplayListOfValues "V"  ;
           if [ "${SELECTED_VALUE}" = ""  ]
           then
                 : ;
           else
                RELEASE_ID="${SELECTED_VALUE}" ;
                break ;
           fi ;;
       * ) break    ;;
   esac
done
#
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} get -r ${RELEASE_ID} -p \
      ${SOURCE_FILE_NAME} 1> ${DIR_NAME}/${SOURCE_FILE_NAME} 2>
${TEMP_FILE_1}
if  [  $? -ne  Ø  ]
then
     DisplayMessage E "${READ_CHKOUT_FAILED}" N
     ERR_MSG='cat  ${TEMP_FILE_1}'
     DisplayMessage E "${OS_ERROR}"
     if [ "${DEBUG}"  = "${TRUE}"   ]
     then
        view ${TEMP_FILE_1}
     fi
     #
     return $FALSE
else
     return $TRUE
fi
}
######################################################################
#  Name     : CheckInNewSourceFile
#  Overview : The function checks in a new source file.
#  Notes    :
######################################################################
CheckInNewSourceFile  ( )
{
```

```
#
if ! GetSourceFileName "CI"
then
    return $FALSE
fi
#
CHKEDIN_FILE_COPY=",${SOURCE_FILE_NAME}"
# insert SCCS control keywords into source file
if ! InsertSCCSKeyWordsIntoFile
then
     return $FALSE
fi
#
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} create  ${SOURCE_FILE_NAME} \
                                                  > ${TEMP_FILE_1}  2>&1
if  [  $?  -ne  Ø  ]
then
    DisplayMessage E "${NEW_CHKIN_FAILED}" N
    ERR_MSG='cat  ${TEMP_FILE_1}'
    DisplayMessage E "${OS_ERROR}"
    if [ "${DEBUG}"  = "${TRUE}"   ]
    then
        view ${TEMP_FILE_1}
    fi
    #
    return $FALSE
else
   # remove copy of checked in file
   rm -f  ${CHKEDIN_FILE_COPY}
   return $TRUE
fi
}
######################################################################
#  Name     : ShowListOfCheckedOutFiles
#  Overview : The function displays a list of all checked out source
#             files.
#  Notes    :
######################################################################
ShowListOfCheckedOutFiles ( )
{
DATETIME='date "+%d/%m/%Y at %H:%M:%S"'
#
HEADER="List of Checked Out Source Files on ${DATETIME}"
FormatUnderscores  "${HEADER}"
echo "  ${HEADER}"        >   ${TEMP_FILE_1}
echo "  ${UNDERSCORE}"   >>   ${TEMP_FILE_1}
#
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} info   > ${TEMP_FILE_2} 2>&1
if  [  $?  -ne  Ø  ]
then
```

```
    DisplayMessage E "${CHKOUT_LIST_FAILED}" N
    ERR_MSG='cat  ${TEMP_FILE_2}'
    DisplayMessage E "${OS_ERROR}"
    #
    if [ "${DEBUG}"  = "${TRUE}"   ]
    then
        view ${TEMP_FILE_1}
    fi
    #
    return $FALSE
elif  [ ! -s  ${TEMP_FILE_2}   ]
then
    DisplayMessage E "${NO_CHKOUT_LIST}"
    return $TRUE
else
   cat ${TEMP_FILE_2}  >> ${TEMP_FILE_1}
   view ${TEMP_FILE_1}
   return $TRUE
fi
}
######################################################################
#  Name     : ReleaseCheckedOutSourceFile
#  Overview : The function releases the checked out source file.
#  Notes    :
######################################################################
ReleaseCheckedOutSourceFile  ()
{
#
if ! GetSourceFileName "CO"
then
    return $FALSE
fi
#
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} unedit ${SOURCE_FILE_NAME} >  \
                                                          ${TEMP_FILE_1}
2>&1
if  [  $?  -ne  Ø  ]
then
    DisplayMessage E "${RELEASE_LOCK_FAILED}" N
    ERR_MSG='cat  ${TEMP_FILE_1}'
    DisplayMessage E "${OS_ERROR}"
    #
    if [ "${DEBUG}"  = "${TRUE}"   ]
    then
        view ${TEMP_FILE_1}
    fi
    #
    return $FALSE
else
    return $TRUE
```

```
fi
#
}
######################################################################
#  Name     : ShowSourceReleaseHistoryIncludingBrances
#  Overview : The function shows the release history for a specific
#             source.
#  Notes    :
######################################################################
ShowSourceReleaseHistory  ()
{
#
if ! GetSourceFileName "CO"
then
    return $FALSE
fi
#
DATETIME='date "+%d/%m/%Y at %H:%M:%S"'
#
HEADER="Release History for ${SOURCE_FILE_NAME} on ${DATETIME}"
FormatUnderscores  "${HEADER}"
echo "  ${HEADER}"        >   ${TEMP_FILE_1}
echo "  ${UNDERSCORE}"   >>   ${TEMP_FILE_1}
#
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} prs -l -r1.1 ${SOURCE_FILE_NAME}
>  \
                                                    ${TEMP_FILE_2}  2>&1
if  [  $?  -ne  Ø  ]
then
    DisplayMessage E "${HIST_LIST_FAILED}" N
    ERR_MSG='cat  ${TEMP_FILE_2}'
    DisplayMessage E "${OS_ERROR}"
    if [ "${DEBUG}"  = "${TRUE}"   ]
    then
        view ${TEMP_FILE_2}
    fi
    #
    return $FALSE
else
   cat ${TEMP_FILE_2}  >>  ${TEMP_FILE_1}
   view ${TEMP_FILE_1}
   return $TRUE
fi
#
}
######################################################################
#  Name     : UpdateDeltaComment
#  Overview : The function updates comments associated with a
#             specific delta.
#  Notes    :
```

```
########################################################################
UpdateDeltaComment ()
{
#
if ! GetSourceFileName "CO"
then
    return $FALSE
fi
#  get version number to update comment for
if ! GetReleaseId
then
    return $FALSE
fi
# get new comment
while  true
do
   clear
   echo "Enter the new comment(a to abandon):\c"
   read  COMMENT
   case  ${COMMENT} in
      "" ) DisplayMessage E "${INVALID_ENTRY}" ;;
        a) return  $FALSE  ;;
        *) break ;;
   esac
done
#
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} cdc -r${RELEASE_ID} -
y"${COMMENT}" \
                            ${SOURCE_FILE_NAME} > ${TEMP_FILE_1}  2>&1
if  [  $?  -ne  Ø  ]
then
    DisplayMessage E "${COMMENT_UPDATE_FAILED}" N
    ERR_MSG='cat  ${TEMP_FILE_1}'
    DisplayMessage E "${OS_ERROR}"
    if [ "${DEBUG}"  = "${TRUE}"   ]
    then
        view ${TEMP_FILE_1}
    fi
    #
    return $FALSE
else
    return $TRUE
fi
}
########################################################################
#  Name     : ShowVersionDifference
#  Overview : The function compares two deltas and reports the
#             difference.
#  Notes    :
########################################################################
```

```
ShowVersionDifference ()
{
#
if ! GetSourceFileName "CO"
then
    return $FALSE
fi
#  get first version number
if ! GetReleaseId
then
    return $FALSE
fi
#
RELEASE_ID_1="${RELEASE_ID}"
#  get second version number
if ! GetReleaseId
then
    return $FALSE
fi
#
RELEASE_ID_2="${RELEASE_ID}"
#
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} sccsdiff  \
          -r${RELEASE_ID_1} -r${RELEASE_ID_2} -p ${SOURCE_FILE_NAME} \
                                                 > ${TEMP_FILE_1}  2>&1
#
if  [  $?  -ne  Ø  ]
then
    DisplayMessage E "${VERSION_DIFF_FAILED}" N
    ERR_MSG='cat  ${TEMP_FILE_1}'
    DisplayMessage E "${OS_ERROR}"
    if [ "${DEBUG}"  = "${TRUE}"   ]
    then
        view ${TEMP_FILE_1}
    fi
    #
    return $FALSE
fi
#
DATETIME='date "+%d/%m/%Y at %H:%M:%S"'
HEADER="Difference Between Release ${RELEASE_ID_1} and ${RELEASE_ID_2}
for ${SOURCE_FILE_NAME} on ${DATETIME}"
FormatUnderscores  "${HEADER}"
echo "  ${HEADER}"        >  ${TEMP_FILE_2}
echo "  ${UNDERSCORE}"   >>  ${TEMP_FILE_2}
cat   ${TEMP_FILE_1}     >>  ${TEMP_FILE_2}
view ${TEMP_FILE_2}
#
return $TRUE
}
```

```
########################################################################
#  Name     : ShowChangesMade
#  Overview : The function compares the checked out source file with
#             the latest delta and reports the difference.
#  Notes    :
########################################################################
ShowChangesMade ( )
{
#
if ! GetSourceFileName "CO"
then
    return $FALSE
fi
#
${SCCS_BIN_DIR}/sccs -d${SCCS_ROOT_DIR} diffs  \
                    -p ${SOURCE_FILE_NAME} > ${TEMP_FILE_1}  2>&1
#
if  [  $?  -ne  Ø  ]
then
    DisplayMessage E "${CHANGE_DIFF_FAILED}" N
    ERR_MSG='cat  ${TEMP_FILE_1}'
    DisplayMessage E "${OS_ERROR}"
    if [ "${DEBUG}"  = "${TRUE}"   ]
    then
        view ${TEMP_FILE_1}
    fi
    #
    return $FALSE
fi
#
DATETIME='date "+%d/%m/%Y at %H:%M:%S"'
HEADER="Difference Between Checked Out Version and Latest Delta"
FormatUnderscores  "${HEADER}"
echo "     ${HEADER}"        >  ${TEMP_FILE_2}
echo "     ${UNDERSCORE}"   >>  ${TEMP_FILE_2}
HEADER="for ${SOURCE_FILE_NAME} on ${DATETIME}"
FormatUnderscores  "${HEADER}"
echo "     ${HEADER}"       >>  ${TEMP_FILE_2}
echo "     ${UNDERSCORE}"   >>  ${TEMP_FILE_2}
cat   ${TEMP_FILE_1}     >>  ${TEMP_FILE_2}
view ${TEMP_FILE_2}
#
return $TRUE
}
########################################################################
#  Name     : ProcessOption
#  Overview : The function processes the selected option.
#  Notes    :
########################################################################
ProcessOption  ()
```



```
{
case  $OPTION  in
       5)   CheckInNewSourceFile ;;
      10)   CheckOutLatestSourceFileForUpdate    ;;
      15)   CheckOutSpecificSourceFileForUpdate    ;;
      20)   CheckInUpdatedSourceFile  ;;
      25)   GetReadOnlyLatestSourceFile    ;;
      30)   GetReadOnlySpecificVersionOfSourceFile    ;;
      35)   ShowListOfCheckedOutFiles  ;;
      40)   ReleaseCheckedOutSourceFile  ;;
      45)   ShowSourceReleaseHistory  ;;
      50)   UpdateDeltaComment  ;;
      55)   RemoveLatestDelta ;;
      60)   ShowVersionDifference ;;
      65)   ShowChangesMade ;;
      99)   ProcessExit  $SEC ;;
      * ) DisplayMessage E "${INVALID_ENTRY}" ;;
esac
}
#####################################################################
#  Name     : ParseCommandLine
#  Overview : The function parses the command line which is assigned
#             to a variable ${ARGV}.
#  Notes    : 1. Only one argument (-D ) is expected. If the option
#                is provided, $DEBUG is set to $TRUE.
#####################################################################
ParseCommandLine ()
{
if  [  "${ARGV}"  =  "-D"  ]
then
     DEBUG="${TRUE}"
     DisplayMessage I "${DEBUG_SET}" N
fi
}
#####################################################################
#  Name     : main
#  Overview : The function implements processing structure.
#  Notes    :
#####################################################################
main ()
{
InitialiseVariable
#
ParseCommandLine
#
while  true
do
  DisplayMenu
  ProcessOption
done
```

```
}
# invoke main ()
# define traps
# trap "HandleInterrupt " $SIGINT  $SIGTERM $SIGTSTP
trap "HandleInterrupt "     2        15      18
# package command  line
ARGC="$#"
ARGV="$@"
main
```

Arif Zaman
ETL Developer (UK)

Articles for inclusion in *AIX Update* can be sent to the editor, Trevor Eddolls, at trevore@xephon.com. A copy of our *Notes for Contributors* can be downloaded from www.xephon.com/nfc.

AIX news

MySQL has announced that its open source MySQL database has enhanced support for AIX.

The MySQL database server architecture promotes extensive re-use of pieces of code within the software.

This version adds to support for all major Linux distributions as well as Unix, Mac OS X, and Windows operating systems.

For further information contact:
MySQL AB, Bangårdsgatan 8, S-753 20 Uppsala, Sweden.
Tel: +46 18 10 18 90.
URL: http://www.mysql.com/products/index.html.

* * *

CONNX Solutions has announced Version 8.8 of its CONNX data access middleware, now with a range of performance and feature enhancements. Support for VSAM VSE data sources, which provides real-time high-performance access to VSAM files under CICS partitions on the VSE operating system, has also been included in the release. Also, direct support for Microsoft .NET technology has been added with the introduction of a pure CONNX OLE DB Provider. Users, says the vendor, will be able to achieve the performance of a native provider while writing their own applications in managed C# or VB .NET code.

Support for C-ISAM databases, which was included in the CONNX 8.7 release, has also been expanded to include Solaris 5.6 and above as well as AIX 4.1 and above. Microfocus COBOL C-ISAM support has also been enhanced.

The software provides read/write real-time access to all enterprise data from any platform as if all the data existed in one relational database. All data is then accessible using standard SQL and any standards-based application.

It acts as a reusable data access framework, supporting C-ISAM, VSAM, DB2, Oracle, RMS, RDB, PostGreSQL, DBMS, Dataflex, POWERflex, SQL Server, Sybase, and Informix and any OLE DB, ODBC, or JDBC data source.

For further information contact:
CONNX Solutions, 1800 112th Avenue NE, Suite #150, Bellevue, WA 98004, USA.
Tel: (425) 519 6600.
URL: http://www.connx.com/products/products.html.

* * *

Serena Software has integrated its ChangeMan DS software change manager for distributed systems with the TeamTrack defect and issue management system from TeamShare.

The combination, we're told, provides TeamTrack users with an automated change management system that helps streamlines software development and improves communication across the enterprise. In addition, the integration also allows joint customers to integrate with other vendors' tools.

ChangeMan DS provides native support across AIX, HP-UX, Solaris, OS/390 USS, and SCO, HP NonStop servers, Linux, MPE/ix, OS/400, and Windows.

For further information contact:
Serena Software, 2755 Campus Drive, 3rd Floor. San Mateo, CA 94403, USA.
Tel: (650) 522 6600
URL: http://www.serena.com.