95

AIX

September 2003

update

In this issue

3 Get rid of confusing error messages

5 Shell script library – part 2

20 The sed command

33 Automated mechanism for changing passwords

50 AIX news

AIX Update

Published by

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

North American office

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

Subscriptions and back-issues

A year's subscription to *AIX Update*, comprising twelve monthly issues, costs £180.00 in the UK; $275.00 in the USA and Canada; £186.00 in Europe; £192.00 in Australasia and Japan; and £190.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the November 1999 issue, are available separately to subscribers for £16.00 ($24.00) each including postage.

AIX Update on-line

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

Editor

Trevor Eddolls

Disclaimer

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

Contributions

When Xephon is given copyright, articles published in *AIX Update* are paid for at the rate of £100 ($160) per 1000 words and £50 ($80) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £20 ($32) per 100 lines. To find out more about contributing an article, without any obligation, please download a copy of our *Notes for Contributors* from www.xephon.com/nfc.

Printed in England.

Get rid of confusing error messages

At some AIX installations, you can observe curious error messages in the **errpt** output after a reboot or if you start the configuration manager, cfgmgr.

If you analyse the occurrences, ie where it happens and where it doesn't, you can find out why it happens.

On all systems where no graphic adapter is installed or onboard and where no additional configuration steps have been done, the error messages look like this:

```
IDENTIFIER TIMESTAMP  T C RESOURCE_NAME  DESCRIPTION
E85C5C4C   0624200303 P S CFGLFT         SOFTWARE PROGRAM ERROR
```

or:

```
E85C5C4C   0626183703 P S STARTLFT       SOFTWARE PROGRAM ERROR
```

A reduced view of **errpt -a** looks like:

```
LABEL:          GRAPHICS
IDENTIFIER:     E85C5C4C
Class:          S
Type:           PERM
Resource Name:  CFGLFT
Description
SOFTWARE PROGRAM ERROR
Probable Causes
SOFTWARE PROGRAM
Failure Causes
SOFTWARE PROGRAM
        Recommended Actions
        IF PROBLEM CONTINUES TO OCCUR REPEATEDLY THEN DO THE FOLLOWING
        CONTACT APPROPRIATE SERVICE REPRESENTATIVE
        REPORT DETAILED DATA
Detail Data
DETECTED   FAILED     RC   ERROR LOCATION
  cfglft   build_dds  32   1114  31
```

A reduced view of **errpt -a -N STARTLFT** looks like:

```
LABEL:          GRAPHICS
IDENTIFIER:     E85C5C4C
Class:          S
Type:           PERM
```

```
Resource Name:  STARTLFT
```

The output is the same as with the other error message:

```
Detail Data
DETECTED   FAILED     RC   ERROR LOCATION
startlft               Ø   12Ø4  6
```

The procedure to overcome this problem is as follows. It should be applied only to systems where no graphic adapter is installed or in use.

Make a system back-up:

```
smit backsys
```

lscons should point to an available tty like */dev/tty0*. If it doesn't, use **smit console** to change it.

Next, use **inittab**:

```
cat /etc/inittab | grep dt
```

Comment statement like:

```
dt:2:wait:/etc/rc.dt
```

Next, check which graphic devices are present with:

```
lsdev -C | egrep '(lft|rcm|sioma|gxme)'
```

Remove them (otherwise you get problems at the next step):

```
rmdev -dl lftØ
rmdev -dl rcmØ
rmdev -dl siomaØ
rmdev -dl gxmeØ
```

Finally, check whether the graphics fileset is installed:

```
lslpp -l devices.graphics.com
```

Then check its dependants with:

```
lslpp -d devices.graphics.com
```

The output looks like this:

```
Fileset               Dependants
-----------------------------------------
<Fileset> is a requisite of <Dependents>
```

```
Path: /usr/lib/objrepos
devices.graphics.com 4.3.3.75
                    devices.pci.3353cØ88.com 4.3.3.Ø

Path: /etc/objrepos
devices.graphics.com 4.3.3.75
                    NONE


lslpp -l devices.pci.3353cØ88.com


Fileset                      Level  State      Description
----------------------------------------------------------
Path: /usr/lib/objrepos
devices.pci.3353cØ88.com  4.3.3.25  COMMITTED  E15 Graphics Adapter
                                               Family
                                               Common Software
```

Now you can choose to remove the graphics device fileset and its dependants with one command:

```
installp -ug devices.graphics.com
```

Clear the errorlog with:

```
errclear -N CFGLFT,STARTLFT  Ø
```

Issue **cfgmgr** and afterwards check **errpt** again.

This will get rid of confusing error messages.

Imhotep
Unix System Administrator (Austria)

Shell script library – part 2

This month we conclude the code for a shell script that uses a library containing predefined function definitions.

```
# consider allowed characters
if [ "${NEXT_CHAR}" != "Ø" -a  \
     "${NEXT_CHAR}" != "1" -a "${NEXT_CHAR}"   != "2" -a  \
     "${NEXT_CHAR}" != "3" -a "${NEXT_CHAR}"   != "4" -a  \
     "${NEXT_CHAR}" != "5" -a "${NEXT_CHAR}"   != "6" -a  \
```

```
          "${NEXT_CHAR}" != "7" -a "${NEXT_CHAR}"   != "8" -a  \
          "${NEXT_CHAR}" != "9"  ]
     then
         if [ "${DEBUG}" = "${TRUE}"  ]
         then
              DisplayMessage E "${INVALID_CHAR_IN_STRING}"
         fi
         return $FALSE
     fi
  fi
  COLUMN_POS='expr $COLUMN_POS + 1'
done
# validate for minus inteher
if  [ "${NEGATIVE_INTEGER}"  = "${TRUE}"  ]
then
    # minimum length must be two
    if [ 'echo "${P_STRING}\c" | wc  -c'  -lt 2  ]
    then
        if [ "${DEBUG}" = "${TRUE}"  ]
        then
              DisplayMessage E "${INVALID_LEN_FOR_NEGATIVE_INT}"
        fi
        return $FALSE
    fi
fi
FUNCTION=""
return $TRUE
}
################################################################################
#  Name     : CheckDateFormat
#  Overview : The function validates a string for a date with one of the
#             following date formats:
#                      - YYYYMMDD
#                      - YYYYMM
#                      - DDMMYYYY
#                      - MMYYYY
#  Input    : String1  ( date to be checked )
#             String2  ( required Format    )
#  Returns  : TRUE   if string matches required format
#             FALSE  otherwise
#  Usage    : if !  CheckDateFormat "01012002"  "DDMMYYYY"
#             then
#                  echo  "Invalid date format"
#             fi
#  Notes    : 1.The function validates day, month, and year separately
#             and, therefore, it is possible that if day and month are
#             swapped the validation will still succeed (ie 01022003
#             02012003). The function does not guard against this kind
#             of transposition.
################################################################################
```

```
CheckDateFormat ()
{
# define function name
FUNCTION="${FUNCTION}:CheckDateFormat"
# assign parameters
if [ $#  -ne  2  ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${INVALID_NO_ARGS}"
    fi
    return $FALSE
fi
P_DATE_STRING="$1"
P_DATE_FORMAT="$2"
# validate against null
if  [ "${P_DATE_STRING}"  = "" ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${NULL_DATE_STRING}"
    fi
    return $FALSE
elif  [ "${P_DATE_FORMAT}"  = "" ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${NULL_FORMAT_STRING}"
    fi
    return $FALSE
fi
LEN_BEFORE_P_STRING='echo "${P_STRING}\c" | wc -c'
LEN_AFTER_P_STRING='echo "${P_STRING}\c" | tr -d "${META_CHARS}"  |\
                                                    wc -c'
if [ $LEN_BEFORE_P_STRING -ne $LEN_AFTER_P_STRING ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${META_CHARS_IN_STRING}"
    fi
    return $FALSE
fi
# validate for numeric
# save current $FUNCTION stack
PREV_FUNCTION="${FUNCTION}"
if  !  IsNumeric  "${P_DATE_STRING}"
then
     return $FALSE
fi
# restore $FUNCTION stack
```

```
FUNCTION="${PREV_FUNCTION}"
# validate for length
if [ "${P_DATE_FORMAT}" = "YYYYMMDD" -o "${P_DATE_FORMAT}" = "DDMMYYYY"
]
then
    REQ_DATE_STRING_LEN=8

elif [ "${P_DATE_FORMAT}" = "YYYYMM" -o "${P_DATE_FORMAT}" = "MMYYYY" ]
then
    REQ_DATE_STRING_LEN=6
else
    # wrong date format supplied
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${INVALID_DATE_FORMAT}"
    fi
    return $FALSE
fi
IN_DATE_STRING_LEN='echo "${P_DATE_STRING}\c" | wc -c'
if [ ${IN_DATE_STRING_LEN} -gt  ${REQ_DATE_STRING_LEN} ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${TOO_MANY_DIGITS}"
    fi
    return $FALSE
elif [ ${IN_DATE_STRING_LEN} -lt  ${REQ_DATE_STRING_LEN} ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${NOT_ENOUGH_DIGITS}"
    fi
    return $FALSE
fi
# extract day, month, and year
if [  "${P_DATE_FORMAT}" = "DDMMYYYY" ]
then
    DAY='echo   ${P_DATE_STRING}    | cut -c1-2'
    MONTH='echo ${P_DATE_STRING}    | cut -c3-4'
    YEAR='echo  ${P_DATE_STRING}    | cut -c5-8'
elif [ "${P_DATE_FORMAT}" = "MMYYYY" ]
then
    DAY=""
    MONTH='echo ${P_DATE_STRING}    | cut -c1-2'
    YEAR='echo  ${P_DATE_STRING}    | cut -c3-6'
elif [  "${P_DATE_FORMAT}" = "YYYYMMDD" ]
then
    DAY='echo   ${P_DATE_STRING}    | cut -c7-8'
    MONTH='echo ${P_DATE_STRING}    | cut -c5-6'
    YEAR='echo  ${P_DATE_STRING}    | cut -c1-4'
```

```
elif [  "${P_DATE_FORMAT}" = "YYYYMM" ]
then
    DAY=""
    MONTH='echo ${P_DATE_STRING}    | cut -c5-6'
    YEAR='echo  ${P_DATE_STRING}    | cut -c1-4'
fi
# establish  leap year
LEAP_YEAR=${FALSE}
# establish leap year
if [ 'expr ${YEAR} % 4' -eq Ø ]
then
     LEAP_YEAR=${TRUE}
fi
# validate month
if [ "${MONTH}" != "Ø1" -a "${MONTH}" != "Ø2" -a "${MONTH}" != "Ø3" -a \
     "${MONTH}" != "Ø4" -a "${MONTH}" != "Ø5" -a "${MONTH}" != "Ø6" -a \
     "${MONTH}" != "Ø7" -a "${MONTH}" != "Ø8" -a "${MONTH}" != "Ø9" -a \
     "${MONTH}" != "1Ø" -a "${MONTH}" != "11" -a "${MONTH}" != "12" ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${INVALID_MONTH}"
    fi
    return $FALSE
fi
# validate day of the month if required
if [ "${DAY}" !=  ""  ]
then
 if [ "${MONTH}" =  "Ø1" -o "${MONTH}" = "Ø3" -o "${MONTH}" = "Ø5"  -o \
      "${MONTH}" =  "Ø7" -o "${MONTH}" = "Ø8" -o "${MONTH}" = "1Ø" -o \
      "${MONTH}" =  "12" ]
   then
      if [[  $DAY -lt 1 || $DAY -gt 31 ]]
      then
         if [ "${DEBUG}" = "${TRUE}" ]
         then
            DisplayMessage E "${INVALID_DAY}"
         fi
         return $FALSE
      fi
 elif [ "${MONTH}" = "Ø4" -o "${MONTH}" = "Ø6" -o "${MONTH}" = "Ø9" -o \
        "${MONTH}" = "11" ]
   then
      if [[  $DAY -lt 1 || $DAY -gt 3Ø ]]
      then
         if [ "${DEBUG}" = "${TRUE}" ]
         then
            DisplayMessage E "${INVALID_DAY}"
         fi
         return $FALSE
```

```
      fi
   elif [ "${MONTH}" =  "Ø2" -a "${LEAP_YEAR}" = "${TRUE}" ]
   then
      if [[  $DAY -lt 1 || $DAY -gt 29 ]]
      then
         if [ "${DEBUG}" = "${TRUE}" ]
         then
            DisplayMessage E "${INVALID_DAY}"
         fi
         return $FALSE
      fi
   elif [ "${MONTH}" =  "Ø2" -a "${LEAP_YEAR}" = "${FALSE}" ]
   then
      if [[  $DAY -lt 1 || $DAY -gt 28 ]]
      then
         if [ "${DEBUG}" = "${TRUE}" ]
         then
            DisplayMessage E "${INVALID_DAY}"
         fi
         return $FALSE
      fi
  fi
fi
FUNCTION=""
return $TRUE
}
################################################################################
#  Name     : FormatDate
#  Overview : The function formats date according to required format.
#  Input    : string1 ( containing date )
#             string2 ( from format     )
#             string2 ( to format       )
#  Returns  : Formatted Date   if date is formatted correctly
#             NULL             otherwise
#  Usage:FORMATTED_DATE='FormatDate "Ø1Ø12ØØ2" "DDMMYYYY" "DD-MON-YYYY"'
#             if [ "${FORMATTED_DATE}" = ""  ]
#             then
#                  echo "Date formatting failed"
#             fi
#  Notes   : 1. When invoking this function, do not set the debug option
#               because any debug message displayed will be assigned to
#               the variable (as shown above).
################################################################################
FormatDate ()
{
# define function name
FUNCTION="${FUNCTION}:FormatDate"
# assign parameter
if [ $#  -ne  3  ]
then
```

```
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${INVALID_NO_ARGS}"
    fi
    return $FALSE
fi
P_DATE_STRING="$1"
P_FROM_DATE_FORMAT="$2"
P_TO_DATE_FORMAT="$3"
# initialize formatted date  with null
FORMATTED_DATE=""
#  define  from date format
FROM_DATE_FORMAT="DDMMYYYY YYYYMMDD"
# define to date  format
TO_DATE_FORMAT="DD-MON-YYYY DD/MM/YYYY YYYY/MM/DD"
# validate against null
if  [ "${P_FROM_DATE_FORMAT}"  = "" ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${NULL_FROM_DATE_FORMAT}"
    fi
    return "${FORMATTED_DATE}"
elif  [ "${P_TO_DATE_FORMAT}"  = "" ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${NULL_TO_DATE_FORMAT}"
    fi
    return "${FORMATTED_DATE}"
fi
# examine date string for meta characters
LEN_BEFORE_P_STRING='echo "${P_STRING}\c" | wc -c'
LEN_AFTER_P_STRING='echo "${P_STRING}\c" | tr -d "${META_CHARS}"  |\
                                                    wc -c'
if [ $LEN_BEFORE_P_STRING -ne $LEN_AFTER_P_STRING ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${META_CHARS_IN_STRING}"
    fi
    return "${FORMATTED_DATE}"
fi
# validate for numeric
if  !  IsNumeric  "${P_DATE_STRING}"
then
     return "${FORMATTED_DATE}"
fi
FUNCTION="${FUNCTION}:FormatDate"
# validate from date format
```

```
FORMAT_VALID="${FALSE}"
for FORMAT in ${FROM_DATE_FORMAT}
do
  if [  "${P_FROM_DATE_FORMAT}"  = "${FORMAT}" ]
  then
      FORMAT_VALID="${TRUE}"
      break
  fi
done
if [  "${FORMAT_VALID}" = "${FALSE}"  ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${INVALID_FROM_DATE_FORMAT}"
    fi
    return "${FORMATTED_DATE}"
fi
# validate to date format
FORMAT_VALID="${FALSE}"
for FORMAT in ${TO_DATE_FORMAT}
do
  if [  "${P_TO_DATE_FORMAT}"  = "${FORMAT}" ]
  then
      FORMAT_VALID="${TRUE}"
      break
  fi
done
if [  "${FORMAT_VALID}" = "${FALSE}"  ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${INVALID_TO_DATE_FORMAT}"
    fi
    return "${FORMATTED_DATE}"
fi
# check date and from date format
if ! CheckDateFormat  "${P_DATE_STRING}"  "${P_FROM_DATE_FORMAT}"
then
    return "${FORMATTED_DATE}"
fi
FUNCTION="${FUNCTION}:FormatDate"
# extract day , month and  year
if [  "${P_FROM_DATE_FORMAT}" = "DDMMYYYY" ]
then
    DAY='echo   ${P_DATE_STRING}    | cut -c1-2'
    MONTH='echo ${P_DATE_STRING}    | cut -c3-4'
    YEAR='echo  ${P_DATE_STRING}    | cut -c5-8'
elif [ "${P_FROM_DATE_FORMAT}" = "YYYYMMDD" ]
then
    DAY='echo   ${P_DATE_STRING}    | cut -c7-8'
```

```
    MONTH='echo ${P_DATE_STRING}    | cut -c5-6'
    YEAR='echo  ${P_DATE_STRING}    | cut -c1-4'
fi
# format  date
# reformat  $MONTH
if [ "${MONTH}"   = "Ø1" ]
then
     MON="JAN"
elif [ "${MONTH}" = "Ø2" ]
then
     MON="FEB"
elif [ "${MONTH}" = "Ø3" ]
then
     MON="MAR"
elif [ "${MONTH}" = "Ø4" ]
then
     MON="APR"
elif [ "${MONTH}" = "Ø5" ]
then
     MON="MAY"
elif [ "${MONTH}" = "Ø6" ]
then
     MON="JUN"
elif [ "${MONTH}" = "Ø7" ]
then
     MON="JUL"
elif [ "${MONTH}" = "Ø8" ]
then
     MON="AUG"
elif [ "${MONTH}" = "Ø9" ]
then
     MON="SEP"
elif [ "${MONTH}" = "1Ø" ]
then
     MON="OCT"
elif [ "${MONTH}" = "11" ]
then
     MON="NOV"
elif [ "${MONTH}" = "12" ]
then
     MON="DEC"
fi
if [  "${P_TO_DATE_FORMAT}"  = "DD-MON-YYYY" ]
then
     FORMATTED_DATE="${DAY}-${MON}-${YEAR}"
     echo "${FORMATTED_DATE}"
elif [  "${P_TO_DATE_FORMAT}"  = "DD/MM/YYYY" ]
then
     FORMATTED_DATE="${DAY}/${MONTH}/${YEAR}"
     echo "${FORMATTED_DATE}"
```

```
elif [  "${P_TO_DATE_FORMAT}"  = "YYYY/MM/DD" ]
then
     FORMATTED_DATE="${YEAR}/${MONTH}/${DAY}"
     echo "${FORMATTED_DATE}"
fi
FUNCTION=""
}
#############################################################################
#  Name     : Process31DaysMonths
#  Overview : The function processes days that fall in months
#             with 31 days.
#  Input    :
#  Returns  :
#  Notes    :
#############################################################################
Process31DaysMonths ()
{
if [  $DAY -gt 31 ]
then
    # no of days exceed 31 days
    # set $DAY  to  Ø1
    DAY=1
    # increment $MONTH by 1
    MONTH='expr $MONTH + 1'
    if [ $MONTH -gt 12  ]
    then
       # no of months exceeded 12
       # set $MONTH  to Ø1
       MONTH=1
       # increment $YEAR by 1
       YEAR='expr $YEAR + 1'
       # establish leap year
       if [ 'expr ${YEAR} % 4' -eq Ø ]
       then
           LEAP_YEAR=${TRUE}
       else
           LEAP_YEAR=${FALSE}
       fi
    fi
elif  [ $DAY  -eq  Ø  ]
then
    # we're taking away days from specified date
    # decrement $MONTH by 1
    MONTH='expr $MONTH - 1'
    if [ $MONTH -eq Ø  ]
    then
       # decrement $YEAR by 1
       YEAR='expr $YEAR - 1'
       # establish leap year
       if [ 'expr ${YEAR} % 4' -eq Ø ]
```

```
        then
           LEAP_YEAR=${TRUE}
        else
           LEAP_YEAR=${FALSE}
        fi
        # re-initialize $MONTH to 12
        MONTH=12
        # re-initialize $DAY to last day of previous month
        DAY=31
    elif [ ${MONTH} -eq  1  -o ${MONTH} -eq 3 -o ${MONTH} -eq 5   -o  \
           ${MONTH} -eq  7  -o ${MONTH} -eq 8 -o ${MONTH} -eq  1Ø  -o \
           ${MONTH} -eq  12 ]
    then
        # re-initialize $DAY to last day of previous month
        DAY=31
    elif [ ${MONTH} -eq  4 -o ${MONTH} -eq 6 -o ${MONTH} -eq 9  -o \
           ${MONTH} -eq  11 ]
    then
        # re-initialize $DAY to last day of previous month
        DAY=3Ø
    elif [ ${MONTH} -eq  2 -a "${LEAP_YEAR}" = "${TRUE}" ]
    then
        # re-initialize $DAY to last day of previous month
        DAY=29
    elif [ ${MONTH} -eq  2 -a "${LEAP_YEAR}" = "${FALSE}" ]
    then
        # re-initialize $DAY to last day of previous month
        DAY=28
    fi
fi
}
#############################################################################
#  Name     : Process3ØDaysMonths
#  Overview : The function processes days that fall in months with
#             thirty days.
#  Input    :
#  Returns  :
#  Notes    :
#############################################################################
Process3ØDaysMonths ()
{
if [  $DAY -gt 3Ø ]
then
    # no of days exceed 3Ø days
    # set $DAY  to  1
    DAY=1
    # increment $MONTH by 1
    MONTH='expr $MONTH + 1'
    if [ $MONTH -gt 12  ]
    then
```

```
      # no of months exceeded 12
      # set $MONTH  to 1
      MONTH=1
      # increment $YEAR by 1
      YEAR='expr $YEAR + 1'
      # establish leap year
      if [ 'expr ${YEAR} % 4' -eq Ø ]
      then
          LEAP_YEAR=${TRUE}
      else
          LEAP_YEAR=${FALSE}
      fi
   fi
elif  [ $DAY  -eq  Ø  ]
then
   # we're taking away days from specified date
   # decrement $MONTH by 1
   MONTH='expr $MONTH - 1'
   if [ $MONTH -eq Ø  ]
   then
      # decrement $YEAR by 1
      YEAR='expr $YEAR - 1'
      # establish leap year
      if [ 'expr ${YEAR} % 4' -eq Ø ]
      then
         LEAP_YEAR=${TRUE}
      else
         LEAP_YEAR=${FALSE}
      fi
      # re-initialize $MONTH to 12
      MONTH=12
      # re-initialize $DAY to last day of previous month
      DAY=31
   elif [ ${MONTH} -eq  1  -o ${MONTH} -eq 3 -o ${MONTH} -eq 5   -o  \
          ${MONTH} -eq  7  -o ${MONTH} -eq 8 -o ${MONTH} -eq  1Ø  -o \
          ${MONTH} -eq  12 ]
   then
       # re-initialize $DAY to last day of previous month
       DAY=31
   elif [ ${MONTH} -eq  4 -o ${MONTH} -eq 6 -o ${MONTH} -eq 9  -o \
          ${MONTH} -eq  11 ]
   then
       # re-initialize $DAY to last day of previous month
       DAY=3Ø
   elif [ ${MONTH} -eq  2 -a "${LEAP_YEAR}" = "${TRUE}" ]
   then
       # re-initialize $DAY to last day of previous month
       DAY=29
   elif [ ${MONTH} -eq  2 -a "${LEAP_YEAR}" = "${FALSE}" ]
   then
```

```
        # re-initialize $DAY to last day of previous month
        DAY=28
    fi
fi
}
#############################################################################
#  Name     : ProcessMonthOfFebruary
#  Overview : The function processs days that fall in month of February.
#  Input    : None
#  Returns  :
#  Notes    :
#############################################################################
ProcessMonthOfFebruary ()
{
if [ "${LEAP_YEAR}" = "${TRUE}" ]
then
   if [  $DAY  -gt 29 ]
   then
      # no of days exceeded 29 for leap year
      # set $DAY to Ø1
      DAY=1
      # increment $MONTH by 1
      MONTH='expr $MONTH + 1'
   elif [  $DAY  -eq Ø ]
   then
       # we're taking away days from specified date
       # set day to last day of January
       DAY=31
       # decrement $MONTH by 1
       MONTH='expr $MONTH - 1'
   fi
# process month of February in non-leap year
elif [ "${LEAP_YEAR}" = "${FALSE}" ]
then
      if [  $DAY -gt 28 ]
      then
          # no of days exceeded 28 for February in non-leap year
          # set $DAY to Ø1
          DAY=1
          # increment $MONTH by 1
          MONTH='expr $MONTH + 1'
      elif [  $DAY  -eq Ø ]
      then
         # we're taking away days from specified date
         # decrement $MONTH by 1
         MONTH='expr $MONTH - 1'
         # set day to last day of January
         DAY=31
      fi
fi
```

```
}
###############################################################################
#  Name     : AddDayToDate
#  Overview : The function adds day ( + or - ) to  a  specified date.
#  Input    : string1 (containing date in format  DDMMYYYY   )
#             string2 (no of  days to be added or substracted)
#  Returns  : Calculated date (in DDMMYYYY format)
#                                    if date is calculated correctly
#             NULL             otherwise
#  Usage    : CALCULATED_DATE='AddDayToDate  "01012002" -35'
#             if [ "${CALCULATED_DATE}" = ""  ]
#             then
#                   echo "Date calculation failed"
#             fi
#  Notes    : 1. When invoking this function, do not set the debug option
#                because any debug message displayed will be assigned to
#                the variable (as shown above).
###############################################################################
AddDayToDate ()
{
# define function name
FUNCTION="${FUNCTION}:AddDayToDate"
# assign parameter
if [ $#  -ne  2  ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${INVALID_NO_ARGS}"
    fi
    return $FALSE
fi
P_DATE_STRING="$1"
P_NO_DAYS="$2"
# initialize $CALCULATED_DATE  with null
CALCULATED_DATE=""
# validate against null
if  [ "${P_DATE_STRING}"  = "" ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${NULL_DATE_STRING}"
    fi
    return "${CALCULATED_DATE}"
elif  [ "${P_NO_DAYS}"  = "" ]
then
    if [ "${DEBUG}" = "${TRUE}" ]
    then
        DisplayMessage E "${NULL_NO_DAYS}"
    fi
    return "${CALCULATED_DATE}"
```

```
fi
# check data format
if ! CheckDateFormat "${P_DATE_STRING}"  "DDMMYYYY"
then
     return $FALSE
fi
FUNCTION="${FUNCTION}:AddDayToDate"
# check no of days
if ! IsInteger "${P_NO_DAYS}"
then
    return $FALSE
fi
FUNCTION="${FUNCTION}:AddDayToDate"
# extract day, month and year
DAY='echo   ${P_DATE_STRING}    | cut -c1-2'
MONTH='echo ${P_DATE_STRING}    | cut -c3-4'
YEAR='echo  ${P_DATE_STRING}    | cut -c5-8'
# establish leap year
if [ 'expr ${YEAR} % 4' -eq Ø ]
then
    LEAP_YEAR=${TRUE}
else
    LEAP_YEAR=${FALSE}
fi
# initialize $NO_DAYS to be added or substracted
NO_DAYS=$P_NO_DAYS
# establish whether days need to be added or subtracted
ADD_DAYS=${TRUE}
if [ ${NO_DAYS} -lt Ø ]
then
     ADD_DAYS=${FALSE}
fi
while [ $NO_DAYS  -ne  Ø  ]
do
   if [ "${ADD_DAYS}" = "${TRUE}" ]
   then
      # increment $DAY by 1
      DAY='expr $DAY + 1'
   else
      # decrement $DAY by 1
      DAY='expr $DAY - 1'
   fi
   # process months with 31 days
   if [ ${MONTH} -eq  1   -o ${MONTH} -eq 3 -o ${MONTH} -eq   5  -o \
        ${MONTH} -eq  7   -o ${MONTH} -eq 8 -o ${MONTH} -eq  1Ø  -o \
        ${MONTH} -eq  12 ]
   then
        Process31DaysMonths
   # process months with 3Ø days
   elif [ ${MONTH} -eq  4 -o ${MONTH} -eq 6 -o ${MONTH} -eq 9  -o \
```

```
        ${MONTH} -eq  11 ]
   then
        Process3ØDaysMonths
   # process month of February
   else
       ProcessMonthOfFebruary
   fi
   # increment or decrement no of days to be added by 1
   if [ "${ADD_DAYS}" = "${TRUE}" ]
   then
      # decrement $NO_DAYS by 1
      NO_DAYS='expr $NO_DAYS - 1'
   else
      # increment $DAY by 1
      NO_DAYS='expr $NO_DAYS + 1'
   fi
done
# left pad  $DAY, $MONTH and $YEAR with zeros
DAY='echo "${DAY}" | awk {'printf("%Ø2d",$1)'}'
MONTH='echo "${MONTH}" | awk {'printf("%Ø2d",$1)'}'
YEAR='echo "${YEAR}" | awk {'printf("%Ø4d",$1)'}'
CALCULATED_DATE="${DAY}${MONTH}${YEAR}"
echo "${CALCULATED_DATE}"
FUNCTION=""
}
# initialize variables
InitialiseVariables
```

Arif Zaman
ETL Developer (UK)

The sed command

The stream editor, **sed**, is a non-interactive text editor that performs editing operations on every line of a file. In its simplest form, **sed** works in a similar way to other commands, such as **grep**, but, when its full power is exploited, it is capable of complex text manipulations. You can use **sed** and its subcommands directly within a script, or you can modify the lines in a specified file according to a number of edit operations located in a text source file.

Since the output of **sed** is sent to standard output by default, it

is ideal for testing any modifications you would like to make prior to making the changes permanent. Alternatively, if your operations produce a lot of output, you can redirect the output to another file and peruse it at your leisure without affecting the original file. Once you are certain that the operations actually perform what you are trying to achieve, then you can overwrite your original.

When a **sed** subcommand includes a pointer to a specific location (address) within a file (either a line number or a search pattern), only the addressed line or lines are affected by the command. Otherwise the command applies to all lines.

At first sight **sed** is a relatively simple command, having limited options. However, the editing operations and pattern matching substitutions can be quite complex, and to fully understand the intricacies of the editor requires extensive practice.

To gain a full understanding of how **sed** works and to experiment with its commands, create the following file, which contains a list of entrants for a competition we have organized, and their associated contestant numbers.

```
$ vi stars

Vicky Spicer            101
Serena Smith            201
Brooklyn Becks          102
Ronaldo Romero          301
Rio Veron               302
Venus Seles             202
Zinedine Batistuta      303
Rivaldo Smith           304
```

Our examples will be demonstrated initially by sending the **sed** output to standard output. To make the changes to **stars** permanent, you should make a copy prior to running each **sed** command, and only later overwrite the original, after the command has achieved its purpose.

GENERAL SYNTAX

The general syntax for **sed** is:

```
sed [-n] [-e command] [-f sourcefile] [file]
```

The **-n** option suppresses all information normally written to standard output.

The **-f sourcefile** option uses **sourcefile** to contain a prepared set of editing commands to be applied to **file**.

The **-e command** option uses the **command** string to perform an editing operation. The subcommands **sed** is to use are placed within each **command**, and there may be any number of **-e** options.

Format of sed subcommands

The **sed** command, when used either with subcommands placed on the command line or with subcommands placed within a text file, can address single or multiple lines within a file on which it is intended to operate. The general format of these subcommands is:

```
[line1[, line2]] operation [parameter]
```

line1 and **line2** are optional line addresses that indicate the line or lines where **sed** should perform its operation. If no line addresses are specified, the operation is performed globally on all lines. Lines can be addressed directly by number or text pattern using fixed character strings, or indirectly by regular expressions.

Operations, or commands, can be grouped at the same address by surrounding the operations by braces, although this cannot be used on the command line and we will discuss this further when we talk about **sed** source files.

The only operation that can take the optional final parameter is the **s** (substitute) operation. The various operations available will be discussed shortly.

Using sed on the command line

When **sed** is used on the command line, the **-e command** option uses **command** as the editing string to be applied. If you are

using just one **-e** flag, it can be omitted. If you put more than one edit command on the **sed** line, each must be preceded by **-e**. For example, individual operations are entered as:

```
sed /Zinedine/d stars
sed /Serena/s/Smith/Williams/ stars
```

but these can be combined in the command:

```
sed -e /Zinedine/d -e /Serena/s/Smith/Williams/ stars
```

Any spaces in the edit subcommand must be surrounded by quotes for **sed** to interpret the command properly. Quotes must also be used if the edit command contains metacharacters, which will most likely happen when using regular expressions. For example:

```
sed -e "/^[ ^I]*$/d" file1 > file2
```

where **^I** represents **Ctrl I**, which is the symbolic representation used to signify that a **tab** has been entered. It does not mean that the characters **^** and **I** have been separately entered. When you press the **tab** key it will usually produce a number of blank spaces, depending on its current size definition.

Replacing character strings

To replace a character string with a different string, the format of the command is:

```
sed s/old_string/new_string/ filename
```

The **s** stands for substitute and the separator, ***/***, can in fact be any character that is not in either of the strings. Using the **stars** file, assume that **Serena** Smith has married prior to our competition and changed her name. We can change her name with:

```
sed s/Smith/Williams/ stars
```

Unfortunately this will change every line in the file which matches the string **Smith**, so that **Rivaldo Smith** becomes **Rivaldo Williams**. To change **Serena Smith**'s name without affecting any other **Smith**, we can use:

```
sed "s/Serena Smith/Serena Williams/" stars
```

Since the strings contain spaces, the entire substitute expression must be surrounded with quotes since it must form only a single argument to **sed**. An alternative way of achieving the same end is by using a text pattern as a line address. For example:

```
sed /Serena/s/Smith/Williams/ stars
```

In this case the substitution of the string **Williams** for the string **Smith** is only performed on the line containing the string **Serena**. The string to be replaced can also be a regular expression. For example, to put a left margin of a tab in the stars file, we can use:

```
sed "s/^/^I/" stars
```

where **^** now indicates a substitution (insertion) at the start of the line. The new string can also be a null string, in which case the original string is deleted. To remove the contestant numbers from the file we could use:

```
sed "s/[ ^I]*[ø-9]*[ ^I]*$//" stars
```

This will delete the pattern, multiple spaces or tabs, followed by multiple digits, followed by multiple spaces or tabs, followed by the end of the line, and should take care of all possible formatting combinations.

Replacing multiple occurrences

Usually, the **s** subcommand tells **sed** to replace only the first occurrence of a pattern on each line. If it is intended to replace all occurrences of the pattern on each line, then the **g** flag must be used. For example:

```
sed s/old_string/new_string/g filename
```

so that:

```
sed /Ron/s/R/W/g stars
```

will change the line to **Wonaldo Womero**.

There may also be occasions when you would like to replace a single character that occurs multiple times on a line, whether the occurrences are contiguous or at separate locations on the line. We can do this using the **n** flag, which is used to specify that the

replacement should be made for only the *n*th occurrence. For example, suppose we wanted to change every second letter **i** on each line to the letter **a**, then we could do this using:

```
sed s/i/a/2 stars
```

This would change the following lines to those shown:

```
Vicky Spacer            1Ø1
Zinedane Batistuta      3Ø3
Rivaldo Smath           3Ø4
```

Assuming that we had one or more tabs between the contestant's name and number, we could replace our multiple tabs with a single colon by using:

```
sed -e "s/^I/:/1" -e "s/^I//" stars
```

The first editing command would replace the first occurrence of a tab with the colon, and the second editing command would replace any further tabs with nothing.

Deleting lines

To delete whole lines of text from a file, the **d** operation can also be used. For example, suppose one of our contestants had made an unexpected exit from our competition. We could delete the name with:

```
sed /Zinedine/d stars
```

This will delete the line containing **Zinedine**. By using regular expressions, it is possible to delete blank, or apparently blank, lines. Try inserting several blank lines (with mixtures of spaces and tabs, or completely blank with no characters) at different locations within the file and then use:

```
sed "/^[ ^I]*$/d" stars
```

This command looks for the start of a line, followed by zero or more spaces or tabs, followed by the end of the line. The **d** operation then deletes all the lines made up entirely of spaces and tabs, or no characters.

Suppressing normal output

Normally, **sed** copies all input lines to the output, transformed by the edit operations performed on them. The **-n** option suppresses this normal output, and only the lines requested with the **p** (print) edit operation appear on the output. For example:

```
$ sed -n /Smith/p stars
Serena Smith        2Ø1
Rivaldo Smith       3Ø4

$ sed -n /Ron/s/R/W/gp stars
Wonaldo Womero      3Ø1
```

If the same edit commands had been used without the **-n** option, then all of the lines would be displayed, and each line containing **Smith** (or **Wonaldo**) would be displayed twice. This is because we see the normal output, as well as the specially requested output. Try the above commands without the **-n** option.

The **-n** option is also useful for selectively displaying parts of a file. For example:

```
sed -n 1,4p stars
```

will display only the first four lines of the file. Similarly:

```
sed -n 1p stars
```

will display the first line. The option can also be used for context displays. For example:

```
sed -n 1,/Smith/p stars
```

will display the first line and all the lines up to and including the next line containing the word **Smith**. In effect this will display only the first two lines. Similar examples are:

```
sed -n /Serena/,/Rio/p stars
sed -n /Smith/,$p stars
```

The first command will display all the lines from the first occurrence of **Serena** up to and including the line containing the word **Rio**, which is four lines. If the first line address appears later in the file than the second line address, then all the lines from the first address up to the end of the file would be displayed. The second

command would display all lines from the first occurrence of **Smith** to the end of the file.

The output from **sed** can be inverted by preceding the operation to be performed with **!**, which will execute the operation on all lines that are not matched by the addresses. For example:

```
sed -n /Smith/!p
```

will output all lines other than those containing the word **Smith**.

USING A SED SOURCE FILE

When **sed** is invoked with the **-f sourcefile** option, the edit commands are taken from the named **sourcefile**. For example:

```
sed -f sourcefile file
```

The name of the file containing the edit commands must be the very next argument after the **-f** option. As usual, you must redirect the output to another file if you want the changes to be permanent.

Suppose, for example, we created a file called **changes** containing the following commands:

```
/Serena/s/Smith/Williams/
/Zinedine/d
$a\
Paddy Rafter          203
1,4{
s/i/a/
     s/[0-9]//g
}
```

If we then ran these commands against our **stars** file with:

```
sed -f changes stars
```

the following changes to the output would occur:

1. **Serena Smith**'s name would be changed to **Serena Williams**.
2. The line containing **Zinedine Batistuta** would be deleted.
3. A line containing **Paddy Rafter** would be added at the end of the file. The **$** sign in front of the **a** operation stands for the last line.

4 The first occurrence of every letter **i** on each of the first four lines would be changed to a letter **a**.

5 The contestant numbers on the first four lines would be deleted.

The output would now look like:

```
Vacky Spicer
Serena Walliams
Brooklyn Becks
Ronaldo Romero
Rio Veron              302
Venus Seles            202
Rivaldo Smith          304
Paddy Rafter           203
```

The last three lines of the **changes** file are an example of multiple commands operating on a group of line addresses, which in this example are lines **1** to **4**. The first command can be placed on the same line as the opening brace, but the closing brace must appear on a line of its own. When using braces, spaces and tabs are allowed at the start of the line, which allows you to indent the operations for better readability.

You must also be aware that you cannot use quotes to surround your commands in a source file, as you would do when editing from the command line. If you do so, you will get error messages.

In our example above, suppose we had wanted to add more than one line at the end of the file. This could have been achieved by changing the **a** operation to:

```
$a\
Paddy Rafter           203\
Elton Bowie            103
```

When using the append operation, the **** must be placed at the end of every line of text to be added, except the last one. You should be aware that you cannot have any characters after the backslash at the end of the line. If you do, and a space will probably be the most common character since it cannot be seen, then you will get an error message saying that a particular function cannot be parsed.

If, for example, you wished to add a line in the middle of the file after the line containing **Serena Smith**, this could be achieved with:

```
/Serena/a\
Sepp Seaman        3Ø5
```

The order in which the commands appear in the edit file are not necessarily the order in which they would be applied to the file to be changed. There is actually a pre-processing stage where **sed** sorts the commands into an order it thinks is logical. For example, deletions take precedence over substitutions.

A further example of an edit file is given below, which can be used to add a blank line after each line in a file:

```
$ vi addblank
a\
<Hit Enter key to insert blank line>
```

The blank lines can be added with:

```
sed -f addblank stars
```

Because a line address was not specified for the **a** operation, the text (blank line) is added after every line. The **-e** and **-f** options can be used together to make the required changes to the stars file:

```
sed -e /Serena/s/Smith/Williams/ -f addblank stars
```

When using both edit operations on the command line and edit commands taken from a text file, the **-e** option must be used for the command line operations, even if there is only one.

EDITING MULTIPLE FILES

The **sed** command can be given more than one filename to edit at a time and the edit commands you specify will be applied to all files. The line numbers increment through all the files and are not reset to 1 at the start of each file. The **$** character is used to mean the last line of the last file.

If we split the **stars** file into three separate files:

```
egrep "[ ^I]+1" stars > popstars
```

```
egrep "[ ^I]+2" stars > tennisstars
egrep "[ ^I]+3" stars > footiestars
```

and use **sed** to print lines 4 to 7:

```
sed -n 4,7p tennisstars footiestars popstars
```

this will display:

```
Rio Veron                       302
Zinedine Batistuta              303
Rivaldo Smith                   304
Vicky Spicer                    101
```

As far as **sed** is concerned, line 4 is the second line of the second file, because the first file contained only two lines.

EDITING WITH PIPELINES

If you do not specify a file to **sed**, standard input is used. This allows **sed** to become a useful filter used in pipelines. For example, to apply a command to every file in a directory:

```
ls | sed "s/^/command /" | ksh
```

The output of **sed** is a series of lines of the form:

```
command file
```

The command lines are then executed by piping them to the shell command **ksh**. Using **sed** in this manner is a useful technique for executing a command on each separate file (or subdirectory) in a directory, although you can often achieve the same end by using **xargs** instead. You may find that you need to run several consecutive **sed** operations against your list of files generated, particularly if you want to build up commands with lots of arguments. Alternatively, you could use a script.

EXAMPLES OF SED

Substitute command

While it is possible to use any character to surround the arguments to the substitute command, the string which actually

addresses the line you want is surrounded by **/** characters – no other character will do. The use of other characters to surround the substitute expressions is necessary when the expression itself contains slashes, for example, when you want to perform a substitution on a full pathname.

An example using non-standard characters is:

```
sed /Serena/s-Smith-Williams- stars
```

The command:

```
sed "s,[Ø-9],,g" stars
```

deletes the contestant numbers from the file. This will leave spaces and/or tabs at the end of the file, and, to get rid of these as well, we need a more precise regular expression:

```
sed "s,[ ^I]*[Ø-9]*$,," stars
```

Transform operation

Another useful operation to the **sed** command is the **y** (transform) operation. Two strings are specified, and each character in the first string is replaced by the equivalent character in the second string. For example:

```
sed y/Ø123456789/9876543210/ stars
```

will transform the contestant digits in the **stars** file. The two strings must be of the same length, and no padding is allowed. If you tried to shorten the first string with:

```
sed y/[Ø-9]/9876543210/ stars
```

you would get an error message. This is because the first string is taken to be five characters long, whereas the second string is ten characters.

Adding blank lines

We have seen how we can add blank lines to a file using a **sed** source file, but how can we do this from the command line? Well, the solution is to use a combination of **sed** and **tr**. First we add a # to the end of every line, and then use **tr** to replace the # with

the new-line character, or **\012** in octal. We have to use **tr** to do this since none of the **sed** operations will accept octal values.

The blank lines can be added with:

```
sed "s/$/#/" stars | tr "#" "\012"
```

If the file that you want to be double-spaced already contains the # character, you can instead use a non-printing character, such as **Ctrl A**:

```
sed "s/$/^A/" file | tr "\001" "\012" > newfile
```

The **^A** indicates where you type **Ctrl A**, not the two separate characters **^** and **A**; **001** is the octal value for **Ctrl A**.

Double spacing

Another way to get double spacing is to use the **r** (read) command to read a file containing only a blank line into the text you want spaced out:

```
sed "r blankline" stars
```

If there is no file with the specified name you give to the **r** command, you don't get an error message; **sed** just carries on and nothing gets read into your file.

Writing to files

The **w** (write) operation can be used to extract lines from one file and read them into another. For example:

```
sed "/Smith/w smiths" stars
```

extracts all the lines in stars containing **Smith** and writes them to the file **smiths**.

The **w** operation can be appended to an **s** operation, and substitutions are performed before the lines are written to the file. For example:

```
sed "s/Smith/Williams/w smiths" stars
```

Using the write operation we could have split our **stars** file with:

```
sed -n   -e "/1..$/w popstars" \
-e "/2..$/w tennisstars" \
-e "/3..$/w footiestars" \
stars
```

Quitting at specified lines

The **q** (quit) operator can be used to display the head of a file. The following command stops producing output after line number **3**:

```
sed "3q stars
```

When given a search pattern, **sed** quits after the first occurrence of a line containing that pattern. For instance:

```
sed /Rio/q stars
```

stops after the line containing **Rio**.

Tonto Kowalski
Guru (UAE)

Automated mechanism for changing passwords

An important security problem occurs when root passwords and important user passwords are not regularly changed. We have noticed in our company that we have to give the root passwords to a lot of people (first/second level supports, administrators, etc), which creates the need to change these root passwords quite often. First of all, to do this is a lot of work – you have to log on to every machine one-by-one and change it. Secondly, it is always painful to update files to let people know that a password has changed and identify which groups need which passwords. That's why I have built an automatic mechanism to change user passwords regularly on all systems in the company and update the appropriate files for the groups needing them.

The mechanism also works well with manual changes. Briefly, it has three stages:

- Stage 1/first script – a password is generated and this password is first of all changed for a dummy user on the main system to get the *encrypted password* and *lastupdate* fields from the */etc/security/passwd* file.
- Stage 2/second script – this is executed on the target system where the password for a given user is to be changed. The fields *encrypted password* and *lastupdate* obtained from the first script are exchanged with the current fields on the target system */etc/security/passwd* file. In this way, the password is changed.
- Stage 3/third script – this is executed on the main system again to update the files. Every group in the company has a directory and, using a file, it identifies in which group's directories the new password file is to be created. Each group can access only their own directory. In this way, they get the passwords they need to know.

STAGE 1/APASSWD.SH

The script works on the main system. The script works with or without parameters. Parameters are used for manual password changes.

Without parameters (automatic):

```
/usr/local/sbin/apasswd.sh
```

The script has the following steps:

1 The script is executed *n* times from the root crontab. It depends on how often in a day you want to run the password changing mechanism. The mechanism is started with the apasswd.sh script, and the target host is randomly selected.

2 In the beginning the script creates/examines an indicator file named *running* to ensure that only one password mechanism is running. That is to avoid automatic and manual password changing mechanisms coinciding. The script checks first of all whether the *running* file exists under the */pwl/.work* directory. If yes, it gives an information message to try it

again later. If not, it starts the automatic password-changing mechanism.

3 The script generates an 8-character password.

4 It changes the password of a dummy user named **pwusr** on the main system. This happens in the script by calling a C program called **pwchanger**. The C code for this program is given below.

It's best to make login/remote login false for the dummy user.

5 It then takes the appropriate fields *encrypted password – 13 characters* and *lastupdate* from the */etc/security/passwd* file.

6 A file containing all hostnames and which groups need which passwords must be manually created (filename: */pwl/.which_group_needs_which_pw*). Here is an example:

```
#-#
#-#
#-# Format:
#-# ------
#-#  host ip-address account pw_group
#-#
tcell002.turkcell.com.tr 10.55.201.28 root first_level,second_level
tcell003.turkcell.com.tr 10.55.201.29 root sap
tcell004.turkcell.com.tr 10.55.201.30 root dba,web
tcell005.turkcell.com.tr 10.55.201.31 root second_level,web
tcell006.turkcell.com.tr 10.55.201.32 root
first_level,second_level,sap,dba
tcell007.turkcell.com.tr 10.55.201.33 root first_level,sap
tcell008.turkcell.com.tr 10.55.201.34 root sap,web,dba
tcell009.turkcell.com.tr 10.55.201.35 root web
.............................................
.............................................
```

Each time, a hostname is randomly selected from this file.

7 Then the script prepares a one-line file named **goner** in a predetermined format under the */pwl/.work* directory. The information contained in it comes from the previous steps.

Format:

```
<hostname> <user> <decrypted pw> <lastupdate> <encrypted pw>
```

Example:

```
tcellØØ5 root Pk9v23u6 1Ø44434669 Yk12uP/mdfgfw
```

Then the **goner** file is zipped, goner.gz, to make the file more secure during ftp.

8 The next step is to send the transfer file to the host (mvsist.turkcell.com.tr).

Each step has an exit code in case of error.

With parameters (manual), first step:

```
/usr/local/sbin/apasswd.sh "<hostname>" "<user>" "<8-char-password>"
```

The script has the following steps:

1 The chosen 8-character password must not include special characters.

2 In the beginning the script creates/examines an indicator file named *running* to ensure that only one password mechanism is running. That is to avoid automatic and manual password changing mechanisms coinciding. The script checks first of all whether the *running* file exists under the */pwl/.work* directory. If yes, it gives an information message to try it again later. If not, it starts the automatic password-changing mechanism.

3 In this step, it checks whether the given parameters are correct.

4 It takes the password entered as a parameter and changes the password of a dummy user named **pwusr** on the main system. This is made in the script by calling a C program called **pwchanger**.

5 It then takes the appropriate fields *encrypted password – 13 characters* and *lastupdate* from the */etc/security/passwd* file.

6 Then the script prepares a one-line file named **goner** in a predetermined format under the */pwl/.work* directory. The information contained in it comes from the previous steps.

Format:

```
<hostname> <user> <decrypted pw> <lastupdate> <encrypted pw>
```

Then the **goner** file is zipped, goner.gz, to make the file more secure during ftp.

7 The next step is to send the transfer file to the host (mvsist.turkcell.com.tr).

Manual password changing can be carried out for both root and non-root users. The manual version of this password changing mechanism is available because sometimes we need to give the root password temporarily to operators. In these cases, this mechanism allows us easily to change the password for a user on a system for a short period. In the meantime, we have to ensure that the automatic mechanism running from **crontab** does not change the password again for this system. That's why, if the user is root, then this hostname is commented in the file *.which_group_needs_which_pw*, which means it is excluded from the next random selection until the comment sign is removed for the system. Briefly, in this step the hostname is commented in this file if the user is root.

Each step has an exit code in case of error.

The second step after making a manual password change is:

```
/usr/local/sbin/apasswd.sh "<hostname>" "<user>"
```

The manually-assigned password has to be replaced afterwards by a randomly-generated password, and the system has to be included again in the list. This script works with two parameters and the steps are as follows:

1 In the beginning the script creates/examines an indicator file named *running* to ensure that only one password mechanism is running.

2 It checks whether the given parameters are correct.

3 It generates an 8-character password.

4 It changes the password of a dummy user named **pwusr** on

the main system. This happens in the script by calling a C program called **pwchanger**.

5 It then takes the appropriate fields *encrypted password – 13 characters* and *lastupdate* from the */etc/security/passwd* file.

6 Then the script prepares a one-line file named **goner** in a predetermined format under the */pwl/.work* directory. The information contained in it comes from the previous steps.

Format:

```
<hostname> <user> <decrypted pw> <lastupdate> <encrypted pw>
```

Then the **goner** file is zipped, goner.gz, to make the file more secure during ftp.

7 The next step is to send the transfer file to the host (mvsist.turkcell.com.tr).

8 It removes the comment in front of the given hostname in the *.which_group_needs_which_pw* file so that the system is available again.

APASSWD.SH

```
#!/bin/ksh
################################################
# The script generates an 8-character password and changes the password
# of a dummy user. This enables the password of a randomly-selected
# remote server to be changed using this 3-character encrypted password
# and its last update.
# A file is created including the info (encrypted password &
# last update, etc) to be replaced later in a local server
# /etc/security/passwd file by another script in order to change its
# given user password ... This script works with and without parameters.
# With parameters runs the manual password changing mechanism.
# Without parameters is the automatic version, which must be scheduled
# in crontab.
################################################
# function that generates random passwords
function pwd_gen {
# inserting the character set into charset array
set -A charset1 'print $letters; print $cap_letters; print $numbers;
print $chars'
```

```
set -A charset2 'print $letters; print $cap_letters; print $numbers'
# generating a password from charset
let i=0
passw=""
while  (( $i < $char_num )) ; do
   ch='print ${charset1[$((RANDOM%${#charset1[*]}))]}'
   passw=${passw}${ch}
   let i=$i+1
done
let i=0
prepassw=""
while  (( $i < 5 )) ; do
   ch='print ${charset2[$((RANDOM%${#charset2[*]}))]}'
   prepassw=${prepassw}$ch
   let i=$i+1
done
typeset -L6 passwd=$passw
typeset -L2 prepasswd=$prepassw
password=${prepasswd}${passwd}
if [[ ${#password} -eq "8" ]] ;then
   print $password
else
   retstr="$retstr102"
   send_mail
   echo $retstr
   exit 102 # password not correctly generated
fi
}
################################################
# This function gets out a random line from the file
function choose_line {
cat $fname | grep -v "^#"| grep . | nl |
while read index rest ; do
   line[index-1]=$rest
done
# getting a random element from the array
random_line='print ${line[$((RANDOM%${#line[*]}))]}'

if [ ${#random_line} -eq 0 ] ; then
   retstr=$retstr103
   send_mail
   echo $retstr
   exit 103 # Random line from the file is not correct
fi
}
################################################
# This function changes the password for the dedicated user
function user_pwd_chg {
# Create a file to make sure only one script is executed
touch $flag_file
```

```
# generating a new passwd for the dedicated user
# (if not manually entered) and changing the password
/usr/local/sbin/pwchanger "$usr" "${decripted_pw:=`pwd_gen`}" > /dev/
null 2>&1
if [ $? -eq Ø ] ; then
   prep_file
else
   retstr=$retstr1Ø1
   send_mail
   echo $retstr
   exit 1Ø1 # pwchanger is not successfully executed
fi
}
################################################
# This function prepares the file to be sent to Host.
function prep_file {
# Getting the name of the system that will have password change.
if [[ $chosen_host = "" ]] && [[ $chosen_user = "" ]] ; then
  choose_line
  chosen_host='print $random_line | awk '{print $1}''
  chosen_host=${chosen_host%%.*}
  #chosen_ip='print $random_line | awk '{print $2}''
  chosen_user='print $random_line | awk '{print $3}''
fi
# getting encrypted password and lastupdate fields
# from /etc/security/passwd
# finding the line # where the user is
chk='cat $pwd_file | grep ${usr}: | wc -l'
if [ chk -eq 1 ] ; then
   let lno='nl -ba $pwd_file | grep ${usr}: | awk '{print $1}''
   twolines='sed -n "$((${lno}+1))','$((${lno}+2))'p' $pwd_file | grep -
E "password|lastupdate" | wc -l'
   if [ $twolines -eq 2 ] ; then
      # taking out encrypted password field
      epwd='sed -n "$((${lno}+1))','$((${lno}+2))'p' $pwd_file | grep
password | awk -F= '{print $2}''
      # taking out the lastupdate field
      lupt='sed -n "$((${lno}+1))','$((${lno}+2))'p' $pwd_file | grep
lastupdate | awk -F= '{print $2}''
      # getting out the spaces from the strings
      encrypted_pw='print $epwd | tr -d '\Ø''
      lastupdate='print $lupt | tr -d '\Ø''
      # writing the fields to the goner file
      echo "$chosen_host $chosen_user $decripted_pw $lastupdate
$encrypted_pw" > $goner_file
      # compressing the file not to be seen with cat during transfer.
      if [ -s $goner_file ] ; then
         /usr/bin/gzip -f $goner_file > /dev/null 2>&1
         if [ "$?" != "Ø" ] ; then
            retstr=$retstr1Ø6
```

```
            send_mail
            echo $retstr
            exit 106 # file not compressed
         fi
      else
         retstr=$retstr107
         send_mail
         echo $retstr
         exit 107 # file does not exist or empty
      fi
   else
      retstr=$retstr105
      send_mail
      echo $retstr
      exit 105
           # sed could not get the correct lines in /etc/security/passwd
   fi
else
  retstr=$retstr104
  send_mail
  echo $retstr
  exit 104 # cannot find user in /etc/security/passwd file or string
douple found.
fi
}
################################################
# This function sends mail to a list of users when an error has occurred
function send_mail {
echo "error/info message" | /usr/bin/mailx -s "$retstr" -r apasswd@root
$mailto
}
################################################
# This function FTPs to the Host
function ftp_file {
# Check whether we are in the right directory in local server
cd ${local_dir} > $logfile
if [[ $(pwd) != ${local_dir} ]]
then
   retstr=$retstr108
   send_mail
   echo $retstr
   exit 108 # cannot change to the local directory
fi
# Check whether the Host is pingable
ping -c 2 ${mvs_host} 1 > /dev/null 2>&1
if [[ $? != 0 ]]
then
   retstr=$retstr109
   send_mail
   echo $retstr
```

```
   exit 1Ø9 # cannot ping Host
fi
# Start FTP Job
ftp -v -n ${mvs_host} << ! >> $logfile
user $mvs_user $mvs_pwd
prompt
bin
$mvs_par
put ${goner_file}.gz ,$mvs_replace'
fi
bye
!
# Check whether successfully connected to Host
grep -q "Connected to mvsist.turkcell.com.tr" $logfile
if [ $? != Ø ]
then
   retstr=$retstr11Ø
   send_mail
   echo $retstr
   exit 11Ø # Cannot connect to Host
fi
# Checking whether the user could log in.
grep -q "Login failed" $logfile
if [ $? -eq Ø ]
then
   retstr=$retstr111
   send_mail
   echo $retstr
   exit 111 # Login failed check user/passwd
fi
# Checking whether ftp is successful
grep -q "bytes sent in" $logfile
if [[ $? != Ø ]]
then
   retstr=$retstr112
   send_mail
   echo $retstr
   exit 112
fi
}
################################################
# This function displays usage
function usage {
cat << EOT
# Automatic version without parameter:
$scriptname
# Changing password manual:
  (encrypted-PW-generation + put comment for <hostname>)
$scriptname "<hostname>" "<user>" "<8-char-password>"
# Changing back manual changed password:
```

```
  (PW-generation + remove comment for <hostname>)
$scriptname "<hostname>" "<user>"
EOT
}
################################################
# This function checks whether the entered hostname is correct and
commented
function hostname_check2 {
if [[ $chosen_user = "root" ]] ; then
   cat $fname | grep "^#" | grep -q "$chosen_host"
   if [[ $? != Ø ]] ; then
      echo $outstr123
      exit 123 # manual given hostname not in List or not commented
(root)
   fi
else
   cat $fname | grep -q "$chosen_host"
   if [[ $? != Ø ]] ; then
      echo $outstr129
      exit 129 # manual given hostname not in List (non-root)
   fi
fi
}
################################################
# This function checks whether the entered hostname
# is correct and not commented
function hostname_check3 {
if [[ $chosen_user = "root" ]] ; then
   cat $fname | grep -v "^#" | grep -q "$chosen_host"
   if [[ $? != Ø ]] ; then
      echo $outstr128
      exit 128 # manual given hostname not in List or commented (root)
   fi
else
   cat $fname | grep -q "$chosen_host"
   if [[ $? != Ø ]] ; then
      echo $outstr129
      exit 129 # manual given hostname not in List (non-root)
   fi
fi
}
################################################
# This function checks whether the entered password is valid.
function passwd_check {
if [[ ${#decripted_pw} = 8 ]] ; then
  if [[ $decripted_pw != +([Ø-9,a-z,A-Z]) ]] ; then
     echo $outstr125
     exit 125
  fi
else
```

```
     echo $outstr126
     exit 126
fi
}
################################################
# This function comments the entered hostname in the file
function commenter {
xline='cat $fname | grep "$chosen_host"'
cat $fname | grep -v $chosen_host > $fname_tmp
echo "#${xline}" >> $fname_tmp
mv $fname_tmp $fname
}
################################################
# This function uncomments the entered hostname in the file
function uncommenter {
cat $fname | grep "$chosen_host" | grep -q "#"
if [ $? -eq Ø ] ; then
  xline='cat $fname | grep "$chosen_host" | sed 's/.//''
  cat $fname | grep -v $chosen_host > $fname_tmp
  echo "${xline}" >> $fname_tmp
  mv $fname_tmp $fname
else
   echo $outstr124
   exit 124
fi
}
################################################
# variables
letters="a b c d e f g h i j k l m n o p q r s t u v w x y z"
cap_letters='echo $letters | tr [:lower:] [:upper:]'
numbers="Ø 1 2 3 4 5 6 7 8 9"
chars="% / ( ) ? + - _ > < . , ; :"
char_num=13
fname=/pwl/.which_group_needs_which_pw
fname_tmp=/pwl/.which_group_needs_which_pw_tmp
goner_file=/pwl/.work/goner
local_dir=/pwl/.work
pwd_file=/etc/security/passwd
usr=pwusr
grps="first_level second_level sap dba web"
grps_dir=/pwl
pwdlog=/pwl/.work/pwd_history
logfile=${local_dir}/ftp.log
mailto="administrators@turkcell.com.tr"
mvs_host=mvsist.eil.risnet.de
mvs_user=ftpuser
mvs_pwd=xxxxxx
mvs_par="site recfm=fb lrecl=8Ø"
mvs_replace=»IVP.ZB.SQ###.BETA48.PW»
flag_file=/pwl/.work/running
```

```
scriptname=$(echo $0)
##############################################
# error/info codes
retstr101="ERROR: pwchanger is not successfully executed! PASSWORD NOT
CHANGED FOR $usr !!!"
retstr102="ERROR: password is not correctly generated! PASSWORD NOT
CHANGED FOR $usr !!!"
retstr103="ERROR: random line from the $fname is not correct! FILE NOT
SENT !!!"
retstr104="ERROR: cannot find user in /etc/security/passwd file or
string douple found! FILE NOT SENT !!!"
retstr105="ERROR: sed could not get the correct lines in /etc/security/
passwd! FILE NOT SENT !!!"
retstr106="ERROR: $goner_file could not be compressed! FILE NOT SENT
!!!"
retstr107="ERROR: $goner_file does not exist or empty! FILE NOT SENT
!!!"
retstr108="ERROR: Cannot change to ${local_dir}! FILE NOT SENT !!!"
retstr109="ERROR: Cannot ping $mvs_host! FILE NOT SENT !!!"
retstr110="ERROR: Cannot connect to $mvs_host! FILE NOT SENT !!!"
retstr111="ERROR: Login failed, check user and password! FILE NOT SENT
!!!"
retstr112="ERROR: Unknown failure! FILE NOT SENT !!!"
outstr127="ERROR: Wrong number of parameters are given !!!"
outstr128="ERROR: Given hostname not in list or already commented (user:
root) !!!"
outstr126="ERROR: Given password is not 8 characters !!!"
outstr125="ERROR: Given password must be letters and/or numbers (without
special characters) !!!"
outstr124="ERROR: Given hostname is not manual changed before! (not
commented in the file) !!!"
outstr123="ERROR: Given hostname not in list or NOT commented before
(user: root) !!!"
outstr129="ERROR: Given hostname not in list (user: non-root) !!!"
##############################################
# initialization
decripted_pw=""
chosen_host=""
chosen_usr=""
# MAIN ########################################
if [ -a $flag_file ] ; then
  print
  echo "Password Changing Machanism is already running, please try
later!!!"
  print
else
  if [[ $# = 3 ]] ; then  # changing password manual (en_passwd gen +
putting comment sign)
     chosen_host=$1
     chosen_user=$2
```

```
     hostname_check3
     decripted_pw=$3
     passwd_check
     user_pwd_chg
     ftp_file
     if [[ $chosen_user = "root" ]] ; then
        commenter
     fi
  elif [[ $# = 2 ]] ; then # changing manual changed password (PW gen +
remove comment sign)
     chosen_host=$1
     chosen_user=$2
     hostname_check2
     user_pwd_chg
     ftp_file
     if [[ $chosen_user = "root" ]] ; then
        uncommenter
     fi
  elif [[ $# = Ø ]] ; then # automatic execution without a parameter.
     user_pwd_chg
     ftp_file
  else
     echo $outstr127
     print
     usage
     exit 127 # wrong number of parameters
  fi
fi
################################################
```

PWCHANGER.C

(/usr/local/sbin/pwchanger <user> <password>):

```
/*
  Use the following command to compile:
     # cc -o  pwchanger pwchanger.c -ls
*/
#include <userpw.h>
#include <sys/types.h>
#include <pwd.h>
#include <usersec.h>
#include <errno.h>
#define  KEYCRYPTLEN 2
void usage (){
  printf(«Usage: pwchanger user passwd\n»);
  exit(1);
}
main (int argc,char *argv[]){
```

```
int id;
struct userpw *p;
char user[PW_NAMELEN], pass[1Ø], pass2[256];
char *nwpass;
char **message;
struct userpw   newpw;        /* passwd structure if getuserpw fails */
if (argc != 3) usage();
/* calculate encrypted passwd ...    */
strcpy(user,argv[1]);
strcpy(pass,argv[2]);
strncpy(pass2,pass,2);
nwpass=(pass,pass2);
/* infos DEBUG
  printf("passwd crypti : %s\n",nwpass);
  printf("Avant modif : \n");
  printf(«------------ \n»);
*/
/* open data bases for read and write */
        setpwdb (S_READ|S_WRITE);
        setuserdb (S_READ|S_WRITE);
p=getuserpw(user);
/* Will verify whether the user exists ... */
                      /* Check whether the user exists */
                       if (getuserattr(user,S_ID,&id,SEC_INT))
                       {
                printf(«\nuser %s inconnu ...\n\n»,user);
                             return(-1);
                exit(ENOENT);
                       }
if (!(p=getuserpw(user))){
printf("structure passwd vide  %s  ...\n\n",user);
printf("%s : user\n",user);
/* initialize new userpw struct  */
strcpy(newpw.upw_name,user);
newpw.upw_passwd = nwpass;
newpw.upw_lastupdate = time ((long *) Ø);
newpw.upw_flags =Ø;
printf("Nom        : %s\n",user);
printf("Password   : %s\n",newpw.upw_passwd);
printf("LastUpdate : %u\n",newpw.upw_lastupdate);
printf("Flags      : %u\n\n\n",newpw.upw_flags);
putuserpwhist(&newpw, message);
putuserattr(user,S_PWD,"!",SEC_CHAR);
putuserattr(user,NULL,NULL,SEC_COMMIT);
p=getuserpw(user);
/*putuserpw(p); */
}
/* infos DEBUG
  printf("Nom        : %s\n",p->upw_name);
  printf("Password   : %s\n",p->upw_passwd);
```

```
  printf("LastUpdate : %u\n",p->upw_lastupdate);
  printf("Flags      : %u\n\n\n",p->upw_flags);
*/
strcpy(p->upw_passwd,crypt((const *)pass,(const *)pass2));
/* infos DEBUG
  printf(«Apres modif MEMOIRE : \n»);
  printf(«------------ \n»);
  printf("Nom        : %s\n",p->upw_name);
  printf("Password   : %s\n",p->upw_passwd);
  printf("LastUpdate : %u\n",p->upw_lastupdate);
  printf("Flags      : %u\n\n\n",p->upw_flags);
*/
setpwdb(S_WRITE);
putuserpw(p);
endpwdb();
}
```

STAGE 2/PASSCH.SH

This script works on the local server of the given user whose password we want to change. As soon as apasswd.sh (first script) sends the **goner** file to the host (MVS system) without an error code, then the host analyses the file and knows in which machine the passch.sh should be executed and with which parameters.

For script execution for remote servers on the host system we have software called **beta** available:

```
/usr/local/sbin/passch.sh "<user>" "<encrypted password>" "<lastupdate>"
```

The script has the following steps and is executed by OPC/beta on the host:

1. The hostname in the **goner** file decides in which system the script is going to be executed and with which parameters.
2. Back-up the */etc/security/passwd* file as */etc/security/passwd.old*.
3. Replace <user> with the *encrypted password* and *lastupdate* fields in the */etc/security/passwd* with the new ones in the **goner** file.

Each step has an exit code in case of error.

Editor's note: the code for passch.sh and stage 3 of this security system will be published next month.

Adnan Akbas
System Administrator
TURKCELL (Germany)

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.

Articles, or article proposals, can be sent to the editor, Trevor Eddolls, at any of the addresses shown on page 2. Alternatively, you can e-mail him at trevore@xephon.com

AIX news

IBM has announced High Availability Cluster Multi-Processing for AIX 5L, V5.1.0 (HACMP V5.1), which offers improved usability, functionality, and performance, and also a new optional package, HACMP/XD (Extended Distance), which offers multiple data back-up and disaster recovery technologies. HACMP/XD provides extended distance for ESS/PPRC peers and unlimited distance for IP-connected peers using HAGEO technology.

New Version 5 features include consolidation of all previous forms of HACMP (HAS, CRM, ES, ESCRM) into one HACMP offering, fast disk fallover, which takes less than ten second, simplified configuration through streamlined user interface, and heartbeat by shared disk, which offers additional protection against network-partitioned cluster data divergence.

There's an enhanced security mechanism, which removes the need for /.rhosts, an integrated cluster file system option, which uses GPFS 1.5, performance rewrite of cluster verification and cluster single-point-of-control, and customized control of startup and fallback application and resource behaviour.

HACMP/XD will immediately support ESS PPRC so that HACMP/XD clusters now support automatic fallover of disks that are PPRC pairs.

For further information contact your local IBM representative.
URL: http://www-1.ibm.com/servers/aix/products/ibmsw/high_avail_network/hacmp_51.html.

* * *

Metron has announced the launch of Athene Version 7.4, which includes automatic reporting, data capture and CustomDB.

With automatic reporting, users will now be able to group servers by application and thus receive reports on a specified application across a group of servers, rather than on a single server basis. The same concept of 'grouping' also applies to disks within Version 7.4.

Athene 7.4 incorporates the ability to capture Netstat data from AIX as well as HP-UX, Solaris, and Tru64.

CustomDB provides an 'integration' facility with the ability to collect data from a wide range of different applications and environments including network statistics and other operating systems.

For further information contact:
Metron, Osborne House, Trull Road, Taunton, Somerset, TA1 4PX, UK.
Tel: (01823) 259231
URL: http://www.metron.co.uk/products/athene/index.html.

* * *

IBM has announced that it is expanding the DB2 solutions supporting Workgroup Edition, Workgroup Server Edition, and Workgroup Server Unlimited Edition by offering enhanced tools for AIX, Linux, and Windows.

The new support is available in DB2 Table Editor for Workgroups V4.3, DB2 High Performance Unload for Workgroups V2.1, DB2 Performance Expert for Workgroups V1.1, and DB2 Web Query Tool for Workgroups V1.3.

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