****
****

Writing Shell Scripts and Shell Calls in VnmrJ / VNMR Macros

****
****

Last update: 2008-03-27

List of Titles in the Sequence of Appearance in the Section Below:

==

1998-01-16:

DOING FTP IN SHELL SCRIPTS:

By using ~/.netrc it becomes possible to use ftp within shell scripts, by defining the input to ftp in a "here document". This can only work if ftp does not ask for a user name and password. Here's an example of such a script:

```
        #!/bin/sh
        if [ $# -eq 0 ]; then
          echo "Usage: ftpsend filename"
          exit
        fi
        ftp host_name > /dev/null << +
        bin
        put $1
        bye
        +
```

Note that you must NOT place a backslash in front of the character following ">>", as this would prevent the shell from replacing $1 with the first argument to the script. The target host name could of course also be made an argument, to have more flexibility.

Such a shell script can be useful to simplify printing on a Codonics dye sublimation printer (a device that we use with imaging systems, and which is based on a SPARC engine). Printing on this device is usually done by sending the data via FTP, and the password (which could be stored in ~/.netrc in this case) is a number (usually we use 2) that defines certain print parameters such as the scaling and the format of the output.

[Agilent MR News 1998-01-16]

1998-06-20:

UNIX TRICKS - EXTRACTING FILE CREATION TIME AND DATE:

Occasionally, users want to process file creation or modification dates in shell scripts or VNMR macros. One way to obtain this information is with the UNIX command "ls -l". Unfortunately, "ls -l" has two different output formats, depending on how old a file is:

```
        vnmr1 - 1> cd /vnmr/bin
        vnmr1 - 2> ls -l pulsetool Vnmr
        -rwxrwxr-x   1 vnmr1    nmr      1884600 May  8 20:37 Vnmr
        -rwxr-xr-x   1 vnmr1    nmr       100540 Oct  1  1997 pulsetool
```

If you want to process the date information is a macro or shell script, this complicates your work, as you first need to figure out which one of the two formats is used. There is one way to always get the same date format, including both the year and the time:

```
        vnmr1 - 1> cd /vnmr/bin
        vnmr1 - 2> tar cf - pulsetool Vnmr | tar tvf -
        -rwxr-xr-x 1001/101 100540 Oct  1 18:21 1997 pulsetool
        -rwxrwxr-x 1001/1011884600 May  8 20:37 1998 Vnmr
```

This example demonstrates that processing "tar" output may be non-trivial, too, as for large files there may be no space between the group ID and the file size in bytes. Therefore, to be on the safe side, you would need to count tokens from the end of the line!

Be aware, however, that for directories the "tar" construct show every single subfile - you may get more output than you expect! For a single file you could use the following construct, where "file" is the file or directory name you want to process:

```
        tar cfP file | tar tvf - | grep 'file$'
```

The "P" option in the first "tar" command suppresses the addition of a trailing "/" to directory names.

[Agilent MR News 1998-06-20]

1999-01-23:

UNIX TRICKS - HANDLING FILES WITH "DIFFICULT" NAMES:

UNIX is fairly liberal in its file naming options: In principle, a file or directory name can be up to 255 characters long (notice however, that not all software can handle extremely long file names!). The characters for file names include numeric as well as lower and upper case alphabetic characters. Many special characters such as @%_-+=:,. can also be used for file names. Restrictions exist with the following special characters:

- the slash (/) is interpreted as directory level separator and cannot appear inside a file name.
- the characters *?[];<>|~#!&$(){}'"`^ are interpreted by the shell and should not be used in file names. It is particularly dangerous to use the asterisk (*) in filenames, as attempts to remove such a file may lead to the removal of other files!
- the dot (.) is used as extension separator, but can otherwise also be used in names. A leading dot makes file names invisible with the standard "ls" command (unless the -a or the -A option is used).
- the backslash (\) is interpreted as "escape" character and permits handling special and wildcard characters in file names, e.g.:

```
      vnmr1 - 1> touch a\&b a\\b
      vnmr1 - 2> ls
      a&b    a\b
      vnmr1 - 3> rm a\&b a\\b
```

- the blank is used as token separator by the shell and does normally not appear inside UNIX file names. However, filenames with blanks can be handled using single or double quotes:

```
      vnmr1> rm "name with blanks"
```

  File names with leading or trailing blanks can be handled the same way, but should be avoided wherever possible, as they may cause lots of confusion.
- tokens with a leading minus (-) character are interpreted as option (rather than as regular argument) by many UNIX commands; a leading '-' character should therefore be avoided. To remove a filename with leading blank you can use "rm --":

```
      vnmr1 - 1> ls
      -I
      vnmr1 - 2> rm -I
      rm: illegal option -- I
      usage: rm [-fiRr] file ...
      vnmr1 - 3> rm -- -I
```

Most software is protected against the creation of files with "illegal" or "difficult" filenames - but as there are ways to remove such files, there are also ways to create them! "Non-standard" filenames can also be obtained when transferring data from a non-UNIX system, such as a Mac, where blanks are often used inside file names.

[Agilent MR News 1999-01-23]

1999-04-08:

READING VNMR PARAMETER FILES WITHIN A UNIX SHELL:

Reading parameter files from a UNIX shell script or a C program is somewhat non-trivial, because VNMR stores parameters on multiple lines, e.g.:

```
        bs 7 1 32767 0 1 2 1 0 1 64
        1 16
        0
        axisf 4 2 4 0 0 4 1 0 1 64
        1 "s"
        4 "m" "n" "s" "u"
```

Every parameter is stored on at least 3 lines (the actual values may be spread over multiple lines in arrayed parameters): the first line contains the name of the parameter and its characteristics (see also section 5.4 of the VNMR User Programming Manual), the second line contains the number of values (1 for non-arrayed parameters), the last line contains enumerative values.

This means that you cannot simply take "grep" to extract the value of a specific parameter from a VNMR parameter file. Fortunately, there is awk / nawk! To read the "system" parameter from /vnmr/conpar, you could use

```
        vnmr1 - 1> nawk '/^system / {getline; print $2}' < /vnmr/conpar
        "spectrometer"
```

Note the syntax in the matching string: by preceding the pattern with a caret (^) we only pick lines that start with the matching string; the blank after the string excludes longer parameter names that start with "system". The "getline" in the "nawk" string skips to the second line, where we pick the second token - the parameter value. You can of course also directly set a shell variable with the parameter value, and we can use "sed" to remove the double quotes from string values (this example is for Bourne shell scripts):

```
        cd /vnmr
        system=`nawk '/^system / {getline; print $2}' < conpar | sed 's/"//g'`
```

For arrayed parameters, this would of course only return the first value.

Alternatively, you could download and install "bin/parhandler" from the user library. This contribution contains a utility "listparam" that prints VNMR parameters in a simple, 1 line-per-value format, which can be used as follows:

```
        system=`listparam /vnmr/conpar | awk '{print $2}' | sed 's/"//g'`
```

[Agilent MR News 1999-04-08]

1999-08-07:

SHELL CALLS IN VNMR - ARGUMENT SEPARATION:

UNIX separates command arguments by spaces rather than commas as in VNMR. The first argument of a "shell" call is either a complete command string, just the command name, or the name of the UNIX command and some of its arguments (separated by spaces). Additional arguments provided in the "shell" call are separated by spaces on a UNIX level. The calls

```
        shell('ls -l *.fid')
```

and

```
        shell('ls','-l','*.fid')
```

are equivalent. Multiple arguments are useful when using local variables:

```
        shell('ls -l',userdir)
```

If UNIX command tokens are to be composed from different MAGICAL elements (such as strings, expressions and variables), spaces are not permitted, and MAGICAL string concatenation MUST be used:

```
        shell('ls -l',userdir+'/maclib')
```

Note that userdir does not end with a slash! If it did end with a slash, this would not matter, as '/' and '//' are equivalent as path level separators.

[Agilent MR News 1999-08-07]

SHELL CALLS IN VNMR - PIPES, INPUT FROM FILES:

VNMR does not provide a "standard input channel" for subshells, i.e., it is not possible to call UNIX commands that require the user to type input. A construct such as

```
        write('line3','type contents of the new macro:')
        shell('cat >',userdir+'/maclib/newmacro)
```

can not work - to the contrary: the cat command could actually be waiting forever for some text and a terminating <Ctrl-d> to be provided via standard input! In order to avoid such situations, VNMR automatically supplements shell command strings with '< /dev/null': this way an empty string (i.e., <Ctrl-d>) is fed into the UNIX command, and hangups can be avoided. However, this leads to a new problem: in calls such as

```
        shell('ls -l /vnmr/maclib | grep ft')
```

the second call receives two inputs as this translates to

```
        ls -l /vnmr/maclib | grep ft < /dev/null
```

This results in an error. The simplest way to avoid this case is to add ';cat' to the command string:

```
        shell('ls -l /vnmr/maclib | grep ft; cat')
```

In UNIX this translates to

```
        ls -l /vnmr/maclib | grep ft; cat < /dev/null
```

i.e., "cat" rather than the command in the pipe chain captures the "default input". The same trick must be used if input is to be taken from a file:

```
        shell('ls -l < file_list; cat')
```

Alternatively, you can use parentheses in the shell call, i.e., pack the commands that are linked via pipe into a subshell:

```
        shell('(ls -l /vnmr/maclib | grep ft)')
```

The subshell now behaves like a single command; this also avoids problems with multiple inputs to the command following the pipe symbol.

[Agilent MR News 1999-08-07]

1999-08-14:

SHELL CALLS IN VNMR - CAPTURING OUTPUT:

The VNMR shell command output is displayed in the text window. Shell calls return at least a newline character, even if the UNIX command(s) produce no output. Writing these newline characters to the text window can cause the text window to start scrolling. If you want to avoid this scrolling, attach a dummy return argument to the shell command, e.g.:

```
        shell('rm -rf tmpfile'):$dummy
```

ANY return argument will cause output to the text window to be suppressed. Each line of output will be fed into one return argument, i.e., if you want to fill multiple variables, you must make sure these values are returned on a separate line each. Make sure you predefine the return variables:

```
        $numfiles=0 shell('ls *.fid | wc -l; cat'):$numfiles
        $text1='' shell('head -1',curexp+'/text'):$text1
```

otherwise the variable type will be determined based on the first character of the returned string (this is valid for ANY MAGICAL return argument).

If you want to fill multiple return variables with one call to the shell command, you need to construct the UNIX command so that every return argument is on a separate line (space-separated tokens are taken as one string!), e.g.:

```
        $file1='' $file2='' $file3=''
        shell('ls -dt1 `find * -name \'*.fid\'`'):$file1,$file2,$file3
```

This finds the three most recent FIDs in the current directory. Extra output is discarded (this shell command lists ALL FIDs - you may want to make sure it doesn't search the entire disk!). The "1" option forces "ls" to list one file (or directory) per line of output.

[Agilent MR News 1999-08-14]

SHELL CALLS IN VNMR - CAPTURING ERROR MESSAGES:

UNIX error messages are fed into the shell from which VNMR was started (typically the console window). To capture errors in VNMR use '2>&1', which attaches error output to the standard output (note that VNMR shell calls are Bourne shells, see also Agilent MR News 1993-09-07):

```
        shell('ls *.fid 2>&1')
```

With this method, standard output and error output are not separated - this should not be used when shell output is captured into return arguments. If on the other hand you want to capture JUST the error output, you can discard the standard output, such as in

```
        $error=''
        shell('ls *.fid 2>&1 >/dev/null'):$error
```

Note that the redirection order matters: '>/dev/null 2>&1' would discard ALL output and always returns an empty string!

[Agilent MR News 1999-08-14]

SHELL CALLS IN VNMR - ESCAPING SPECIAL CHARACTERS:

The MAGICAL interpreter is checking for backslash ('\') characters within strings. If you need backslashes in a shell call you need to escape these with a second backslash. Similarly, single quotes are used as string delimiters in MAGICAL; if you need single quotes in a shell command string, these also need to be escaped with a backslash character or enclosed with reverse single quotes:

```
        shell('sed \'s/^\\//#/g\' <',$file,'; cat')
```

This shell call displays a file ($file) with all slashes ('/') at the beginning of a line substituted by a hash character ('#'). The beginning of the line in the "sed" call is indicated by the caret ('^') character. The slash serves as pattern delimiter in the "sed" call and therefore needs to be escaped with a backslash (which in turn requires a backslash as escape for the MAGICAL interpreter).

[Agilent MR News 1999-08-14]

1999-09-18:

WRITING TCL SCRIPTS THAT EXECUTE COMMANDS IN VNMR:

VNMR includes several Tcl/Tk scripts/utilities which interact with VNMR itself, i.e., Tcl tools that send MAGICAL commands to VNMR. If you want to develop new utilities of this kind you need to observe a few things: it is best to take one of the VNMR utilities such as the "temp" command as example:

- for sending commands to VNMR we had to modify the standard Tcl "wish" (the Tcl "window shell"). To use the modified "wish" you need to start your Tcl script with a line
  ```
       #! /vnmr/tcl/bin/vnmrwish -f
  ```
- this will start a background utility. This is intentional and built into the vnmrwish program. It allows you to have such a tool running while you use VNMR the normal way. If instead you want VNMR to wait for the Tcl utility to terminate (e.g., if you need this utility to make input into VNMR), you can substitute the above header line for
  ```
       #! /vnmr/tcl/bin/vnmrWish -f
  ```
  The upper case "W" makes the Tcl tool run in foreground, i.e., no command entry in VNMR is possible while this utility is running. "vnmrWish" only became available with VNMR 6.1.
- It is best to start the Tcl tool using a "wrapper shell script". For the VNMR "temp" utility (/vnmr/tcl/bin/temp) this wrapper script is a shell script /vnmr/bin/vnmr_temp and contains the following lines:
  ```
       #!/bin/csh
       setenv TCLDIR $vnmrsystem/tcl
       setenv TCL_LIBRARY $TCLDIR/tcllibrary
       setenv TK_LIBRARY $TCLDIR/tklibrary
       cd $TCLDIR/bin
       ./temp "$1" &
  ```
  This wrapper script ensures that the necessary environment variables are defined, and that the Tcl script is called from the right directory etc. In principle, this (C shell detour) should not be necessary, as these environment variables are also defined in the user's .login - but it is safer to do it that way.
- This shell script is then called via VNMR macro "temp" which contains the following shell call:
  ```
       shell('/usr/bin/csh',systemdir+'/bin/vnmr_temp "'+vnmraddr+'"'):$dum
  ```
  This macro doesn't just call /vnmr/bin/vnmr_temp - it also passes on the contents of the "vnmraddr" parameter as an argument. That parameter contains the local host name, the (software communication) port number, and the VNMR process-ID. Remember: there could be multiple copies of VNMR running, even for the same user (considering background VNMR calls during an acquisition), i.e., the Tcl "vnmrsend" command (built into "vnmrwish") can not just check the process table for a program named "Vnmr"!
- In the Tcl script, the command "vnmrsend" is used to send VNMR a command, e.g.:
  ```
       vnmrsend sethw('vt','reset')
  ```

Note that over the recent VNMR releases we have kept upgrading to newer versions of Tcl/Tk:

VNMR 5.2:	Tcl 7.4 / Tk 4.0
VNMR 5.3:	Tcl 7.6 / Tk 4.2
VNMR 6.1:	Tcl 8.0 / Tk 8.0

[Agilent MR News 1999-09-18]

1999-09-25:

WRITING A TCL SCRIPT THAT RETURNS A VALUE TO A MACRO:

In the last issue of Agilent MR News we gave you an introduction into how VNMR typically calls Tcl/Tk utilities, and how such utilities can interact with VNMR. This involves several software layers interacting with each other and seems complicated for a simple input utility for VNMR. There is a simpler solution if such a utility does not need to call VNMR commands. For example, if you want a simple Tcl/Tk tool that can be called from within VNMR and which returns an input value to VNMR.

In this case, it is not necessary to create a separate wrapper script calling a Tcl script. You can create a single shell script in /vnmr/bin which can be called from within a VNMR macro using

```
        shell('script_name'<,shell_arguments>):$answer
```

In this case, the shell script can take VNMR input via shell arguments, and it can write information to standard output. this can then be read into one or several VNMR variables (remember that if you want to fill several VNMR variables from a single shell call, each of these values need to be returned on a separate line). Let's take an example: a shell script which will ask for a single input value. VNMR should be able to specify the prompt, but if you call the script without arguments, it will display a default prompt. A typical VNMR call would then be

```
        shell('tcl_input','"Enter pulse width: "'):$pw
```

assuming that the script tcl_input is in the command path, typically ~/bin or /vnmr/bin (please refer to the article in Agilent MR News 1999-08-28 on handling the return arguments from such a shell call).

In VNMR 6.1 you can use the following shell script to achieve this task:

```
        #! /bin/sh
        # Tcl script to prompt for user input \
        exec $vnmrsystem/tcl/bin/vnmrWish -f "$0" "$1"
        proc sendAndExit {} {
           global value
           puts $value
           exit 0
        }
        frame .ot
        if {$argc > 1} {
           set label [lindex $argv 0]
        } else {
           set label "Enter value: "
        }
        label .ot.label -text $label
        entry .ot.en -width 10  -textvariable value -relief sunken -bd 2
        bind .ot.en <Return> "sendAndExit"
        pack .ot.label .ot.en -in .ot -side left
        pack .ot -side top -expand y -fill x
```

Note a few peculiarities about this script:

- we call vnmrWish (this works in VNMR 6.1 only!) to make the Tcl script execute in foreground. With the background version "vnmrwish" we could not take over any output into VNMR, see the last issue of Agilent MR News.
- Note the backslash at the end of the second line: This script first runs a Bourne shell. The Bourne shell reads the second line as comment and ignores the trailing backslash. It then executes the third line, in which it calls a Tcl shell (vnmrWish) that executes the entire script (the same file!). DO NOT DELETE THAT BACKSLASH!
- For the Tcl shell all lines starting with a hash sign ('#') are comment. A backslash at the end of a comment line extends the comment to the following line, i.e., the Tcl shell does NOT execute the line starting with "exec", but instead the Tcl script that follows!

[Agilent MR News 1999-09-25]

2000-07-08:

HANDLING SHELL CALLS WITH PIPES OR INPUT REDIRECTION:

In Agilent MR News 1999-08-07 we showed how to avoid ambiguous input errors in shell calls involving pipes ("|") or input redirection ("<"). Such errors are caused by VNMR adding "< /dev/null" to every shell call in order to avoid commands that are hanging, waiting for standard (keyboard) input. The recipe we have proposed so far involved adding "; cat" to the command string, as shown in the article above.

Bruce Adams proposes a different approach that may be easier to understand: simply make such shell command strings look like a single UNIX command by using parentheses (i.e., execute such command strings in a subshell):

```
        shell('(wc -l < /vnmr/vnmrrev)'):r1
```

or

```
        shell('(cat /vnmr/vnmrrev | wc -l)'):r1
```

Both solutions ("; cat" and the subshell) should be functionally equivalent. In both cases, however, you are bypassing the protection (through the default input from "/dev/null") against commands that might be waiting for input: it is up to you to ensure that such shell commands don't expect keyboard input.

[Agilent MR News 2000-07-08]

2000-07-22:

BATCH PLOTTING OF DATA USING VNMR IN A SHELL SCRIPT:

Periodically, Tim White (Australian Defense Force Academy, Canberra, Australia) needs to convert a series of VNMR 1D data into PostScript (EPS) files. Tim proposed the following shell script for batch processing (the editor has converted Tim's "bash" script into a regular Bourne shell script, as most users don't have "bash" installed):

```
        #!/bin/sh
        echo "This takes a while so be patient  :-)"
        prc="wft aphx pl pscale pltext pap"
        targetdir=$HOME/PostScript/
        curdir=`pwd`
        for i in `find * -name '*.fid' -type d -print`; do
          base=`basename $i | sed 's/fid/eps/'`
          source=$curdir/$i
          target=$targetdir/$base
          vbg 5 "n1=plotter plotter='psplot' rt('"$source"') $prc \
                 page('"$target"') plotter=n1"
          sleep 12
          echo "File $base stored in $target"
        done
```

This script calls VNMR in background, in exp5. Note that while this script is executing, exp5 must be unused in foreground. The 12 seconds "sleep" interval should be sufficient for VNMR to complete its background task. If you have a slower workstation and are getting "Can't Lock Experiment" errors, you should increase the sleeping time. "prc" can of course contain any VNMR processing and plotting macro(s), as long as they DON'T call "page".

The above script produces an "EPS" file for EVERY VNMR FID in the current working directory and in any subdirectory below. Note that you need to have a PostScript plotter (called "psplot" in the above example) defined for VNMR (in "/vnmr/devicenames"), even if in reality you don't have such a device! Define it as type "PS_A" for portrait type output, or as type "PS_AR" for landscape type output. The output is saved in a directory specified in the variable "targetdir".

Note that this script will NOT work in a "cron" job - in the next issue we will discuss how to make this possible.

[Agilent MR News 2000-07-22]

2000-09-21:

LAUNCHING SHELL WINDOWS FROM WITHIN VNMR:

Starting a shell window from within VNMR is easy: just type "shell", or select the "UNIX" button in the second-row main menu (menu "main2"), which is equivalent. The disadvantage with this is that this shell tool will run in foreground, i.e., any commands typed in VNMR after this will only execute once you quit the shell window. There are several ways to resolve this:

- Use either the CDE toolbar or the CDE or OpenWindows background menu to launch shell windows - typically, tasks that you do in such windows aren't directly linked to what you are currently doing in VNMR anyway.
- Call a background "shelltool" from within a VNMR "shell" command:

  ```
       shell('shelltool &')
  ```

  You will notice that the resulting window differs from the result of "shell" in the layout (font, size). You could either solve this with command line arguments, such as

  ```
       shell('shelltool -fn 9x15bold &')
  ```

  use

  ```
       xlsfonts | more
  ```

  in a shell window for a list of available fonts. Alternatively (and much better), call a "dtterm":

  ```
       shell('dtterm &')
  ```

  This allows you to change the window and font size dynamically while the window is open. You can also modify the menu "/vnmr/menulib/main2" by changing line 26 from

  ```
       mstring[$ix]='shell menu(`main`)'
  ```

  to

  ```
       mstring[$ix]='shell(`dtterm &`) menu(`main`)'
  ```

[Agilent MR News 2000-09-21]

2001-02-11:

SIMPLIFYING SHELL SCRIPTS - "xargs":

Albin Otter (University of Alberta, Edmonton, Canada) was confronted with the task of having to periodically cancel pending print jobs that cause the print spooling to hang (see the article below). The idea was, to set up a little shell script that could be called via "cron". One approach to this would be to use a "for" loop in a Bourne shell, e.g.:

```
        #!/bin/sh
        printjobs=`lpstat -o all | awk '{ print $2 }'`
        for j in $printjobs; do
          cancel -u $j
        done
```

The reason for the "for" loop is that "cancel" only accepts one job-ID per call, therefore

```
        cancel -u `lpstat -o all | awk '{ print $2 }'`
```

would NOT work. With the help of a local UNIX "guru", Albin was able to do the above job in a single line:

```
        lpstat -o all | awk '{ print $2 }' | xargs -n1 cancel -u {}
```

The "lpstat" command returns a list of all pending print jobs, with the job-ID in the second column. The "awk" call prints the second token from every line of the "lpstat" output, and these job-IDs are then fed into the "xargs" command. "xargs" in this incantation calls "cancel" once for every line of input, constructing no more than one command line argument ("-n1" argument) in lieu of the placeholder "{}". As this construct contains nothing that is specific to either the Bourne or the C shell, you can even drop the header shell script header line (the "for" loop syntax in the first version is specific, therefore you should have a header line "#!/bin/sh" to make sure the system is not trying to run this in a C shell.

With the second solution you don't even need to construct a shell script - you can directly add a line such as

```
        30 3 * * * lpstat -o all | awk '{ print $2 }' | xargs -n1 cancel -u {}
```

(which calls this chain of commands at 3:30 a.m. every day) to the "cron" jobs for root:

```
        su
        csh
        setenv EDITOR vi
        crontab -e root
```

[Agilent MR News 2001-02-11]

2001-04-09:

ESCAPING BACKSLASH CHARACTERS IN VNMR SHELL CALLS:

The MAGICAL "escape" character is the backslash ("\") - as with UNIX shells: certain UNIX commands involve special characters such as the semicolon which are normally interpreted by the UNIX shell (the semicolon is the UNIX command separator). To prevent the interpretation of such a character, you can "escape it" with a backslash character:

```
        find . -name '*.fid' -exec ls -ld {} \;
```

in this call, the semicolon terminates the command string that is executed by the "find" command. The wildcard argument "*.fid" also needs to be protected from an interpretation by the shell - we need to enclose it in single quotes ("'").

If you want to use such a "shell" call from within VNMR, you need to escape both the backslash and the single quotes:

```
        shell('find . -name \'*.fid\' -exec ls -ld {} \\;')
```

The braces ("{}") in the UNIX command string that is executed by "find" stand for the names of the files found, i.e., of the files which match the given condition.

[Agilent MR News 2001-04-09]

2001-06-09:

RENAMING LOTS OF FILES:

In the last issue (Agilent MR News 2001-06-06) we proposed a scheme for filenames containing date information that offers improved / automatic sorting in directory listings ("ls" in UNIX, "files" in VNMR) - but we did not provide a recipe on how to rename an archive of data with older style filenames. This is a task that is best done using a shell script. If you have a VNMR FID archive with filenames "<something>ddmmyy.fid", you can use the following script to rename these files to the new style:

```
        #!/bin/sh
        # assumes filename format is "<something>ddmmyy.fid"
        for f in *[0-9][0-9][0-9][0-9][0-9][0-9].fid; do
          newname=`echo $f | nawk '
          {
            len = length($1)
            ext = substr($1,len-3,4)
            year = substr($1,len-5,2)
            if (year > 80)
              year = 1900 + year
            else
              year = 2000 + year
            month = substr($1,len-7,2)
            day = substr($1,len-9,2)
            body = substr($1,1,len-10)
            printf("%s%d-%02d-%02d%s\n",body,year,month,day,ext)
          }'`
          if [ -d $newname ]; then
            echo "$f: $newname already exists - file NOT renamed"
          else
            mv $f $newname
            echo "$f renamed to $newname"
          fi
        done
```

Store this file as "~/bin/isoname", then (in a shell window) type

```
        chmod 755 ~/bin/isoname
        rehash
```

and it is ready to use! Note that this version does NOT search directory trees, but only the current working directory. For a searching macro you could replace the line

```
        for f in *[0-9][0-9][0-9][0-9][0-9][0-9].fid; do
```

with

```
        for f in `find . -name '*[0-9][0-9][0-9][0-9][0-9][0-9].fid'`; do
```

[Agilent MR News 2001-06-09]

2002-05-25:

HOW TO FIND A SPECIFIC FID FILE?

Today's large disks of 20, 40, or more GBytes can hold a huge number of NMR FIDs - enough to make it hard to find a particular data set! While VnmrJ has its own database utility (Locator) that helps handling such large numbers of datasets, VNMR users usually need to resort to UNIX utilities for finding their data. In this article we would like to give you some pointers on how to do this.

How do you find a file with a given name? What if you only know parts of a name? The obvious solution is to use "find", e.g.:

```
        find . -name 'abc8*.fid' -print
```

This would report all VNMR FIDs with a name starting with "abc8". The "." is the search path (you can specify multiple directories as search path, with absolute or relative path names). In order to use wildcard characters ("*" for any number of characters, "?" for exactly ONE character, "[a-z]" for a lower case character, "[0-9]" for a numeric character, etc.) you need to put single quotes around the search string. If you are looking for FIDs it is a good idea to add ".fid" to the search string. The "-print" argument is not required in Solaris (SVR4 UNIX) - "find" prints by default if there is no "-exec" or "-ok" argument.

A more difficult task is to find FIDs that have a particular string inside their "text" file! Here, a combination of "find" and "grep" can help:

```
        find . -name 'text' -exec grep -l 'abc8*' {} \;
```

This will return the path names to all "text" files containing the specified string (but not the contents of these files). Note that "{}" stands for the file that "find" found, and the "\;" is required to terminate the command string executed by "find". To retain the FID file names only you could filter away the last part of the file name:

```
        find . -name 'text' -exec grep -l 'abc8*' {} \; | sed 's/\/text$//'
```

It's actually not necessary to type any of these commands: you can simply open the CDE file manager and select "Find..." from the "File" menu. This starts a search utility that lets you search one search path at a time. You can search by file name, by content (or both!), and/or by file size, owner, permission, and modification date. If you want to search "/vnmr" or any other path that is a symbolic link, you need to select "Follow symbolic links" (in "find" this corresponds to using the "-follow" option).

[Agilent MR News 2002-05-25]

MAINTAINING A TEXT ARCHIVE FOR EASIER DATA ACCESS:

The commands in the previous article rely on "find" searching the disk when you call the command, and for an entire 20 or 40 GByte disk this can take a LONG time! A much faster solution would be to build / maintain a database of FID files with the associated text file contents. Such a database (as a text file) can be constructed with a simple shell script "~/bin/listtxt" such as

```
        #!/bin/sh
        # listtxt - collect text information
        path="/data $HOME"
        outfile=$HOME/FID_database
        chmod +w $outfile
        grep '.' `find $path -name text` /dev/null >$outfile 2>/dev/null
        chmod -w $outfile
```

This script generates a text file with the following type of content:

```
        filename1:textline1
        filename2:textline1
        filename2:textline2
        filename3:textline1
```

(multi-line text will lead to multiple entries for a single FID file). The "find" command returns the path name of the text files found. "grep" matches ANY text, i.e., it will print out all text lines, and the first "/dev/null" serves as a second file argument in case only one FID is found ("grep" only reports the file name if multiple files are searched). The output is fed into the specified text file "~/FID_database", the search path is specified below the script header. The "2>/dev/null" discards error output (omit if you want errors to be reported to you by e-mail). We secure the database file by keeping it read-only except while we create it.

This script should be executed at regular intervals, e.g., through "cron". To do this, type

```
        setenv EDITOR dtpad
        crontab -e
```

then add the following line to your "crontab" file:

```
        30 1 * * 1-5  listtxt
```

(make sure "listtxt is executable by typing "chmod +x ~/bin/listtxt"!). This calls "listtxt" at 1:30 a.m. every weekday, causing the FID database to be rebuilt from scratch (this avoids "dead" entries from files that you have deleted).

You could open the database file with a text editor and use the "find text" utility to look for a particular string - but this is clumsy, considering the potential size of the database. An easier-to-use solution would be a shell script "~/bin/searchtxt" such as

```
        #!/bin/sh
        # searchtxt - search text contents in FID database
        db=$HOME/FID_database
        if [ $# -eq 0 ]; then
          while [ "x$r" = x ]; do
            echo "Enter keyword or phrase to search for: \c"
            read r
          done
        else
          r=$*
        fi
        rm -f /tmp/searchtxt.out
        lines=`awk -F: '{print $2}' <$db |grep -in "$r" |awk -F: '{print $1}'`
        if [ "x$lines" != "x" ]; then
          for ln in $lines; do
            head -$ln $db | tail -1 | awk -F: '{print $1}' | \
                    sed 's/\/text$//' >> /tmp/searchtxt.out
          done
        fi
        cmd=`basename $0`
        if [ ! -f /tmp/searchtxt.out ]; then
          echo "$cmd: no matching files!"
        else
          cat /tmp/searchtxt.out | sort -bdfu
        fi
```

This script searches the text contents only (the "-i" option to "grep" makes the search case-insensitive). To search the filenames, simply replace "$2" on the line starting with "lines=" by "$1". To search both the filename AND the text, change that line to

```
        lines=`cat $db | grep -in "$r" | awk -F: '{print $1}'`
```

If the search string can be assumed NOT to contain spaces, the above line and the subsequent "if" statement can be simplified to

```
        cat $db | nawk -F: '{if ($2 ~ /'$r'/) print $1}' | \
                sed 's/\/text$//' > /tmp/searchtxt.out
```

The editor would like to thank Sue Rhodes (CSIR, Modderfontein, Rep. of South Africa) for help in testing early versions of this script.

[Agilent MR News 2002-05-25]

2002-06-03:

USING A FID TEXT DATABASE FOR ARCHIVING SELECTED DATA:

In the last issue (Agilent MR News 2002-05-25) we proposed a utility "listtxt" for maintaining an archive of FID files with their associated text files, in order to be able to search not only for file names, but also for specific text annotations. The latter task can be done with the second proposed shell script, "searchtxt", which can be set up to search either the text or both the filename AND the text.

This utility can be instrumental in a task that many NMR labs are facing periodically: upon the completion of a Ph.D., or after the conclusion of a research project one would like to archive the collection of all NMR data relating to that person or research project. This is easy if the data are already organized such that the relevant files are in one directory location, but can be tricky or tedious if the files are spread over several directory trees, or if the "collection attribute" (name of the researcher or the name or code of the compound) is only found in the text. In the first case, you can use "find" to locate the files in a first step, putting the result into a text file (for the purpose of the script below use an absolute search path):

```
        find /space -name 'abc8*.fid' > search_results
```

If you need text search and/or if you have already built a FID database, you don't need to search the disk, but you can use the "searchtxt" utility that we presented in the last issue:

```
        searchtxt "abc8" > search_results
```

Now, assuming you have enough disk space for TWO temporary copies of these data files on your disk, you can use the following script "~/bin/collect_fids" to collect these data in a single directory and then directly generate an ISO 9660 file system:

```
#!/bin/sh
fidcollection=$HOME/search_results
if [ $# -gt 0 ]; then
  ch1=`echo $1 | cut -b1`
  if [ $ch1 = '/' ]; then
    fidcollection=$1
  else
    fidcollection=`pwd`/$1
  fi
fi
if [ ! -s $fidcollection ]; then
  echo "File $fidcollection not found or empty, aborting."
  exit 1
fi
cat $fidcollection | sed 's/^\///' > $fidcollection.tmp
date=`date '+%Y-%m-%d_%H:%M'`
dir=`pwd`
targetdir=$dir/fids_$date
mkdir $targetdir
echo "   Files to be archived:"
cat $fidcollection
echo "   Collecting files ... "
cd /
tar cf - -I $fidcollection.tmp | (cd $targetdir; tar xfBp -)
cd $dir
rm $fidcollection.tmp
size=`du -sk $targetdir | cut -f1`
sizeMB=`echo $size | awk '{printf("%4.2f\n",$1/1024)}'`
free=`df -k . | tail -1 | awk '{print $4}'`
freeMB=`echo $free | awk '{printf("%3.1f\n",$1/1024)}'`
if [ $free -lt $size ]; then
  echo "$sizeMB MBytes of data collected in $targetdir"
  echo "Free disk space ($freeMB MBytes) too small for CD file system"
  exit
fi
echo "Enter archive volume name (<17 chars, <Return> to abort): \c"
read vol
if [ "x$vol" = x ]; then
  echo "$sizeMB MBytes of data collected in $targetdir"
else
  vol=`echo $vol | sed 's/^[ ]*//' | sed 's/[ ]*$//' | tr ' ' '_'`
  vol=`echo $vol | cut -b1-16`
  iso=`pwd`/$vol.iso
  mkisofs -d -D -f -J -l -L -N -r -V $vol -o $iso $targetdir
  rm -rf $targetdir
  echo "Data collected in ISO 9660 file $iso"
fi
```

The only (optional) argument to this script is the name of the text file containing the list of the FIDs to be collected (NOTE: the files in this list MUST be given with ABSOLUTE PATH NAMES for the script to work!). The default file name is defined at the start of the script.

It would of course be possible to build the "searchtxt" call into this script, but a two-step procedure gives you the option to remove entries or split the search results into multiple files (in case the resulting ISO 9660 file system is too big), or to add extra ones, e.g.:

```
searchtxt "abc9" >> search_results
```

The above script then collects the FIDs (and parameter files) listed in the search results in a temporary directory. Note that we remove the leading "/" as we need to use relative filenames. You can then stop the script by giving a simple return (empty string) as ISO 9660 volume name, otherwise the script directly generates the ISO 9660 file system under the name "volume_name.iso", and the temporary FID directory is removed. In order to avoid erroneous volume names, we make sure that leading and trailing blanks are removed, remaining blanks are translated to underscore characters, and the result is truncated to 16 characters maximum.

In order to avoid naming conflicts, the above script "exports" the full file paths into the ISO 9660 file system. For instance, if you archived a file

```
/export/home/vnmr1/mydata/abc.fid
```

that file can later be retrieved from the CD-ROM under

```
/cdrom/volume_name/export/home/vnmr1/mydata/abc.fid
```

or

```
/cdrom/cdrom0/export/home/vnmr1/mydata/abc.fid
```

This may sound complex - but it is a safe and generally applicable solution.

Thanks to Reinhard Machinek (University of Goettingen, FRG) for suggesting the above application of "searchtxt"!

[Agilent MR News 2002-06-03]

2002-07-22:

HANDLING FILES WITH "ILLEGAL" FILE NAMES:

In Agilent MR News 1999-01-23 we provided some hints for handling file names with "illegal" characters. A recent discussion on AMMRL prompted the editor to add a few explanations beyond what was already stated in the above article. First and foremost, we should re-state that the CDE file manager can be used to delete such files (just drag them to the trash can) or change their names: selecting the file name with the left mouse button gives you an insertion cursor that you can move with the left and right arrow and then add or delete characters in a file name; you can also highlight a partial or complete file name and retype (parts of) a name.

There are two possible problems of completely different nature with using or addressing such file names in a shell script or command line:

- the shell may interpret special characters such as "*?[];<>|~#!&$(){}'"`^"; the solution for such names is to
    - enclose such file names in single quotes (in this case you CANNOT use wildcard characters - you must specify the COMPLETE file name)
    - enclose PART of a file name in single quotes, e.g.: to remove a series of files with names starting with "a*bcd" you could use

      ```
      rm 'a*'bcd*
      ```

      Here, the first asterisk is NOT interpreted by the shell, but the second one is.
    - use backslashes to hide special characters from the shell:

      ```
      rm a\*bcd*
      ```
- the above tricks do NOT help if a file name starts with a "-" character: in this case, the shell is NOT the culprit, but it's the UNIX command that interprets arguments starting with "-" as command line options, and a command such as

  ```
  rm -abcde
  ```

  reports that "a", "b", "c", "d", and "e" are illegal options regardless of whether this argument is enclosed in quotes or whether the "-" is escaped with a backslash character (only "f", "i", "r" and "R" are allowed options in the case of "rm"). The problem is that the COMMAND (not the shell) uses the leading "-" to differentiate between command line options and file name arguments. Using wildcard characters usually does not help:

  ```
  rm *abcde
  ```

  is interpreted by the shell, the command still receives the previous argument which is interpreted as command line option. Putting the wildcard character between quotes or using a backslash to hide it does NOT help either: most UNIX commands are NOT able to interpret wildcards (with very few exceptions it's the shell which handles wildcard characters). However, there ARE solutions for this problem:
    - the PROPER solution is to use "-" as first file name argument, e.g.:

      ```
      rm - -abcde
      ```

      or

      ```
      rm -r - -abcde
      ```

      In shell scripts or command line calls you could use this as a general solution with "rm", "mv", and "cp" etc. if you cannot exclude the presence of file names with a leading "-".
    - with commands such as "rm", "mv", and "cp", file name arguments MUST be specified AFTER any command line options. Therefore, a call such as

      ```
      rm myfile -abcde
      ```

      DOES work. For the same reason using wildcard characters such as in

      ```
      rm *abcde
      ```

      MAY work if expanding the wildcard character does NOT yield the name name starting with "-" as first result (e.g., if there are also file names starting with "+" or "%"). As "-" precedes all alphanumeric characters in the ASCII table (see "man ascii"), file names starting with "-" are usually shown at the top of a file list (which by default is sorted alphabetically). You could use constructs such as

      ```
      rm `ls -r *bcde`
      ```

  to invert the sorting order.

- recursively removing / moving / copying directories with such file names, or making the "illegal" file name part of a longer or absolute path works without problem, e.g.:

  ```
          rm -r myfids/-abcde.fid
  ```

  or (in a C shell)

  ```
          rm ~/-abcde
  ```

  (note that in a Bourne shell or in a "shell" call from within VNMR you must use "$HOME" rather than "~").

[Agilent MR News 2002-07-22]

2003-02-25:

MAKING SHELL CALLS WORK IN VnmrSGI:

Catherine Gharbaoui (Ligand Pharmaceuticals, San Diego, CA) found that in VnmrSGI (under IRIX 6.5.8), "shell" calls such as

```
        shell('cat file_name')
```

and

```
        shell('uuencode file_name decode_path > filename.uu'):$dum
```

need to be changed into

```
        shell('cat file_name; cat')
```

and

```
        shell('uuencode file_name decode_path > filename.uu; cat'):$dum
```

to make them work as expected. Note that both "cat" and "uuencode" also (optionally) accept standard input - this causes an error because of the implicit "< /dev/null" in VNMR "shell" calls (see Agilent MR News 1999-08-07, Agilent MR News 2000-07-08). In Solaris, both versions work OK.

[Agilent MR News 2003-02-25]

2003-03-22:

AVOIDING "cp" FOR COPYING DIRECTORIES WITH SYMBOLIC LINKS:

Over the past years (Agilent MR News 1998-03-26, 2003-01-25) we posted articles pointing out that it is undesirable to use "cp -r" on directories containing symbolic links. There are several reasons for this:

- "cp" follows symbolic links, i.e., it treats them as regular files: the target will contain plain files and directories instead. In the case of a VNMR software directory this will cause "config" to fail, as it will fail to remove and re-create the links "/vnmr/stdpar" and "/vnmr/tests";
- copying files instead of symbolic links also consumes extra disk space;
- it is possible to create a symbolic link pointing to a parent directory. In this case, "cp -r" follows the link and copies the parent directory, along with another copy of the source directory, causing a loop which "cp" will follow 20 times ("cp" stops once a file path includes 20 symbolic links). This could easily fill a disk of any size - and even just removing the resulting nested directory trees can be time consuming and painful.

The editor's favorite method for copying directories properly is

```
        cd source_dir
        tar cf - . | (cd target_dir; tar xfBp -)
```

In Agilent MR News 1999-10-15 we posted an alternate method using "cpio":

```
        cd parent_of_source_dir
        find source_dir -print -depth | cpio -pdm new_parent_dir
```

While both methods work OK, they are hard to memorize, and even harder to understand for beginners. The editor therefore constructed a short shell script "pcp" ("preserving copy") that does the same job as easily as "cp -r":

```
        #!/bin/sh
        # pcp - recursive copy that preserves symbolic links
        if [ $# -gt 2 -o $# -eq 0 ]; then
          echo "Usage:  pcp <source_dir> target_dir
          exit
        fi
        if [ $# -eq 2 ]; then
          cd $1
          target_dir=$2
        else
          target_dir=$1
        fi
        if [ ! -d $target_dir ]; then
```

```
          mkdir $target_dir
          if [ ! -d $target_dir ]; then
            echo "pcp error: cannot create $target_dir"
            exit
          fi
        fi
        tar cf - . | (cd $target_dir; tar xfBp -)
        echo " ... done."
```

If you specify only one argument, then "pcp" copies the working directory to the specified location. For transferring directories such as "/vnmr", "pcp" or the commands above MUST be called as root in order to ensure that no file ownership is altered. Note that you can use the "find" command to detect / locate symbolic links, e.g.:

```
        find . -type l
```

[Agilent MR News 2003-03-22]

WRITING SHELL SCRIPTS IN A MODERN UNIX ENVIRONMENT:

Traditionally (i.e., in SunOS and older Solaris versions), shell scripts used to be written either as C shell or (mostly) as Bourne shell scripts (the latter is said to run more efficiently, and it's syntax is a bit easier to learn). Unlike operating systems such as MS-DOS which recognize file types by file name extensions, UNIX looks for a "magic number" (or string) in the text to decide how (with which interpreter etc.) to execute a given file. In the case of shell scripts there are two conventions:

- if the first character in a shell script is a ":" (possibly followed by comment on the same line), the file is assumed to be a Bourne shell script; if on the other hand the first character in the script is a "#" (again possibly followed by comment on the same line), the file is interpreted as a C shell script.
- alternatively and PREFERABLY (see below), the interpreter program can be specified explicitly with a FIRST line reading
  ```
  #!/bin/sh
  ```
  for a Bourne shell script, or
  ```
  #!/bin/csh
  ```
  for C shell scripts. The exclamation mark may be followed by a blank. Such a line MUST be the FIRST line in the script.

We STRONGLY recommend using the second selector mechanism: recent Solaris versions offer new shell interpreters, such as the Korn shell ("ksh", or "/bin/ksh"), or the "bash" ("GNU Bourne Again shell", "/bin/bash"). With these new options, the dual choice of the first selector mechanism above is no longer adequate - plus, it may actually fail! For instance, if you are calling a shell script starting with a ":" in the "bash" environment, the script will be interpreted by "bash" rather than "/bin/sh" (the addition of a line

```
        #!/bin/sh
```

further down in the script has NO EFFECT (see also bug "load_nmr.6103"). The second mechanism always works: it explicitly specifies the shell interpreter with the full command path.

[Agilent MR News 2003-03-22]

2003-11-23:

WORKAROUNDS FOR "ARGUMENTS TOO LONG" ERRORS (by Dimitris Argyropoulos, Varian)

When working with software, one may wish to search a directory for files that contain a particular expression. Typical examples with VNMR would be

- you want to find which macros are interfering with a particular parameter variable, or
- you want to find examples for uses of a particular command or MAGICAL operator / function

For example, to find all occurrences of "substr" in any VNMR macro one would open a shell window and type

```
        cd /vnmr/maclib
        grep substr *
```

However if one tries to do this in the VNMR 6.1C or VnmrJ 1.1C "maclib" after having installed either Chempack 2.2 ("chempack/CP" from the on-line user library) and/or BioPack ("psglib/BioPack" from the on-line user library) in "/vnmr", an error message

```
        Arguments too long
```

appears. This means that the arguments generated by the "*" wildcard, i.e., the resulting list of filenames is longer than the maximum available buffer of the shell and the command cannot be executed. In fact, there are two limits:

- one imposed by the operating system kernel, and
- one imposed by the shell used.

Usually these are very long and not easy to reach, but as VnmrJ and VNMR have been evolving "maclib" may grow out of this limit. There is no such limit neither to the "grep" command nor to the size of the "/vnmr/maclib" directory. In fact, all shell calls with the "*" wildcard in this directory, such as

```
mv * /vnmr/maclib.bk
```

would fail, as will any other command line that expands to a huge number of file names, such as wildcards covering the contents of many directories (e.g., "ls -ld /vnmr/*/*"). There are several workarounds for this:

- the "cheap workaround" is simply to "split up the wildcard search range", for example,

  ```
  grep substr [_A-l]*
  grep substr [m-z]*
  ```

  Depending on the number of files you may have to use more and smaller ranges - a trial-and-error approach. Don't forget files starting with a special character, such as "_"!
- instead of sending all the filenames at once with the "*" wildcard one can use the "xargs" utility which will send the files up to the maximum and then continue with the rest until the end. An example of this would be:

  ```
  ls -d * | xargs grep substr
  ```

  (see also Agilent MR News 2001-02-11 for another use of "xargs"). The "-d" option in "ls" is needed if "/vnmr/maclib" contains subdirectories such as "maclib.imaging": without the "-d" option, "ls *" would also yield the (simple) names of all files in these subdirectories, which would then not be found by the "grep" command.
- since in this case the limit is imposed by the C shell, one can use another shell instead of the default C shell: any of the Bourne shell ("sh"), Korn shell ("ksh"), Cornell shell ("tcsh") or Bourne Again Shell ("bash") place the limit higher and thus solve the problem, at least for "/vnmr/maclib":

  ```
  cd /vnmr/maclib
  sh
  grep substr *
  exit
  ```

  This is the fastest solution, as "grep" is called once only.
- if the limit is reached also with these other shells, then, apart from the "xargs" utility, one may want to use a structure with a "while ... done" loop, such as the one below (written specifically for the Bourne shell):

  ```
  cd /vnmr/maclib
  sh
  ls | while read f; do grep substr "$f"; done
  exit
  ```

  In this case the execution is slow, as "grep" is called once for every single file. Note the inclusion of "$f" in double quotes in order to take care of possible filenames with "strange" characters such as spaces.
- you can use the "find" utility to call "grep" - in this case, you could even cover files in subdirectories:

  ```
  cd /vnmr/maclib
  find . -type f -exec grep substr "{}" /dev/null \;
  ```

  Note that the first argument to "find" here is ".", NOT "*" (the latter would cause exactly the problem we want to avoid!). The "{}" in the "exec" part stands for the filename(s) found, the "-type f" avoids calling "grep" on directories, and the extra "/dev/null" is supplied such that "grep" sees two filenames (if only one filename is supplied, the name of the file with matching lines is not shown in the output). This solution is again MUCH slower than the "xargs" version, not only because it also covers files in subdirectories, but mainly because "find" will call "grep" for every single plain file found.

Note that the "*" wildcard does NOT cover "dot files" such as ".login"; if you want such files to be covered (but not "." and "..") you would need to specify either "* .??*" (all regular files plus filenames starting with a dot and having at least two more characters), or "* .[a-zA-Z0-9]*" (all regular files plus filenames starting with a dot followed by an alphanumerical character). If "ls" is involved, you can use the "-A" option to cover "dot files", but excluding "." and ".." (the "-a" option would include these as well). Using "." as "find" search path covers ALL subfiles, including "dot files".

[Agilent MR News 2003-11-23]

TIME STAMPING "AFTER THE FACT":

In Agilent MR News 2003-10-05 we discussed how to introduce and use a "timestamp" parameter for comparing data creation dates. In Agilent MR News 2003-10-17 we followed up with an article discussing time stamping in VnmrJ.

Craig Grimmer (University of Natal, Republic of South Africa), uses an entirely different approach for time stamping which we would like to present here: every evening before midnight Craig runs a "cron" job in which he adds or inserts the current date into the newly acquired FIDs, i.e., he adds a time stamp to the file name rather than using a parameter value. This not only has the advantage that the creation date is easily recognizable from the file name, but it also automatically sorts files chronologically in listings using "ls" etc.! One proposal which would prepend the filenames with the date value is realized in the following shell script:

```
        #!/bin/sh
        d=$HOME/data
        date=`date '+%Y-%m-%d_'`
        for f in `find $d -name '*.fid' -a -mtime -1 | \
           grep -v '20[0-9][0-9]-'`
        do
          base=`basename $f`
          dir=`dirname $f`
          mv $f $dir/${date}$base
        done
```

The "find" command finds all files that have been created or modified within the last day (assuming the FIDs are all stored in "~/data" or a subdirectory of "~/data"), and which are named "*.fid", and with the "grep -v" command we remove files that already have a string "2000-" up to "2099-" in their name (so we don't rename a file twice). "$date" is set to the date in ISO-8601 format, with an underscore character appended. On November 24th, 2003 a FID file "fid1d.fid" would be renamed to "2003-11-24_fid1d.fid". Within a given directory, "ls" would now list files by date, and within a given date files are shown in alphabetical order.

If you prefer date sorting WITHIN an alphabetical listing, you could insert the date between the body of the name and the ".fid" extension:

```
        #!/bin/sh
        d=$HOME/data
        date=`date '+_%Y-%m-%d'`
        for f in `find $d -name '*.fid' -a -mtime -1 | \
           grep -v '20[0-9][0-9]-'`
        do
          base=`basename $f '.fid'`
          dir=`dirname $f`
          mv $f $dir/$base$date.fid
        done
```

or you could use "sed" for the filename modification by changing the commands in the loop to

```
        new=`echo $f | sed "s/\.fid$/$date.fid/"`
        mv $f $new
```

Store this script (let's call it "renamefids") in your "~/bin" and make sure it is executable:

```
        chmod +x ~/bin/renamefids
```

To call such a script every day, one minute before midnight, edit your "crontab" file with

```
        setenv EDITOR dtpad
        crontab -e
```

and add the following line

```
        23 59 * * * $HOME/bin/renamefids
```

As long as this script is run once per day only, there is no need to protect it against creating duplicate filenames (but you may need to be careful when testing the script!).

Compared to parameter-based time stamping, these mechanisms permit comparing dates (or selecting files acquired on a given date) directly by looking at file names - it is not necessary to retrieve the data to see the time stamps.

This "cron" approach is not needed in VnmrJ: the current version of VnmrJ has new "Svfname" and "autoname" options that permit building such time stamps into the file name when the file is saved, see Agilent MR News 2003-10-17.

[Agilent MR News 2003-11-23]

2004-08-21:

UNIX HINTS - WHEN TO USE "cp" WITH THE "-p" OPTION:

Users occasionally get confused about UNIX ownerships and the role of "cp" and other UNIX commands in the ownership/permission settings. Before looking into "cp -p", let us first consider some fundamental differences between the UNIX "cp" and "mv" commands. To understand what is happening we first must understand how UNIX stores files:

- a directory entry (as shown with "ls" or "ls -i") consists of a filename and an "i-Node" number, which can be considered the "anchor point" of the file itself.
- the i-Node contains information about the file type, the protection bits, a UID/GID (the ID of the user who created the file), the file's creation and last modification date, and pointers to the file's contents which are stored on separate blocks (see chapter 5 "Files, Permissions, and Owners" in the VnmrJ 1.1D manual "Solaris Installation and Administration", or the equivalent chapter in the previous "System Administration" manual).

When you use "mv" to move a file between directories, the old directory entry is erased, and a new directory entry is created in the specified place, pointing to the same i-Node - the i-Node itself and the file contents are NOT touched at all (at least as long as you move a file within the same UFS file system). "cp" is different: it also creates a new directory entry, but then it creates a COPY of the i-Node and the underlying data, and by default, "cp" will then set the file's ownership to the UID and GID of the user who called "cp", and the file's creation / modification dates will be set to the time when "cp" was executed. In other words: you lose the information about the file's original creator and the creation and modification dates. Moreover, because of occasional problems with the write permission in the target directory (you should always IMPORT, RATHER THAN EXPORT FILES!) some users tend to call "cp" as root in order to avoid permission issues. While this indeed avoids permission problems, it has the VERY UNDESIRABLE effect of leaving behind files that are owned by root, which is likely to cause problems when a user or a program tries to modify or remove these files later. When you call "cp" as an ordinary user you may still lose the file's creation / modification date, which often is valuable information: think of a "maclib" with possibly hundreds if files - being able to do "ls -lt" or "ls -ltr" for a (reverse) time-sorted listing often provides useful hints about when a macro was last modified, etc.

"cp -p" (or "cp -rp" in the case of a recursive copy) TRIES TO preserve the permissions, the modification date, and the user & group ownerships. The latter two involve calling "chown", hence are only effective when using "cp -p" as root (other users can't call "chown"). The most prominent case where it is STRONGLY recommended to use the "-p" option is when copying as root, see above (you should NEVER do things as root, unless this is an absolute necessity). For the other cases it is often (or even mostly) still desirable to use the "-p" option in order not to lose the file modification dates etc.; Note that the "-p" option will bypass a user's "umask" setting, see "man umask".

There are cases where one needs to be root to copy files, e.g., when copying from a remotely mounted disk, from a file system with restrictive permission settings and possibly different user definitions (UID/GID). You may still want to use the "-p" option to preserve file creation / modification dates, but in the end you will then need to fix the ownership of the target files. This can easily be done, both for the UID and the GID at the same time - e.g.:

```
        chown -R vnmr1:nmr file1 file2 ...
```

If you have used "cp -r", but then find that you do NOT want to preserve the original modification date(s), you can use "touch" (see "man touch"):

```
        touch file_name
```

or (as "touch" does not have a recursive option):

```
        find directory_name(s) -exec touch {} \;
```

[Agilent MR News 2004-08-21]

2005-02-13:

MAKING MACROS AND SHELL CALLS WORK IN BACKGROUND:

Occasionally, users are confronted with macros that work OK when called in foreground, such as directly, via the command line, or as part of conditional, i.e., "wexp", "wnt", etc. processing for an acquisition in the current ("foreground") experiment. However, when called in background - e.g., as part of an automation run, or with conditional processing in an experiment OTHER THAN the current one - the very same macros fail. What complicates diagnostics for such cases is the fact that any error message may only be seen in the console window, i.e., is easily overlooked or lost. Most of these cases are caused by the following issues:

- When doing an acquisition in the current experiment (i.e., in foreground), the acquisition processes "hand over" the data to the running copy of VnmrJ or VNMR, which then stores the FID and perform the specified conditional processing. In the case of automation or background acquisition (i.e., in an experiment other than the current one), the automation or acquisition processes "fork" a "background" copy of VNMR ("/vnmr/bin/Vnmrbg" in the case of VnmrJ), which then does the data saving and conditional processing. That background program "tries to" run as the respective user - but typically, that copy is running in a different working directory - and consequently, any macro that is referring to relative file paths is likely to fail. Hence,
  - use full (absolute) path names wherever possible, e.g.:
    ```
    userdir+'/data/studies/'+...
    ```
  - if this is unwieldy, make sure you do a "cd" with an absolute path, e.g., "cd(userdir+'/data/studies')" before using relative path names.

  Making assumptions about the working directory is never a good idea!
- For the very same reason, UNIX environment variables such as "$HOME" which work OK in foreground VNMR may NOT be defined in background operation. DON'T use "$HOME" in macros; rather use constructs such as
  ```
  cd(userdir+'/..')
  ```
  or construct the equivalent of "$HOME" using
  ```
  shell('dirname',userdir):$home
  ```
  and then use "$home", e.g.:
  ```
  cd($home+'/mydata')
  ```
  Note that "shell" calls from VNMR use a Bourne shell ("sh", or "bash" in Linux), i.e., "~" cannot be used in lieu of "$HOME" either. In the case of automation, "$HOME" MAY be defined, but may point to the home directory of the user who started the automation run: another reason NOT to use "$HOME"!
- The acquisition process runs as root. In the case of a background acquisition in an experiment file that is located on an NFS-mounted disk this may cause the forked background copy of VNMR to fail to save the data: by default (for security reasons), the NFS protocol causes root to appear as "nobody" on the remote disk. This means that a FID can only be saved into a directory with global write access - and for good reasons experiment directories are writable only for the owner. The only solution for this issue is to export file systems with shared / remote home directories with root permission, see Agilent MR News 1998-05-07.

[Agilent MR News 2005-02-13]

2005-03-20:

MIGRATING BETWEEN SOLARIS, REDHAT LINUX, MACOS X:

Now that we have VnmrJ running on Solaris, RedHat Linux and MacOS X you may wonder whether your macros, parameter panels, pulse sequences, shell scripts, etc. are compatible on all these platforms. This is a complex subject, and a comprehensive discussion is beyond the scope of a single article in Agilent MR News - however, a few hints and pointers may be helpful at this stage; we will add more to this in future issues, as questions arise. Let's start with a few basic things:

- Binary executables are NOT transportable! Typical examples are compiled C programs and compiled C pulse sequences (from "seqlib") - such modules must be assumed to be specific to both the hardware AND the operating system on / under which they were compiled. NEVER try executing a SPARC / Solaris executable under Linux. Even within a given platform, binary executables are specific to the operating system, i.e., a SPARC / Solaris executable CANNOT be assumed to run under SPARC / Linux; similarly, a SPARC / Solaris executable will NOT work under Solaris on Intel ("x86") hardware, etc.
- On the other hand, VnmrJ maintains platform independence with binary NMR DATA FILES, i.e., the FID format is independent of the host platform - see also Agilent MR News 2005-01-22).
- Also, text files such as macros, VNMR parameter files, pulse sequence source code, XML files (such as VnmrJ panel layouts, etc.) are independent of the platform and should be valid on all environments supported by VnmrJ, even though there are a few quirks in this picture, see below.

The key point is that your macros, VnmrJ panel layouts and menu definitions, parameter sets, etc. should be transportable between Solaris, RedHat Linux and MacOS X (Darwin): the MAGICAL interpreter ("engine") is identical on all three supported platforms. Unfortunately, there are also a few traps that we need to deal with:

- even though they are pure ASCII, shell scripts MAY be platform-specific - making them platform-independent may require special efforts, see below;
- UNIX system administration and other files may be in different locations, may have platform-specific contents and format.
- this not only affects shell scripts, but it may also affect macros that use "shell" calls. Note that we have not verified that every standard VnmrJ macro works on all three platforms mentioned in the title, but that certainly is the goal; if you encounter platform-specific macro problems, please report them as a bug (VnmrJ for MacOS X is for processing only, hence we have not looked at acquisition-related macros).

The key issue is with more complex and maybe very specific "shell" calls in macros - and DEFINITELY in many shell scripts. The main reasons for such compatibility issues are:

- Solaris is based on SVR4 UNIX, while Linux and MacOS X are based on BSD UNIX (as was SunOS up to version 4.1.x), which implies that certain UNIX commands may not be present on all platforms, may have different names, or may use a platform-specific set of arguments;
- the UNIX command path definitions may be different. For instance, under Solaris it is customary to have "." (the working directory) as part of the command path for users other than root - under Linux this is normally NOT the case, and out-of-path executables in the working directory must be called WITH the path: you must use "./extract" for the installation of user library contributions, e.g.:

  ```
  cd /vnmr/userlib
  ./extract maclib/loadbiopack
  ```

  Note that this ALSO works under Solaris!
- In Solaris, the most commonly used shells are "csh" (the C shell) and "sh" (the Bourne shell); in Linux and MacOS X, the newer "tcsh" is used in lieu of the C shell, and "bash" (the "Bourne again shell") is used in lieu of "sh". These replacements are designed to be full super-sets of their "csh" and "sh" counterparts (i.e., they support all features of the older / simpler equivalents and have extra facilities added). Note that if you specify "sh" under Linux, this automatically executes a "bash"! This substitution should be harmless, but there may be complications that we haven't run into yet.
- What may complicate the problem is the fact that what you think is a UNIX executable very often actually is a "shell built-in command": most shells use built-in functionalities for commands such as "pwd", "ls", "echo", even commands such as "time", etc. - and these built-in functions typically DIFFER from the functionality of the standard UNIX commands that you find described in UNIX textbooks!

As stated above, we can't give you a comprehensive recipe here - but here are a few key hints, based on the editor's limited experience with shell scripts under Linux and Darwin (MacOS X):

- ALWAYS EXPLICITLY specify the shell type in the header line! this is best done with a line

  ```
  #!/bin/sh
  ```

  for Bourne / "bash" shell scripts, and with

  ```
  #!/bin/csh
  ```

  for C shell or "tcsh" scripts (both rarely used). Note that both "tcsh" and "bash" are available in Solaris 9, but may NOT be present in older Solaris versions - better stick with the above two choices (the bulk of the shell scripts uses "sh" only anyway).
- Avoid shell built-in functions such as "echo" and "pwd". This can be achieved in two ways:
  - call commands with their absolute path, e.g., use

    ```
    echo=/bin/echo
    pwd=/bin/pwd
    ```

    and thereafter use "$echo" and "$pwd" in lieu of "echo" and "pwd".
  - For more platform-independence you may want to use

    ```
    echo=`which echo`
    pwd=`which pwd`
    ```

    Note that the "which" construct does NOT work under "tcsh" in SOME environments, but with "sh" and "bash" it is OK.
- If you are a "nawk" / "awk" addict (such as the editor): note that while in

Solaris "nawk" is MUCH more user-friendly than "awk" in Solaris - but under Linux and MacOS X only "awk" exists - with the "nawk" feature set, though.

- For longer shell scripts it seems best to define a short "compatibility header section" such as (in "sh" / "bash" syntax)

```
        echo=`which echo`
        time=`which time`
        pwd=`which pwd`
        if [ `uname` = SunOS ]; then
          awk=nawk
        elif [ `uname` = Linux ]; then
          echo="$echo -e"
          time="$time -p"
          awk=awk
        elif [ `uname` = Darwin ]; then
          time="$time -p"
          awk=awk
        else
          awk=awk
        fi
```

  And for the remainder of the script use "$echo", "$time", "$pwd" and "$awk" in lieu of "echo", "time", "pwd", "awk", and "nawk" throughout. The "echo" part in the above definition permits using "\c" for suppressing the newline character at the end of the output.

Note that so far VnmrJ only supports RedHat Enterprise Linux 3.0; if we were to discuss RedHat vs. SuSe (or Debian) Linux, this would open yet another can of worms!

[Agilent MR News 2005-03-20]

2005-04-24:

HINTS ON DESIGNING SHELL START-UP SCRIPTS - AND A WARNING:

Whenever you log into UNIX, open a shell window (which starts your default shell, as defined in "/etc/passwd"), and also usually whenever you start a shell script, the relevant shell interpreter runs a start-up script, typically located in the user's home directory. The following table lists the start-up scripts for the various shell types, in the order of execution. The general rule is that such start-up scripts are optional; the list below shows these start-up scripts for a number of popular shells, in the order in which they are executed - names in parentheses indicate scripts that are only looked up if the preceding file is not present:

Shell Type	Shell Command	Start-up Script
C shell	/bin/csh	~/.cshrc
Bourne shell	/bin/sh	/etc/profile -> ~/.profile
Bourne again shell	/bin/bash	/etc/profile -> ~/.bash_profile (-> ~/.bash_login (-> ~/.profile))
Korn shell	/bin/ksh	/etc/profile -> ~/.profile
Enhanced C shell	/bin/tcsh	/etc/csh.cshrc -> ~/.tcshrc (-> ~/.cshrc)

The execution of these start-up scripts is invisible and automatic, often not even known to the user - nevertheless they are very useful, particularly with the user's interactive shells. For login shells (i.e., the user's initial shell environment after a "login", "su -", "rlogin" or "telnet") special rules apply (extra start-up scripts such as "~/.login" are executed), see the corresponding "man" command, e.g., "man csh". An article on customizing UNIX accounts using such start-up scripts was posted in Agilent MR News 2003-08-11. Remember that the "~" naming shown above works in most shell environments, but is NOT valid in the Bourne shell (there you need to use "$HOME" instead).

There is one exotic and unexpected call of "~/.cshrc": if your default shell is a C shell ("/bin/csh", the default with VnmrJ / VNMR accounts in Solaris), and you execute a

```
        su some_user -c some_command
```

as used as part of the install scripts of the User Library contributions "psglib/BioPack" and Chempack ("chempack/CP", "chempack/CP3"), then YOUR script "~/.cshrc" is executed, NOT the one of the "target user"!

The standard setup for VnmrJ / VNMR users includes a basic set of "dot files", including a ".cshrc" and a ".login", both of which we distribute in a directory "/vnmr/user_templates" - they are installed via "makeuser" or when updating users via "vnmrj adm". The supplied version of ".cshrc" unfortunately suffers from several deficiencies, see the bug report "dotcshrc.j1101" below. As outlined in the bug report, a properly designed ".cshrc" should contain a "separator" statement as follows:

```
        if ($?USER == 0 || $?prompt == 0) then
          #statements for shell scripts ONLY may be added here
          exit
        endif
```

ABOVE this "if" statement you should have all lines that are GENERALLY APPLICABLE, such as

```
        umask 022
        limit coredumpsize 0
```

(the second line would suppress core dumps from C programs that crash in a C shell environment). INSIDE the above "if" statement you should add those lines which apply to shell scripts EXCLUSIVELY (the editor's file has that segment empty, as shown above). BELOW that statement you should add those lines which should be run for INTERACTIVE SHELLS ONLY. This includes

- alias definitions (undesirable in shell scripts)
- history-related lines (not used in shell scripts)
- prompt definition (not used in shell scripts)
- any other function that you want to call for interactive shells / shell windows.

One example for the very last point would be a "cd" command to direct a user's interactive shells to a specific location OTHER THAN the home directory, e.g., to have a user produce output in some specific location (e.g., a "directory of the day, week, or month"), in order to limit clutter in the user's home directory, particularly in shared accounts. This may sound like an exotic idea, but it may have its use in specific situations. There is a reason why we pick this example, of course: one user implemented a "cd some_directory" call in a user's "~/.cshrc", WITHOUT the above separator - which caused serious problems: as the "cd" then also was executed as part of any C shell script that this user is calling, it gets such shell scripts confused about the working directory, causing aborts and other strange problems - among other things, the BioPack and Chempack install scripts may run into endless loops when trying to run "su vnmr1 -c some_script" for the administrative part of a local installation.

In the case of C shell scripts (which are exceptions - most of the shell scripts that we use are Bourne shell scripts) one could change the script header line from

```
        #!/bin/csh
```

to

```
        #!/bin/csh -f
```

to SUPPRESS the execution of "~/.cshrc", hence providing a moderately faster execution, and avoiding complications from mis-configured start-up files. The same is available with "tcsh" scripts - however, this mechanism does NOT exist for other shells, plus, there are typically DESIRABLE parts in a user's "~/.cshrc" (such as the "umask" command) which you WANT to be executed - hence using the "-f" option for "csh" and "tcsh" scripts is usually NOT recommended as a general recipe.

[Agilent MR News 2005-04-24]

2005-04-29:

HINTS ON DESIGNING SHELL START-UP SCRIPTS - FOLLOW-UP:

Charlie Fry (Univ. Wisconsin-Madison) pointed out a couple things in the article about shell start-up scripts in the last issue (Agilent MR News 2005-04-24) and in the preceding article in Agilent MR News 2003-08-11 that may cause confusion or may not be entirely clear - hence these clarifications:

- Keep in mind that given the variety of shell environments and at the same time the complexity of each one of the shell interpreters mentioned in these articles, none of them can be discussed in a comprehensive manner within the scope of Agilent MR News - for complete details please refer to the UNIX "man" pages (e.g., "man csh"), and to UNIX literature.
- In a UNIX context, the term "path" is used in two ways:
    - the "path" to a file, or file path, describing where in a file system hierarchy a particular file is stored, or through which "apparent path" a file can be accessed. The latter case may include symbolic links, e.g., a file can be accessed as "/vnmr/psglib/s2pul.c", but in reality it may be stored in "/export/home/vnmrj1.1d/seqlib/s2pul.c".
    - more specifically, the "path" variable in the UNIX shell describes the UNIX "command path", i.e., a list of directories in which the shell looks for a given UNIX command such as "ls", "cat", "awk", etc. - in

the order of their priority (i.e., specifying the sequence in which they are searched for a given command). The command path may include "." (STRONGLY discouraged for root, for other users "." is usually placed near the (right) end of the path (low priority), for security reasons. Note that if you call UNIX commands by absolute file path (e.g., "/bin/ls"), the UNIX command path ("path" variable) is NOT used.

When we refer to a command "cd some_directory_path", this changes the WORKING DIRECTORY. It MAY affect the command path (in case this includes ".", now pointing to a different directory), but we NEVER refer to the working directory as "path" (unless we were to state "path to the working directory"). In Agilent MR News 2003-08-11 we discussed issues relating to the UNIX command path. The warning in the article in the last issue (Varian MR News 2005-04-24) referred to changing the working directory in a shell start-up script, NOT the command path - it's this change to the working directory which may confuse / disrupt the execution of shell scripts - much more than alterations to the UNIX command path.

- It is usually OK to expand the command path (certainly for users other than root), as long as such expansions are added NEAR THE (right / low priority) END of the path. If you PREPEND directories to the command path, this may cause non-standard (e.g., BSD) utilities to supersede standard (SVR4) commands - adding directories to the BEGINNING of the command path should only be done by people who understand the consequences of such a step.
- in the standard Solaris environment (as set up by the VnmrJ / VNMR utility "makeuser"), the user's default shell is "/bin/csh". In this case, the user's "~/.cshrc" is executed WHENEVER a "csh" is launched - this includes opening new shell windows. For login shells (e.g., the shell that is launched after the user's initial login, but also after a "su - username", "rlogin", "telnet"), "~/.cshrc" is executed FIRST, FOLLOWED BY "~/.login". As "~/.cshrc" ALWAYS precedes the execution of "~/.login". Changes and settings made in "~/.login" are "inherited" by any subsequent shell calls up to a logout, so in principle changes in "~/.login" OR "~/.cshrc" would seem to be equivalent (while defining aliases on BOTH files is definitely overkill and may cause confusion), but if you consider making changes to the "alias", "prompt", etc. definitions, then "~/.cshrc" is certainly more practical, as these changes become valid immediately when opening a new shell window or calling a C shell using "csh", while changes in "~/.login" require you to log out and back in again to see the effect. We therefore recommend placing such definitions in "~/.cshrc" rather than in "~/.login".

  In running C shells, you can force the execution of "~/.cshrc" by typing

  ```
  source ~/.cshrc
  ```

  Typically, ".cshrc" only ADDS aliases and definitions; if ever you want to "start from scratch" (i.e., remove all existing alias definitions first) without logging out, you can do that with

  ```
  unalias -a
  source ~/.cshrc
  ```
- ALL shell environments know "$HOME" - also the C shell. We usually use the C shell shortcut "~" in lieu of "$HOME" - this is also known to the other shell environments EXCEPT for the Bourne shell, in which ONLY "$HOME" may be used to indicate the user's home directory. ALL shells EXCEPT the Bourne shell also recognize "~username" as the specified user's home directory. In the Bourne shell, this particular functionality is NOT available. To evaluate another user's home directory in a Bourne shell ("sh") environment you could use the following - somewhat clumsy - construct instead:

  ```
  userhome=`echo 'echo ~username' | csh -f`
  ```

  or

  ```
  userhome=`csh -c 'echo ~username'`
  ```

  i.e., use an embedded C shell call within a Bourne shell. Of course, one could also read "/etc/passwd" to evaluate a home directory, e.g.:

  ```
  vnmr1home=`cat /etc/passwd | nawk -F: '{if ($1 == "vnmr1") print $6}'`
  ```
- In the last issue (Agilent MR News 2005-04-24) we recommended a construct

  ```
  if ($?USER == 0 || $?prompt == 0) then
    #statements for shell scripts ONLY may be added here
    exit
  endif
  ```

  as a logical separator between general definitions (above), lines JUST for scripts (inside this "if" statement) and lines that are to be executed for interactive shells EXCLUSIVELY (after the "endif"). Note that shells are - unlike many compiled languages such as C - somewhat picky about the placement of blank spaces: you MUST have spaces around the "==" on the above "if" line, otherwise the construct fails. You may have extra spaces inside the parentheses, and the spaces around the "||" are optional, too, but the spaces around the "==" are MANDATORY. You could also use

```
     if (! $?USER || ! $?prompt) then
```

or (probably better / clearer about logical precedences)

```
     if ((! $?USER) || (! $?prompt)) then
```

instead - here, the spaces after the "!" (logical negation) are again MANDATORY. If you don't anticipate ever having anything but "exit" inside the "if" statement, you could shorten this to one line:

```
     if ($?USER == 0 || $?prompt == 0) exit
```

or

```
     if (! $?USER || ! $?prompt) exit
```

or

```
     if ((! $?USER) || (! $?prompt)) exit
```

[Agilent MR News 2005-04-29]

2005-05-16:

AN UNEXPECTED "tar" COMPATIBILITY ISSUE:

In a UNIX environment, "tar" is used almost universally to collect multiple files or directory trees into a single file which can then be compressed for efficient (space-preserving) storage / archiving, or for efficient transfers over networks. A typical command line for creating a compressed "tar" archive in a single step might be

```
        tar cf - files_to_archive | compress > archive_name.tar.Z
```

or (using the more efficient, but slower "gzip" in lieu of "compress"):

```
        tar cf - files_to_archive | gzip > archive_name.tgz
```

In the process of updating Chempack 3.1 (the last version was put on-line on 2005-05-13), Krish Krishnamurthy (Eli Lilly, Indianapolis) and the librarian ran into an irritating compatibility issue with "tar".

The Solaris manual entry to "tar" states that within that command, file path names can be up to 256 characters long, whereby that name can be composed of a directory part of up to 155 characters and a "basename" (the actual filename) of up to 100 characters. The "E" option permits extending these limits - but in the countless instances in which the editor has used "tar" over the past 20 years, it never seemed necessary to use that option.

Now, Krish created a "tar" archive for parts of Chempack 3.1. This was done on a Linux system. The librarian received this archive (compressed) on his Sun Blade 150/650 running Solaris 9. The file was expanded, and from a "tar" listing ("tar tvf") the contents of the archive appeared to be complete and intact - however, trying to extract the archive led to a directory checksum error. The file checksum was OK, and the same archive files extracted OK under Linux, hence data corruption problems could excluded. Further investigations revealed the following:

- The problem occurs when the complete pathname for any file to be archived is longer than around 86 characters.
- The issue was observed for "tar" archives created under RedHat Enterprise Linux 3.0 or under RedHat Linux 9 (running within VMware), and trying to extract these archives under Solaris 9.
- These archives can be extracted without problem under Linux - the issue so far seems to be specific to extracting Linux "tar" archives under Solaris - while, as shown above, the Solaris "tar" command can handle much longer filenames without a problem (this was verified in order to exclude the possibility of a bug in the Solaris 9 "tar" command).
- Using the "--posix" option when creating the archive under Linux does NOT help (note that the Linux "tar" does not seem to have an "E" option).

The editor verified the problem also under MacOS X (10.4):

- The Linux archive could be extracted WITHOUT PROBLEM under MacOS X
- A "tar" archive was created from the very same data under MacOS X: under Solaris 9, "tar tf" worked without error, but extracting the data caused the same directory checksum error.
- Judging from the "man" information, MacOS X uses the GNU version of "tar", as does Linux. It appears that a POSIX type archive is created by default. The command

```
     tar cof ...
```

or

```
     tar -c -o -f ...
```

or

```
tar -c --old-archive -f ...
```

or

```
tar -c --portability -f ...
```

(note the double dash "--" in front of "verbose" arguments) enforces a "V7" style rather than a POSIX compliant archive; however, with this, creating the archive from these data fails with error messages indicating

```
file name is too long (max: 99); not dumped
```

Note that on the other hand, "tar" archives from the same data (and even "tar" archives with substantially longer pathnames) created under Solaris 9 CAN be extracted both under Linux AND under MacOS X (10.4) without problems, so exporting "tar" archives FROM Solaris appears to be OK.

Conclusion: assuming that Solaris 9 uses the correct SVR4 coding of "tar", it appears that the GNU version of "tar" (and hence RedHat Enterprise Linux 3, RedHat Linux 9 and MacOS X) may create archives which are not fully SVR4 (i.e., Solaris) compliant, while SVR4 (and hence Solaris) "tar" archives apparently are always readable, also with the GNU version of "tar". Our investigations on this problem are not comprehensive and may not be completely accurate yet, but we still STRONGLY recommend observing the following precautions for creating "tar" archives under MacOS X and Linux:

- As long as "tar" archives are created and extracted on the SAME software platform, "tar" seems to function within the limitations given in the UNIX manuals.
- When transferring "tar" archives between software platforms (i.e., between different operating environments), or if you anticipate (or cannot exclude) that some day you need to extract your archives on a different platform, you should avoid pathnames of more than 85 characters in ANY SUBFILE.
- We strongly recommend verifying that "tar" archives can be extracted on the target platform - looking at a "tar" catalog ("tar tvf") and/or extracting the archive on the originating platform may NOT be good enough!

To check an archive directory for excessive pathname lengths open a UNIX shell window (either with a standard width of 80 characters, or by adjusting the width to 80 - 85 characters) and use the command

```
tar cf - files_to_archive | tar tf - | more
```

to see whether there are any extra-long output lines that wrap around.

Traditionally, MacOS X users would rather create "StuffIt" ("*.sit") or "zip" archives rather than compressed "tar" archives. In a mixed platform environment, "zip" archives are the better choice, as "zip" and "unzip" are now almost universally available. However, for reasons of compatibility with existing data archives it may sometimes be preferable or a requirement to stick with (compressed or "gzipped") "tar" archives. Also, earlier versions of Solaris (Solaris 7 and older) did not include the "zip" / "unzip" commands.

[Agilent MR News 2005-05-16]

2005-05-24:

GNU VS. SVR4 "tar" INCOMPATIBILITY ISSUE - A WORKAROUND:

In the last issue (Agilent MR News 2005-05-16) we reported an irritating incompatibility issue with "tar" archives containing very long file names. While it isn't up to us to decide whether the GNU "tar" or the SVR4 / Solaris version is "right", we primarily wanted to give you a procedure that should help avoiding potential problems from unreadable archives - a "preventive workaround".

However, there may be cases where you are already "stuck" with unreadable archive files on a Solaris system, and you may not be in a position to get revised versions that can be read using the SVR4 version of "tar" - this is not the end of the world: you do not need to purchase a Linux-based PC to resolve the problem! As Jonathan Moody (MIR Preclinical Services, Ann Arbor, MI) and Dimitris Argyropoulos (Varian Germany) pointed out, the GNU version of "tar" is available from the Sun Freeware Web site at

http://www.sunfreeware.com/

For Solaris 9, version 1.15.1 is available as compiled object that installs into "/usr/local"; you apparently also need to download and install the GNU "libiconv" from the same source to make it work. Thanks, Jon and Dimitris, for this information!

[Agilent MR News 2005-05-24]

2005-06-07:

DISPLAYING TEXT FROM WITHIN VnmrJ MACROS:

In "classic" VNMR, you could use "cat", "shell", or "write('text','...')" calls, along with commands such "dg", "dgs", "da", etc. to display text in the bottom (text / Tcl-dg) window. If you were using Tcl-dg, the bottom window would automatically switch to the text panel when such commands were called. This had the advantage that such text output would be visible automatically - but it also had the disadvantage of disrupting the "Tcl-dg workflow", and such panel switching could also happen inadvertently, e.g., when a programmer or user forgot to capture the output from "shell" calls with a dummy return argument, such as in

```
        shell('rm -f',$files):$dummy
```

(all "shell" calls return at least an empty line).

In VnmrJ, text output does NOT switch to the text panel, which maintains the workflow and avoids disruptive panel changes. On the other hand, occasionally one would like to give text feedback to the user. The commands

```
        write('line3','...')
```

or

```
        write('error','...')
```

can be used for single lines of output which is displayed above the command line and in the message widget at the bottom of the VnmrJ window: both are identical, except that the second one also issues a beep, and the message widget will display the text in red. There are several potential issues with these mechanisms:

- the message lines above the commend line may be hidden
- subsequent messages may erase the single line message display in the widget at the bottom of the window. Of course, you can always open the message panel - but that assumes that you know that there is something to look up!

As an alternative, one can use "banner" to display well visible text messages from within a macro - the BioPack installation utilities and BioPack itself use that option extensively. However, also this option has its limitations:

- it erases the contents of the graphics window
- it is hardly usable for large amounts of text
- text is centered - displaying differently formatted, left-aligned, or multi-column text is tricky (it requires adding blanks to shorter lines, such that all lines have exactly the same length).

One user was looking for way to present the output of "da" from within a macro in VnmrJ. One possibility to do this is by first writing the output to a text file, and then to use a pop-up window to display the information, e.g.:

```
        $txtfile=curexp+'/tmptext' $dummy=''
        printon da printoff($txtfile)
        shell('xterm -geometry 80x60 -sb -e less',$txtfile):$dummy
        shell('rm -f',$txtfile):$dummy
```

In this construct we use an "xterm" (basic X11 terminal window) of the specified geometry (X and Y size in characters), with a scroll bar ("-sb"). In this window we execute the command "less" (see the article below) to display the text information, and when the window is closed, we delete the temporary text file. Note that with this construct the macro execution will be paused until the user types "q" to terminate the "less" command. You can use "xterm" in a background call if you want the macro to continue while the text is presented in the pop-up window. In this case, we should of course delete the text in the same shell call, following the execution of the "xterm":

```
        $txtfile=curexp+'/tmptext' $dummy=''
        printon da printoff($txtfile)
        shell('(xterm -geometry 80x60 -sb -e less',$txtfile,
              '; rm -f', $txtfile,') &'):$dummy
```

Note that in "shell" calls, commas separating multiple arguments translate to spaces. Also, the above multi-line layout for a command with several arguments only works in a macro - if you want to test things on the command line, this "shell" call would need to be entered on a single line.

[Agilent MR News 2005-06-07]

UNIX HINT - WHY "less" MAY BE MORE THAN "more":

The most basic UNIX command to display is "cat". This prints out the entire contents of a text file in the current terminal window. In a scrolling window, you may then need to scroll back to see the top of the file. It may easily happen that the size of the text file exceeds the capacity of the text buffer of the given shell window - and the top of the text is lost to the viewer.

To avoid the buffer size problem, and to start by presenting the top of large amounts of text, most of us use "more", e.g.:

```
        more text_file
```

or

```
        some_command | more
```

Now, you can hit the space bar to cause "more" to display the next page, you can type "q" to exit, or "b" to move back to the previous page. "b" ONLY works if you have NOT reached the end of the text, and it also does not work for piped input (second example above). If the displayed text is shorter than one page, "more" behaves exactly the same way as "cat". In the construct discussed in the article above, "more" would NOT be usable (at least not without extra, auxiliary constructs), as

- for short texts, "more" would exit immediately after showing the text, and the window would close before the user has a chance to read the text;
- similarly, once the end of a multi-page text is reached, "more" exits, the window would close, and the user would not be able to read the last page.

That's exactly where "less" is useful: "less" is a substitute for "more", with ADDED features (so, indeed, "less" is MORE than "more"!): one of the extra features is that in the case of small texts, "less" STILL remains in the interactive mode and expects a "q" to be typed to terminate execution. For complete information type "man less" in a UNIX shell window. One may indeed consider using "less" in lieu of "more" all the time (it may take a while to get used to this change, though!), by adding a line

```
        alias more less
```

to your "~/.cshrc".

[Agilent MR News 2005-06-07]

2005-10-12:

SHELL SCRIPT HINT - HANDLING ERRORS, QUERYING RETURN CODES:

Different basic types of errors may occur in UNIX shell scripts, causing script failures / aborts, or also just error messages:

- basic shell script syntax errors (such as a missing "fi" terminating an "if" or the like) may cause a shell script to abort even before that part of the program is executed. In general, shell script syntax errors cause a shell script to abort with an error message.
- if you call a non-existent program / command name (often a typo), a script will abort with an error message reporting "command not found".
- if a command / program that you call within a shell script encounters an error or aborts, etc., the shell script itself will CONTINUE, irrespective of whether you "capture" the error message or not.

In the first two cases, you will obviously need to debug the shell script. Unknown commands are easy to locate, as the error will mention the name of the program that was not found. Errors in shell script control structures (such as "if" ... "then" ... "else" ... "fi") may be tricky to locate, as for instance a missing "fi" (the "sh" / Bourne shell equivalent to "endif" in VnmrJ macros) may only be detected when reaching the very end of the script. The only real remedy here is a consistent and persistent indentation scheme, e.g.:

- always make sure corresponding "if", "else", and "fi" tokens are at identical indentation levels,
- use two or more blanks per indentation, such that the indentation levels are easily recognizable also in longer scripts.

But what about errors that occur in UNIX commands called by the script? There are various options that one might consider:

- you may want to ignore an error message and continue execution of the script anyway. In such a case it is usually desirable to suppress the error message, as it may irritate the user. This can be achieved by redirecting the error output to "/dev/null", e.g.:

  ```
      rm tmpfile 2>/dev/null
  ```

  This removes the file "tmpfile" if it exists, but does not produce an error if it doesn't exist. That's actually not a good example, as the command "rm" has a "-f" option for that exact purpose:

  ```
      rm -f tmpfile
  ```

  will do the same as the above.
- You could of course also direct the error output into a file and then use the information in that file for further decisions / processing:

  ```
      rm tmpfile 2>errorlog
  ```

  or you can combine standard and error output into a single file:

  ```
      some_command >logfile 2>&1
  ```

  where the "2>&1" adds the error output (designator 2) to the standard output (designator 1).

- There are cases where a program may NOT produce a suitable error message, but simply "exit with a non-zero return status" (or "exit status"). The return status for a given command is explained in the UNIX "man" pages. In C programming terms, this means that the program exits with a "return(1)" call (possibly with other non-zero arguments). The default return status is 0; the convention is that any non-zero return value indicates an irregular condition. The "problem" with this is that the program return value is not directly visible. However, in a shell script, "$?" indicates the return value of the LAST EXECUTED COMMAND. The syntax would be as follows:

  ```
  some_command
  if [ $? -ne 0 ]; then
    echo "Command \"some_command\" encountered a problem, aborting."
    exit
  fi
  ```

  With certain commands the return value is used not just to indicate an error condition, but also to return the result of the command. One example is the "diff" command (see "man diff"), where

  - return status 0 indicates that the two files that were compared are identical,
  - return status 1 indicates that the files are different, while
  - a return status greater than 1 would indicate an error.

  For example:

  ```
  diff file1 file2 >/dev/null
  if [ $? -eq 0 ]; then
    echo "file1 and file2 are identical"
  elif [ $? -eq 1 ]; then
    echo "file1 and file2 are different"
  else
    echo "An error occurred while comparing file1 and file2, aborting."
    exit
  fi
  ```

  Note that the standard "diff" output (indicating how exactly the files differ from each other, provided they are text files) is discarded in this example - we only use the return status of the command.

The constructs above assume Bourne shell syntax, i.e., you should start such shell scripts with a line

```
#!/bin/sh
```

[Agilent MR News 2005-10-12]

2005-10-18:

SHELL SCRIPT HINT - TEST SYNTAX:

String comparisons are a frequent source for shell script failures, and often hard to debug. Consider the following case: in a shell script (this example is taken from a user library install script) we create a local shell variable that contains the name of the instrument, e.g.,

```
spectrometer=`sed -n '3,3p' $vnmrsystem/vnmrrev`
```

(we extract the third line from "/vnmr/vnmrrev")

```
if [ $spectrometer = inova ]; then
  system='UNITY INOVA'
elif [ $spectrometer = uplus ]; then
  system='UNITYplus'
elif [ $spectrometer = unity ]; then
  system='UNITY'
elif [ $spectrometer = mercplus ]; then
  system='MERCURYplus'
elif [ $spectrometer = mercvx ]; then
  system='MERCURY-Vx'
...
```

Later in the same shell script we want to make decisions based on the value of "$system", e.g.:

```
if [ $system = MERCURYplus ]; then
  ...
```

This looks like a harmless comparison - and it will indeed work - SOMETIMES! There are two potential problems with the above construct:

- if the variable "system" happens to be empty (a notorious case when a variable is set from user input!), at run-time that construct will read

  ```
  if [ = MERCURYplus ]; then
  ```

which of course causes a syntax error. One solution that programmers sometimes resort to is the following modification:

```
if [ x$system = xMERCURYplus ]; then
```

i.e., prepend both sides of the comparison with the character "x", which will not fail with an empty variable; at run-time this translates to

```
if [ x = xMERCURYplus ]; then
```

which is proper shell syntax.

- however, the above "x trick" still fails if the string contains blanks: in the case of "system='UNITY INOVA'", at run-time the test would read

  ```
  if [ xUNITY INOVA = xMERCURYplus ]; then
  ```

  which would give an error

  ```
  test: unknown operator INOVA
  ```

  as the "test" syntax takes the second token as an operator ("=", "!=", etc.)

The real problem with both cases above (empty value or value with blanks) is that such scripts SOMETIMES work, sometimes not - and that can make such programs very hard to debug. The PROPER solution to both cases is in the PERSISTENT use of quotes around strings:

```
        if [ "$system" = "MERCURYplus" ]; then
```

This way, also empty strings and strings with one or multiple blanks will be interpreted as single command line token! Even though the quotes are not strictly required around a fixed string value without blanks, it is good practice to use them on both sides of the comparison operator.

[Agilent MR News 2005-10-18]

2005-10-26:

SHELL SCRIPT HINT - CHECKING FOR DIFFERENCES BETWEEN FILES:

Occasionally one needs to check whether a file has been modified or not, by comparing two versions of the same file. There are various ways to do this in a (Bourne) shell script. One user suggested comparing checksums:

```
        if [ `cksum $file1 | awk '{print $1}'` -ne \
             `cksum $file2 | awk '{print $1}'` ]; then
          #files are different
          ...
```

At first, this recipe looks OK - the "cksum" command (see Agilent MR News 2004-03-22) produces a 32-bit numeric checksum, i.e., the probability that two different files produce the same output is absolutely negligible, i.e., the procedure should really be safe. There is one potential problem with this, though: "cksum" produces an UNSIGNED 32-bit number (0 .. 4294967296), while "awk" and "nawk" work with SIGNED 32-bit numeric precision only (-2147483648 up to 2147483647), and any number outside that range (specifically, 2147483648 up to 4294967296) might be silently truncated to the maximum signed value, 2147483647. The above construct works because it does not treat the value of "$1" as a number (and also the Bourne shell "test" / "[ ]" operator seems to handle large numbers proerly) - nevertheless, it is at least a dangerous approach! The construct

```
        cksum $file1 | awk '{printf("%ld\n",$1)}'
```

does NOT work as expected! In the above case, using "cut" in lieu of "awk" or "nawk" would be slightly more efficient:

```
        if [ `cksum $file1 | cut -f1` -ne `cksum $file2 | cut -f1'` ]; then
          #files are different
          ...
```

One could also think of a different approach, still using "cksum":

```
        if [ "`cksum < $file1`" != "`cksum < $file2`" ]; then
          #files are different
          ...
```

There are various subtle points about that approach:

- We use the "<" (take input from) operator rather than specifying the file directly - this way, the output of "cksum" does NOT include the file name, but just the checksum and the file size in bytes;
- We compare both the checksum AND the file size in bytes, so this makes this approach even safer; note that the output of "cksum" is enclosed in double quotes - the two numeric output values are treated as a single string (numeric, with spaces) - this is necessary in order to not to break the "test" syntax, see Agilent MR News 2005-10-18.
- As we are now comparing strings rather than numbers, we MUST use string comparison operators ("=", "!=") rather than numeric ones ("-eq", "-ne", "-ge", "-gt", "-le", "-lt"), otherwise "test" will cause an error.

- By explicitly handling the numeric output of "cksum" as a string, we avoid potential problems with numeric overflow.

However, it is NOT NECESSARY to use "cksum" to compare two files - it is much simpler to use "diff", see the article in Agilent MR News 2005-10-12: in either case the script needs to read both files completely; "diff" does a bit-by-bit comparison, so it is 100.000% safe - "cksum" also reads the same amount of data, but adds extra CPU load to calculate a checksum that is marginally less secure than the direct comparison. So, the best solution is the one we posted in Agilent MR News 2005-10-12:

```
        diff file1 file2 >/dev/null
        if [ $? -eq 0 ]; then
          echo "file1 and file2 are identical"
        elif [ $? -eq 1 ]; then
          echo "file1 and file2 are different"
        else
          echo "An error occurred while comparing file1 and file2, aborting."
          exit
        fi
```

The real use of "cksum" is NOT where you have two potentially different versions at hand, but rather with FTP sites and file deliveries by e-mail, where it permits comparing the RECEIVED file with the original / server copy by means of the checksum, as a direct comparison is not possible or not feasible in such situations. You could do multiple downloads and compare the downloaded copies - but this still would NOT tell you whether the download procedure itself alters the file, such as when you select the ASCII transfer mode for downloading a binary file by FTP.

[Agilent MR News 2005-10-26]

2005-11-02:

SHELL SCRIPT TRICKS - CHECKING USER INPUT:

As with macros (see Agilent MR News 2004-07-07), the trickiest part in writing interactive shell scripts is often the part that deals with the human input. The problem is that the programmer should NOT expect the user to type EXACTLY and ONLY what is anticipated as a possible / "legal" answer - a user might

- give a blank response (typically indicating the selection of the default option / answer)
- type upper- in lieu of lower case, or vice versa
- respond "yes" in lieu of just "y", or vice versa, etc.
- have unexpected blank characters in the response (these may not be visible to the user!)
- maybe have a minor typo in the response ...

Ideally, a shell script should not just produce an error or even abort in such situations, but be a bit more user-friendly, trying to read / guess the user's intent, wherever possible! It turns out that in many cases this isn't too hard to achieve. Here is one example for a user input construct:

```
        #!/bin/sh
        echo=`which echo`
        if [ `uname` = Linux ]; then
          echo="$echo -e"
        fi
        ...
        $echo "Please give your answer (y|n) [y]: \c"
        read answer
        if [ `$echo "$answer" | grep -ic n` -ne 0 ]; then
          # answer is "n" / "no" or equivalent
          ...
        else
          # answer is (or defaults to) "y" / "yes" or equivalent
          ...
        fi
```

Of course, the question is meant to be a "real" one! Here are a couple explanations on this construct:

- In the header of the (Bourne) shell script we deal with compatibility issues (Solaris vs. Linux vs. MacOS X); we avoid the shell built-in "echo" function by using the absolute command path.
- Under Linux we need the "-e" option in order for "echo" to "understand" the "\c" syntax for the suppression of the linefeed when asking for user

feedback.

- When an expected response is a simple choice (such as "y" vs. "n") it is desirable to list the available options - "(y|n)" in the lines above - as well as the default answer (in square brackets "[ ]" above) that is assumed when the user responds with an empty string by typing "[Return]" only.
- The extra blank between the colon and the "\c" makes it harder for the user to detect extra blanks that (s)he may have typed (we can handle those, see below), but it makes the input dialog more readable overall.
- In the "if" statement that follows we do NOT compare the feedback with an expected response option directly, but we simply feed the user's answer ("$answer") into a "grep -ic" call, looking for a (single) character that is indicative of the NON-DEFAULT response. The "-i" option makes "grep" case-INsensitive. At the same time, the "grep" command will give a non-zero count ("-c" option) even if the specified pattern just matches a SUBstring. In the above case, "n", "N", "no", "NO", "nO", "non", "njet", "NEIN" would ALL be accepted as negative response, while an empty string (i.e., selecting the specified default) , "y", "Y", "yes", "YES", "Ja", "da", "si", "Oui", "OK", "agree" etc. would ALL correctly be interpreted as positive response - isn't that user-friendly??
- Note that "grep -c" (with standard input) puts out a single number (the number of matching lines, 0 or 1 in the above case) - therefore we MUST use a numeric comparison operator in the "test" / "[ ]" call, see also Varian MR News 2005-10-26.

The above "grep" trick is very helpful in checking user input in general, but also in cases where it is not clear whether a value is in upper or lower case. It avoids clumsy constructs such as

```
        if [ "$system" = inova -o "$system" = "Inova" -o "$system" = "INOVA" ]
        then
          ...
```

or alternatively / better, but still clumsy

```
        system=`echo $system | tr '[A-Z]' '[a-z]'`
        if [ "$system" = inova ]; then
          ...
```

[Agilent MR News 2005-11-02]

2005-12-04:

UNIX TRICKS - MANAGING PROCESSES:

In Solaris 8 and older versions, UNIX process management typically involved filtering the output of "ps" through a "grep" call, in order to find specific processes quickly and efficiently. An example:

```
        ps -ef | grep Vnmr
```

This may produce output such as

```
        vnmr1 16112 16110  0 23:00:26 pts/17   0:00 Vnmr -mforeground ...
        vnmr1 16110 16109  0 23:00:26 pts/17   0:00 master Vnmr
        vnmr1 16118   877  0 23:01:10 pts/15   0:00 grep Vnmr
```

In this output you could then extract a process-ID (second column) - typically for the purpose of sending a signal to that process. In the above example you could then use

```
        kill -3 16112
```

to send a "HUP" (hangup) signal, causing it to exit PROPERLY (note the difference to sending a "KILL" signal using "kill -9", which would cause VNMR to exit without writing its buffered data back to the disk, see also Agilent MR News 1998-10-30 and Agilent MR News 1999-12-09).

Interactively, the above procedure may be understandable - in a shell script it is a bit tricky to use. One "trap" is that the "grep" command itself MAY show up in the output (it doesn't always!). There are several ways to avoid this: you could use "ps -e" in lieu of "ps -ef", which limits the output to the "basename" of the executable - in the above case you would obtain

```
        16112 vnmr1 pts/17   0:00 Vnmr
        16110 vnmr1 pts/17   0:00 Vnmr
```

This suppresses the "grep" line, because "ps -e" would only show "grep" without the search pattern argument. On the other hand, the output of "ps -e" is also much less informative, in that (among other things) the command line arguments are not shown. In the above case it is even not clear which one of the two lines refers to the VNMR foreground process. Another method to avoid the complication with "grep" showing up in the process table is first to capture the output of "ps" in a file, then to use "grep" in a second step:

```
        ps -ef > process_table
        grep Vnmr < process_table
```

Here, the "grep" command can't be listed, but you retain the full output of "ps -ef".

Solaris 9 and RedHat Linux have more elegant and more powerful utilities for process management: to find process-IDs you can use the command "pgrep". In the above example, "pgrep Vnmr" would produce

```
        16112
        16110
```

With the "-l" option, you get slightly more output: again in the same example, "pgrep -l Vnmr" yields

```
        16112 Vnmr
        16110 Vnmr
```

"pgrep" can't be used to obtain the full command line arguments - but it has a number of other options, see "man pgrep". A related command, "pkill", can be used to send signals to processes that match a given pattern. For example, "pkill find" would kill ALL processes that have the SUBstring "find" in their executable name. It is probably a good idea to specify the "-x" option as well, and to specify the full command name (without path) - this would capture processes matching the FULL pattern only. For example, "pgrep sh" will find all processes that have "sh" in their name, i.e., "csh", "sh", "vnmrwish", "sdt_shell", etc., while "pgrep -x sh" will ONLY list Bourne shells, "sh". In the case of "pkill", a useful option may be "-n" - this limits the effect to the most recently called instance of the specified process name pattern. For an application of some of these utilities see the article below.

For interactive work, you actually don't need any of the above. In CDE, you can select "Find Process" from the "Desktop Controls" menu in the CDE toolbar. This opens a very powerful, interactive utility (the underlying program is "sdtprocess") that permits searching, filtering and controlling processes, sorting by any column in the output of "ps -ef", including

- process-ID
- process base name
- user (e.g., to focus on your own processes)
- CPU usage (to find CPU hogs)
- memory usage (to find memory hogs)
- start time
- parent processes (to determine process hierarchies / dependencies)
- full UNIX call / command line with arguments etc.

A similar utility is available in RedHat Linux, under "Applications" -> "System Tools" -> "System Monitor".

[Agilent MR News 2005-12-04]

CONTROLLING PROCESSES FROM WITHIN VNMR:

At one site with several systems running VNMR, the VNMR "acqmeter" utility has become so popular that the system administrator decided to add the command "acqmeter" to "~/vnmrsys/maclib/login" (for specific users) or to the system's "/vnmr/maclib/bootup" macro (for all users). This way, whenever a user starts VNMR, the "acqmeter" utility is started automatically.

There is one problem with this approach, though: the "acqmeter" utility is a background process (you would not want "acqmeter" to block the VNMR command line until the user decides to terminate it with "exit" from the pop-up menu!), and so it continues to run even when the user exits from VNMR - and when VNMR is started again, a second instance of "acqmeter" is launched: this way one could easily accumulate a number of "acqmeter" windows that clutter the screen until the user does a full logout or manually closes the extra instances of the utility. To avoid this situation we need to kill "acqmeter" when exiting VNMR. In Solaris 9, this can be done by adding a single line

```
        shell('pkill -x Acqmeter 2>/dev/null &'):$dummy
```

near the top of the macro "/vnmr/maclib/exit" (see the article above for an explanation of the command "pkill"). Note that

- the UNIX executable is "Acqmeter" ("/vnmr/bin/Acqmeter"), not "acqmeter";
- "pkill" will send a termination signal to all processes named "Acqmeter" - but a UNIX user can only terminate his/her own processes, hence this does not affect other users that might be logged into the same system, and who might be running their own instances of "Acqmeter";
- we run the "pkill" command in background (such that exiting VNMR is not delayed);
- we discard any error output (such as from "pkill" reporting that it is unable to kill another user's "Acqmeter" process)
- the ":$dummy" captures any shell feedback (such as a newline, causing VNMR

to switch to the text display, see Agilent MR News 1997-05-15).
In Solaris 8, "pkill" is not available, and therefore the solution (i.e., the construct to add to "/vnmr/maclib/exit") is a bit more complex:

```
        $me='' $pid=''
        shell('who am i | cut -d" " -f1; cat'):$me
        shell('ps -ef | grep -w',$me,
              '| grep -w Acqmeter | tail -1 | awk \'{print $2}\'; cat'):$pid
        if $pid<>'' then
          shell('kill -9',$pid,' 2>/dev/null &'):$dummy
        endif
```

Here, we first select the lines containing the local user name (i.e., the user's own processes only), then we look for processes named "Acqmeter" and extract the process-ID which we then use in a "kill" call on the last instance (if any) of "Acqmeter". This could be expanded to capture multiple (say, up to three) instances of the "Acqmeter" process:

```
        $me='' $pid1='' $pid2='' $pid3='' $dummy=''
        shell('who am i | cut -d" " -f1; cat'):$me
        shell('ps -ef | grep -w',$me,
              '| grep -w Acqmeter | awk \'{print $2}\';cat'):$pid1,$pid2,$pid3
        if $pid1<>'' then
          shell('kill -9',$pid1,' 2>/dev/null &'):$dummy
        endif
        if $pid2<>'' then
          shell('kill -9',$pid2,' 2>/dev/null &'):$dummy
        endif
        if $pid3<>'' then
          shell('kill -9',$pid3,' 2>/dev/null &'):$dummy
        endif
```

Here, the second "shell" call returns any second or third process-ID on separate lines, thus permitting capturing each process-ID in a separate recipient variable in MAGICAL, see also Agilent MR News 1999-08-14.

[Agilent MR News 2005-12-04]

2006-03-06:

SWITCHING BETWEEN SOLARIS AND LINUX - "su", "ps":

In Agilent MR News 2005-03-20 and Agilent MR News 2005-03-28 we posted articles that discussed issues in connection with porting shell scripts from Solaris to Linux platforms, or making shell scripts compatible with a variety of platforms. The transition between Solaris and Linux not only affects shell scripts, but also many commands that users may call frequently. In this and in future notes we will try to highlight the most prominent changes between these two operating environments:

- the output format of any given command may differ;
- command options and argument format may be entirely different! If in doubt, please consult the UNIX / Linux manual, e.g.: "man ps"
- some commands may only be available in one of the operating environments; one such example is the Solaris "nawk" command which in Linux is covered by the standard "awk" command, see Agilent MR News 2005-03-28.

If you operate both platforms, you will almost inevitably run into error messages because you happened to type the command or argument format of the "other" platform. Here are two examples of differences between Solaris and Linux:

- Solaris follows the SVR4 standard: the command "ps" is typically used with the "-ef" option. "ps -ef" produces the following output:

```
         UID   PID  PPID  C    STIME TTY      TIME CMD
        root     0     0  0   Feb 27 ?        0:03 sched
        root     1     0  0   Feb 27 ?        0:16 /etc/init -
        root     2     0  0   Feb 27 ?        0:00 pageout
        root     3     0  1   Feb 27 ?       11:41 fsflush
        ...
```

In RedHat Linux, the output of "ps -ef" is slightly different:

```
        UID        PID  PPID  C STIME TTY          TIME CMD
        root         1     0  0  2005 ?        00:00:01 init [5]
        root         2     1  0  2005 ?        00:00:01 [migration/0]
        root         3     1  0  2005 ?        00:00:00 [ksoftirqd/0]
        root         4     1  0  2005 ?        00:00:02 [migration/1]
        ...
```

However, Linux users will mostly use the "BSD options" of the "ps" command;

a typical call is "ps -aux", producing the following output:

```
     USER  PID %CPU %MEM   VSZ  RSS TTY    STAT START   TIME COMMAND
     root    1  0.0  0.0  4740  520 ?      S     2005   0:01 init [5]
     root    2  0.0  0.0     0    0 ?      S     2005   0:01 [migration/0]
     root    3  0.0  0.0     0    0 ?      SN    2005   0:00 [ksoftirqd/0]
     root    4  0.0  0.0     0    0 ?      S     2005   0:02 [migration/1]
       ...
```

- In Solaris, the command "su" or "su user_name" changes to the new user identity, but preserves the environment of the calling user. For a "more complete switch", one needs to call "su -" or "su - user_name", see also Agilent MR News 2005-07-23 for a related article. In RedHat Linux, "su" behaves quite differently:
  - "su" or "su user_name" does NOT preserve the original environment;
  - to preserve the original set of environment variables one needs to use the "-p" option, i.e., use "su -p" or "su -p user_name".

[Agilent MR News 2006-03-06]

2006-03-21:

CAVEAT FOR WRITING SHELL SCRIPT FOR MacOS X:

Traditionally in the UNIX world (and typically also in Linux) user names are short (usually just lower case) alphanumeric strings, starting with a non-numeric character. In MacOS X, however, it is not uncommon to see user names (and host names) containing blank characters, i.e., consisting of multiple words (such as "firstname lastname"). Normally, MacOS X users are not aware of any complications from using such names - it's all handled gracefully by the operating system. However, when you are writing shell scripts this suddenly DOES matter! A construct such as

```
        user=.......
        if [ $user = vnmr1 ]; then
          ...
```

will FAIL if "$user" translates to several words! The proper solution (and a protection against any adverse effects from this) is ALWAYS to use double quotes around shell variables, i.e., change the above "if" statement to

```
        if [ "$user" = vnmr1 ]; then
```

See also Agilent MR News 2005-10-18. In the case of the user library, this very problem may have the unfortunate side-effect that upon installing a contribution in "/vnmr", the directory "/vnmr/bin" may be erased - we will create and post a bug report on this in the next issue and fix any install script (and other shell script) that may cause such problems. Conclusion: if you want to install user library contributions under MacOS X, make sure you do this from accounts with single-word name ONLY, until the librarian has re-checked the contributions which currently claim to be MacOS X compatible.

[Agilent MR News 2006-03-21]

2006-04-03:

A SUBTLE TRAP IN WRITING UNIVERSAL SHELL SCRIPTS:

In Agilent MR News 2006-03-21 we posted a note pointing to potential problems that might occur in shell scripts when handling file names containing blank spaces, as encountered fairly commonly in MacOS X (the same applies to MS Windows). The proposed / recommended solution was to use double quotes around shell variable calls, especially in tests for "if" and "while" constructs and the like:

```
        if [ "$user" = vnmr1 ]; then
```

For "total" protection you might want to consider using double quotes around ALL shell variable calls - HOWEVER, there is one instance where this may cause problems! Consider the following construct:

```
        if [ -f $filename.* ]; then
```

i.e., we test whether there is a file matching the contents of "$filename", with any extension. In the editor's case, in the user library "extract" script he was checking for the presence of a contribution with

```
        if [ -f $dir/$filename.* ]; then
```

to check whether the README file ("*.README") AND/OR the contribution itself ("*.tar.Z", "*.tgz", etc.) is present. These constructs work OK - the "-f" option to the "test" function (implemented here through the square brackets ("[" .. "]") DOES work even if the wildcard match yields multiple matches. Now, if we want to implement "filename blank space protection" by changing this construct to

```
        if [ -f "$dir/$filename.*" ]; then
```

that construct suddenly fails - what happened? The problem is NOT the wildcard match itself (as long as we use double quotes rather than single quotes, the shell WILL do the wildcard matching) - however, in that particular case there were indeed MULTIPLE MATCHES - which caused this construct to translate into (for example)

```
        if [ -f "bin/showconfig.README bin/showconfig.tar.Z" ]; then
```

i.e., now the "test" function was looking for a SINGLE file with a name consisting of the TWO names: "bin/showconfig.README bin/showconfig.tar.Z", i.e., the double quotes now combine multiple tokens into a single one with embedded blanks - and there is of course no file with that name, the test yields "false"! The solution: ONLY in the case of the following (and related) "test" operators, a wildcard argument AFTER the operator also works if the matching names contain blanks, AND/OR if it matches multiple names:

-x	true if there are matching executables
-f	true if there are matching plain files
-s	true if there are matching plain files with non-zero size
-d	true if there are matching directories
-h	true if there are matching symbolic links

In other words: IN THIS SPECIFIC CASE the double quotes should NOT be used.

Note that the "=" and "!=" comparison operators (as in the first construct above) follows DIFFERENT rules, see Agilent MR News 2006-03-21.

[Agilent MR News 2006-04-03]

2006-07-05:

A SUBTLE TRAP IN WRITING UNIVERSAL SHELL SCRIPTS - FOLLOW-UP:

In Agilent MR News 2006-04-03 we discussed the use of double quotes around variable names to protect (Bourne / "sh") shell scripts against unexpected blanks in file names, in order to make such scripts usable in environments such as MacOS X (where blanks in file names are a common and frequently used feature).

The same article stated that using double quotes around wildcard (file name) expressions must be AVOIDED, as it would cause the script to interpret multiple matches as a single word / path name with embedded blanks. It was further stated that in (Bourne) shell test functions such as

```
        if [ -f $dir/$filename.* ]; then
```

multiple matches would still yield the expected result - HOWEVER, this is valid in Solaris, but NOT in RHEL 4 (where "bash" is used in lieu of "sh"). We therefore STRONGLY recommend AVOIDING the above type of test unless a FIXED filename is specified - i.e., do NOT use "-f", "-d", "-x", "-s", or "-h" test functions in connection with wildcard expressions. There are alternatives that DO work as universal replacements:

- for a simple test for the presence of directory entries with a matching name (regardless of the file type) you may use a construct such as
  ```
        if [ `ls -d $dir/$filename.* 2>/dev/null | wc -l` -gt 0 ]; then
  ```
  the "-d" option stops "ls" from listing the contents of subdirectories rather than just the name of the subdirectory only. If there is no file or subdirectory with matching name, "ls" will report an error - this error is suppressed / discarded via the "2>/dev/null" option.
- If you also want to check for specific file TYPES you can often use "find" in lieu of "ls": for plain files use
  ```
        if [ `find $dir/$filename.* -prune -a -type f 2>/dev/null | wc -l` ...
  ```
  to check for directories ONLY use "-type d", for symbolic links ("-h" with the standard "test" function) use "-type l". Note the "-prune" option: this prevents "find" from descending into subdirectories. The following will NOT work as expected:
  ```
        if [ `find $dir -prune -a -name 'pattern.*' -a -type d ...
  ```
  because here, the "-prune" will cause "find" to stop searching at the level of the specified directory itself; however, you COULD use
  ```
        if [ `find $dir/* -prune -a -name 'pattern.*' -a -type d ...
  ```
  If the matching expression contains a variable name, that variable name can NOT be enclosed in single quotes - yet, the wildcard MUST be protected against interpretation by the shell:
  ```
        if [ `find $dir/* -prune -a -name $filename'.*' -a -type d ....
  ```

[Agilent MR News 2006-07-05]

2007-02-20:

USING THE LINUX / UNIX "find" WITHIN MAGICAL:

A user wanted to use the UNIX "find" utility from within a VnmrJ macro. The UNIX command scheme might involve a call such as

```
        find $vnmruser -name '*.fid' -a -mtime +8
```

(find files or directories named "*.fid" that have last been modified more than 8 days ago). Note that in a UNIX shell we need to "protect" the wildcard argument "*.fid" from an interpretation by the shell, hence the single quotes: WITHOUT quotes that argument would be expanded into whatever file names match the wildcard expression in the current directory (and if there is no match you would get a syntax error from the "find" utility); with DOUBLE quotes (at least in some shells) the wildcard would be expanded into all matching file names and passed to "find" as a single argument consisting of (possibly) multiple names.

To make this call inside a VnmrJ macro, using a "shell" call, you could use one of the following options:

```
        shell(`find`,userdir,`-name '*.fid' -a -mtime +8`)
```

(note that the "back quotes" must be the OUTER set, as within the UNIX shell they would have an entirely different meaning!) or alternatively, using backslashes for "escaping" / hiding the quotes inside the "shell" call from the MAGICAL interpreter:

```
        shell('find',userdir,'-name \'*.fid\' -a -mtime +8')
```

In these calls, comma-separated arguments translated to space separation in the UNIX call; other alternatives would be

```
        shell(`find `+userdir+` -name '*.fid' -a -mtime +8`)
```

or

```
        shell('find '+userdir+' -name \'*.fid\' -a -mtime +8')
```

Note the extra spaces in these calls!

That part is straightforward; the tricky part comes with feeding the results into local MAGICAL variables: "find" returns all results in one path / file name per line, and MAGICAL transfers this into one return argument per line:

```
        shell('find',userdir,'-name \'*.fid\' -a -mtime +8'):$name1,$name2,...
```

However, you don't know how many names "find" will feed back! One can find the number of results first, e.g.:

```
        nres=0
        shell(`find`,userdir,`-name '*.fid' -a -mtime +8 | wc -l; cat`):$nres
```

and then you could have a branching tree with the appropriate number of return arguments, e.g.:

```
        if $nres=0 then
          write('error','no matching files found')
        elseif $nres=1 then
          shell(`find`,userdir,`-name '*.fid' -a -mtime +8`):$n1
        elseif $nres=2 then
          shell(`find`,userdir,`-name '*.fid' -a -mtime +8`):$n1,$n2
        elseif $nres=3 then
          shell(`find`,userdir,`-name '*.fid' -a -mtime +8`):$n1,$n2,$n3
        elseif ...
```

which obviously is not only very clumsy, but also very restricted, unless you are interested in a limited number of first possible matches only. Also, "find" may take a while to search a directory tree down to the bottom, so duplicating the "find" call just to evaluate the number of results is not efficient at all.

In conclusion, a different approach is needed in which "find" is called only once, and which permits processing an arbitrary number of search results. To do this, we store the result of "find" in a temporary file and then use "nrecords" to find the number of records / results, and then "lookup" inside a loop to process the results one by one:

```
        $tmp='/vnmr/tmp/'+$0+'.tmp'
        shell(`find`,userdir,`-name '*.fid' -a -mtime +8 >`,$tmp):$dummy
        nrecords($tmp):$nres
        lookup('file',$tmp)
        $ix=1 $res=''
        while $ix<=$nres do
          lookup('readline'):$res[$ix]
          $ix=1+$ix
        endwhile
        rm($tmp)
```

In this case we are filling the results into an arrayed local MAGICAL variable: local ("dollar") variables can be arrayed just the same way as many acquisition parameters (i.e., you must fill the array from the bottom up) - with the key difference that arrayed local variables have no effect on the "array" and "arraydim" parameters. See Agilent MR News 2007-02-02 and documents referred to therein for more information on parameter arrays in general.

Thanks to Richard Lewis, AstraZeneca, Loughborough, U.K., for inquiries and suggestions leading to that article!

[Agilent MR News 2007-02-20]

2008-03-27:

HOW DOES THE UNIX AND LINUX EVOLUTION AFFECT THE USER?

UNIX (and with it Linux), unfortunately, is not - and has never been - a single and uniformly defined operating system, but has evolved over history. Worse than that, UNIX was initially developed at AT&T and at some point commercialized, while a group at Berkeley University essentially re-coded the OS (in order to avoid legal issues) and thereafter pursued their own path in the evolution of the OS. This not only implies that the definition of the UNIX command interface has changed over history, but unfortunately, this also involves considerable divergence between the two main branches of development, the ex-AT&T System V UNIX (e.g., SVR4, on which the Sun/Solaris OS is based), and the Berkeley version of the OS (e.g., BSD 4.x), many components of which later got incorporated into the GNU OS (a project of the Free Software Foundation), and from there into Linux. There were various efforts to unify the UNIX command language - but after several decades there still isn't a single, well-defined standard. One such proposed standard is POSIX, another one was proposed by the X/Open Company which now owns the UNIX trademark. Version 4 of that X/Open standard is called XPG4 (X/Open Portability Guide, version 4), published in 1992. The groups working on Linux made considerable efforts to comply with the X/Open standard - but the latter cannot be the sole (and permanent) basis of an ever expanding, open source project such as Linux.

Sun (back in 1982) started off with SunOS, an operating environment that was based on BSD 4.2 (one of the founders of Sun Microsystems, Bill Joy, also has been one of the key exponents in the BSD development team). In 1988, Sun announced that they would switch to a SVR4-based OS which they called Solaris, and they also made a strong statement that they would adhere to the SVR4 standards. At the same time, Sun is a full member of the X/Open group, and they certainly made efforts to comply with the X/Open standards. But as the XPG4 standard still hasn't been globally adopted and diverged from their existing Solaris / SVR4 OS definition, they were facing a tough decision: should they follow XPG4 and thereby risk potential upsetting of its customer base, or rather stick with their existing definition, ensuring that their user's programs and shell scripts would continue to run without hiccups? As they had at that point already made considerable investments to move from the scientific into the commercial sector, they appear to have decided for the second option:

- the standard commands follow their "traditional" Solaris / SVR4 definition;
- XPG4 compliant versions of the relevant commands were made available and are placed in a separate command directory, "/usr/xpg4/bin", but are NOT part of the default command path.
- if users / developers prefer to work in an XPG4-compliant environment, they could simply add "/usr/xpg4/bin" to the command path and make sure it precedes "/usr/bin" (or "/bin", which is the same in Solaris).

This still is the case in Solaris 10: "/usr/xpg4/bin" is NOT part of the default Solaris command path.

Things are different in Linux (see also Agilent MR News 2008-02-05 and articles referred to therein), in that there is no rigid, printed definition how things must work, but the Linux OS evolves more on the basis of mutual understanding and agreement among the Open Source developers. Still, Linux has remained fairly close to the definitions offered by the SVR4 and XPG4 standards, while expanding on the command set and offering a vast amount of additional, Linux-specific utilities: typically, shell scripts written under Solaris or other SVR4 or XPG4 compliant UNIX versions will run under Linux, requiring very few / minor (if any) adjustments. However, the converse should NOT be assumed, especially of course if Linux-specific commands are used).

On top of all that, there are subtle differences between the various Linux distributions, and within each flavor, there is evolution, possibly not just involving expansion, but also subtle syntax changes - see the note below for an example.

[Agilent MR News 2008-03-27]

REDHAT LINUX AND THE "head" AND "tail" COMMANDS:

The article above may appear fairly esoteric and theoretical - but when it comes to concrete changes / evolutions in command definitions, such changes can have very real (and potentially severe) consequences to the user (hence Sun's hesitations to alter their command definitions!). One such "incident" is currently happening with the UNIX / Linux "head" and "tail" commands:

- in the original SVR4 definition, the number of text lines (N) that are shown by these commands can be specified as "-N", e.g.: "head -3" and "tail -7". The "tail" command can also be used with a "+" character in lieu of the "-", e.g.: "tail +10" prints all lines starting with line 10 (as opposed to counting lines from the end of the file). In the case of "tail" this is the ONLY syntax supported by the Solaris / SVR4 command definition.
- the X/Open definition of these commands specifies the use of a "-n" command option, followed by a numeric argument. The following calls are the XPG4 equivalent to the SVR4-compliant calls above:
  ```
  head -n 3
  tail -n 7
  tail -n +10
  ```
  This syntax is supported by "/usr/xpg4/bin/tail" and "/usr/xpg4/bin/head" and by the Linux definition of these commands; the SVR4 / Solaris "head" command ("/usr/bin/head") ALSO supports the XPG4 syntax, but the SVR4 "tail" command does NOT.
- In the Solaris / XPG4 "head" command, the "-n" option only supports positive (unsigned) integers, whereas in Linux
  ```
  head -n -N
  ```
  means to print ALL BUT THE LAST N LINES - this functionality is absent in Solaris (SVR4 or XPG4).

So far, the original SVR4 syntax was also supported by the XPG4 and Linux versions of these commands (also the MacOS X commands support both syntax versions), therefore so far there was no stringent need to switch to the new syntax - HOWEVER, we just found that with RHEL 5.1, the SVR4 "-N" argument syntax is NO LONGER VALID - one MUST use the "-n" option syntax to specify the number of lines, see "man tail". In RHEL 5.1, the SVR4 syntax causes the "-N" argument to be interpreted as an additional file name, with two adverse consequences:

- there is an error message, indicating that the file "-N" was not found:
  ```
  tail: cannot open `-N' for reading: No such file or directory
  ```
  If "tail" is used with an input pipe, this would cause an additional error about ambiguous input specification.
- if a proper file name ("file_name") was specified in connection with "-N", this will be interpreted as multiple files, causing an extra line
  ```
  ==> file_name <==
  ```
  to be inserted at the top of the output.

This change will affect VnmrJ users in various ways:

- If you have macros and/or shell scripts using "head" and/or "tail" calls, you MUST switch to the new syntax. We will probably offer a facility that will make the old syntax still usable within macros (but not in shell scripts that are called from outside of VnmrJ) - but this support will NOT last forever, so better switch to the new syntax NOW!
- If you anticipate having to support RHEL 5.1 systems as well as systems running RHEL 4.0.x or older, you must still switch to the new syntax which is also supported by the older RHEL versions.
- if you don't plan on switching to RHEL 5.1 for the time being, it still is a good idea to use the new syntax, such that your software is future-proof.
- Badly, if you want your macros and shell scripts to be compatible with both Solaris AND all RHEL versions, you can NOT simply switch to the new syntax, but in Solaris you either need to stay with the old syntax, or you need to switch to the executables in "/usr/xpg4/bin" - i.e., you will need to add some "software switches" into such utilities.
- we will of course make new versions of VnmrJ compatible with the RHEL 5.1 "head" / "tail" syntax - but for current releases this not only causes VnmrJ-internal macros and shell scripts to fail where they use "head" or "tail" (this could be fixed by a VnmrJ patch) - it is very likely to cause the VnmrJ INSTALLATION to fail (as a matter of fact, several users have failed installing VnmrJ 2.1 or VnmrJ 2.2 under RHEL 5.1!) - and this cannot easily be fixed with a VnmrJ patch (unless we were to offer a Linux "pre-install patch" for VnmrJ). At this point we have not decided what we

are going to do to make existing VnmrJ versions compatible with RHEL 5.1, and for which VnmrJ version(s) we will possibly offer such an installation fix. CONCLUSION: DON'T TRY INSTALLING existing VnmrJ releases on RHEL 5.x!

In the article below, the editor proposes possible solutions for writing macros and shell scripts with "head" and "tail" command calls that work under Solaris as well as all RHEL versions.

[Agilent MR News 2008-03-27]

WRITING UNIVERSAL SHELL SCRIPTS AND MACROS - "head" AND "tail":

As outlined above, you will need to change all macros and shell scripts that use calls to the UNIX / Linux "head" and "tail" commands if you want that software to work under RHEL 5.1. Within RedHat Linux you can simply switch to the new command syntax, but if you want your macros and shell scripts to stay back-compatible with Sun / Solaris systems, the necessary amendments are more complex. Here is a recommendation for shell scripts:

- to find all shell scripts that use "head" or "tail" commands, open a shell terminal and call

```
     cd ~/bin
     egrep 'head|tail' *
```

  If your "~/bin" also contains binary / compiled executables, you better use

```
     cd ~/bin
     egrep 'head|tail' script1 script2 ....
```

  i.e., list the shell scripts explicitly and avoid running "grep" or "egrep" on compiled executables, as this could produce binary output, possibly causing havoc in the current shell and forcing you to close / kill the terminal window.
- if you have shell scripts with "head" calls, simply switch these to the new syntax, e.g.: search for "head -" and change all calls

```
     head -1 my_file
```

  into

```
     head -n 1 my_file
```

  this also works under Solaris.
- For scripts using "tail", the fix is more complicated. Here's the editor's proposal (in Bourne or "bash" shell syntax). Near the top of the script use

```
     #!/bin/sh
     ...
     tail=tail
     if [ -x /usr/xpg4/bin/tail ]; then
       tail=/usr/xpg4/bin/tail
     fi
```

  If your shell script has a "Solaris / Linux compatibility switch" already, you can of course also use that, e.g.,

```
     tail=tail
     os=`uname -s`
     if [ `echo $os | grep -ci sunos` -ne 0 ]; then
       tail=/usr/xpg4/bin/tail
       ...
     fi
```

  Thereafter, change all lines calling "tail" from, e.g.,

```
     tail -3 my_file
```

  into

```
     $tail -n 3 my_file
```

  and all lines with

```
     tail +3 my_file
```

  into

```
     $tail -n +3 my_file
```

  i.e., switch to the new syntax and as command name use a local variable that either translates to "tail" (Linux) or "/usr/xpg4/bin/tail" (Solaris only). Note that "/usr/xpg4/bin/tail" is present in Solaris 10 and in all predecessor versions (we checked back to Solaris 2.4).

Equivalent constructs need to be added to macros using "head" or "tail" in MAGICAL "shell" calls:

- to find all macros that use "head" or "tail" commands, open a UNIX / Linux shell terminal and call

```
     cd ~/vnmrsys/maclib
     egrep 'head|tail' *
```

- change all "head" calls to the new syntax, e.g.: change

```
     shell('head -'+$line, $file):$out
```

  into

```
     shell('head -n', $line, $file):$out
```

(remember that commas between arguments in "shell" calls will be translated to blank spaces, whereas string concatenation using "+" suppresses such extra blank characters).

- as in shell scripts, you will need to insert a switch for "tail" near the top of such macros, e.g.:

```
     $tail='tail'
     exists('/usr/xpg4/bin/tail','file','x'):$e
     if $e then
       $tail='/usr/xpg4/bin/tail'
     endif
```

and below that, change all "tail" calls to the new syntax, e.g.:

```
     shell('tail -'+$line, $file):$out
```

into

```
     shell($tail,'-n', $line, $file):$out
```

and in case of calls with "+N" argument change

```
     shell('tail +'+$line, $file):$out
```

into

```
     shell($tail,'-n +'+$line, $file):$out
```

Again, note the subtle difference between concatenated strings and separate string arguments!

The editor (for for BioPack and his contributions in the "bin" and "maclib" subdirectories) and Krish Krishnamurthy (for Chempack 4.1) have started making the necessary adjustments in the Agilent MR User Library - this will take a while to be complete and tested. You will see indications of the associated updates if you look into "additions" - but at the moment (i.e., prior to the availability of a VnmrJ release that is compatible with RHEL 5.1) this alone is neither relevant nor (by itself) a reason to download and reinstall a contribution.

[Agilent MR News 2008-03-27]

==