160 VM

December 1999

In this issue

3 Several ways to print files
12 An extended TIME function
19 A Script-to-HTML translator
28 Working with long REXX strings
39 EXCEL in REXX
46 A full screen console interface – part 17
53 VM news

update

VM Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38030
From USA: 01144 1635 38030
E-mail: trevore@xephon.com

North American office

Xephon/QNA
1301 West Highway 407, Suite 201-405
Lewisville, TX 75077-2150
USA
Telephone: 940 455 7050

Editorial panel

Articles published in *VM Update* are reviewed by our panel of experts. Members of the panel include Reinhard Meyer (Germany), Philippe Taymans (Belgium), Romney White (USA), Martin Wicks (UK), and Jim Vincent (USA).

Subscriptions and back-issues

A year's subscription to *VM Update*, comprising twelve monthly issues, costs £180.00 in the UK; $275.00 in the USA and Canada; £186.00 in Europe; £192.00 in Australasia and Japan; and £190.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the January 1990 issue, are available separately to subscribers for £16.00 ($23.00) each including postage.

Editor

Trevor Eddolls

Disclaimer

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

VM Update on-line

Code from *VM Update* can be downloaded from our Web site at http://www.xephon.com/vmupdate.html; you will need the user-id shown on your address label.

Contributions

Articles published in *VM Update* are paid for at the rate of £170 ($250) per 1000 words for original material. To find out more about contributing an article, without any obligation, please contact us at any of the addresses above and we will send you a copy of our *Notes for Contributors*.

Several ways to print files

The utilities described here print files from VM in several different ways. With these REXX procedures you can print CMS files and VM spool files to RSCS connected printers, SNA printers, via the VSE POWER list queue, or to TCP/IP printers.

PRSNA EXEC

PRSNA prints a CMS file or a VM spool RDR or PRT entry to a printer via RSCS. The printer can be any RSCS printer; it does not have to be an SNA printer. The syntax for the call is:

```
PRSNA  printer fn ft <fm>
PRSNA  printer PRT spoolnum
PRSNA  printer RDR spoolnum
```

where:

- ‘printer’ is the printer name (eg P010109 or PRT4234).
- ‘fn’ is the filename.
- ‘ft’ is the filetype.
- ‘fm’ is the filemode (default A).
- ‘spoolnum’ is the spool-id.

Hardcoded values

The RSCS service machine is ‘RSCS’ – you should change this accordingly. The printer names must be defined in RSCS as a directly-connected printer or a VTAM-connected SNA printer (LU).

```
/*********************************************************************/
/* Print a file or spool entry to an RSCS printer                    */
/*********************************************************************/
/* Call:   PRSNA  printer fn ft <fm>                                 */
/*         PRSNA  printer PRT spoolnum                               */
/*         PRSNA  printer RDR spoolnum                               */
/*                printer = printer name (eg P010109 or PRT4234)     */
/*                fn  = filename                                     */
/*                ft  = filetype                                     */
/*                fm  = filemode   (default A)                       */
```

```
/*                spoolnum = spool-id                                  */
/*********************************************************************/
trace off
parse upper arg printer fn ft fm .
if printer = '?' then signal errparm
address xedit 'EXTRACT /FN /FT /FM /ALT'
if rc = Ø & ¬(FTYPE.1 = 'FILELIST' & right(FMODE.1,1) = Ø) then do
   if fn = '' then fn = FNAME.1
   if ft = '' then ft = FTYPE.1
   if fm = '' then fm = FMODE.1
   if alt.1 > Ø then address xedit 'SAVE'
   end
else do
   if printer = '' then signal errparm
   if fm = '' then fm = 'A'
end
'CP SET IMSG OFF'
/*********************************************************************/
/* PRT or RDR spool file: tag file, transfer to RSCS                 */
/*********************************************************************/
if fn = 'RDR' | fn = 'PRT' then do
   'CP TAG FILE' ft printer
   'CP TRANSFER' fn ft 'TO RSCS RDR'
   signal ende
end
/*********************************************************************/
/* CMD file: tag spool device and PRINT to RSCS RDR                  */
/*********************************************************************/
'CP SPOOL PRT TO RSCS'
'CP TAG DEV PRT' printer
'PRINT' fn ft fm
'CP TAG DEV PRT'
'CP SPOOL PRT FOR *'
'CP SET IMSG ON'
ende:
'CP SET IMSG ON'
exit
/*********************************************************************/
/* Help                                                              */
/*********************************************************************/
errparm:
'VMFCLEAR'
address cms 'type prsna exec * 1 12'
```

PRVSE EXEC

PRVSE prints a CMS file to the POWER LST queue. The syntax for the call is:

```
PRVSE  <vse> fn ft <fm> <(CL class> <CO copies>
```

where:

- ‘vse’ is the VSE machine to which to submit the print job (default comes from global variable $VSEDEF).
- ‘fn’ is the filename. If fn=SEL, then all files will be printed that were previously selected by EXEC SEL1/SEL; ft is then ignored.
- ‘ft’ is the filetype.
- ‘fm’ is the filemode (default A).
- ‘class’ is the class in the LST queue (default D).
- ‘copies’ is the number of copies (default 1).
- ‘fn’, ‘ft’, ‘fm’ can also be generic (eg TEST*).

The output is shifted one column to the right, because the file is submitted as data cards to the VSE job. This is done because the file could contain VSE and/or POWER JECL cards that would be (mis)interpreted by VSE or POWER.

A file PRVSE JOB is needed, containing a skeleton for the submitted print job. The printing in VSE is done by program DTSRELST, an ICCF utility program.

The output of DTSRELST is separated into the job control output and the actual printed file by a utility, CAPRUTL0, which is included in Computer Associates’ RAPS product – this can also print LST queue entries to a CICS printer. If you don’t have CAPRUTL0, you will have to separate the printed file and the job control pages manually.

The prerequisite procedures EXXX and INCLUDE XEDIT were published in *Saving all relevant VM/VSE data – part 2,VSE Update*, Issue 26, June 1997. SEL and SEL1, which are not mentioned here, allow you to preselect files; they are then all printed with one call to PRVSE.

```
/*********************************************************************/
/* Print CMS file via VSE to POWER LST queue                         */
/*********************************************************************/
/* Call:   PRVSE  <vse> fn ft <fm> <(CL class> <CO copies>           */
/*                vse = VSE machine to which to submit print job     */
/*                      (default comes from global var $VSEDEF)      */
/*                                                                   */
/*                fn  = filename                                     */
```

```
/*                      if fn=SEL, then all files will be printed,    */
/*                      that were previously selected by              */
/*                      EXEC SEL1/SEL; ft is then ignored             */
/*               ft  = filetype                                       */
/*               fm  = filemode   (default A)                         */
/*               class = class in the LST queue (default D)           */
/*               copies= number of copies    (default 1)              */
/*               fn, ft, fm can also be generic (eg TEST*)            */
/************************************************************************/
/* PS: The output if shifted one column to the right                    */
/************************************************************************/
trace off
'GLOBALV SELECT $$GLOB$$ GET $vse1    $vse2    $vse3    $vsedef',
                           '$sysid1  $sysid2  $sysid3         '
parse upper arg vse fn ft fm . '(' opts
if vse = '?' then signal errparm
if vse = substr($vse1,4,3) then vse = $vse1
if vse = substr($vse2,4,3) then vse = $vse2
if vse = substr($vse3,4,3) then vse = $vse3
address xedit 'EXTRACT /FN /FT /FM /ALT'
if rc = Ø & ¬(FTYPE.1 = 'FILELIST' & right(FMODE.1,1) = Ø) then do
   if vse = '' then vse = $vsedef
   if vse ¬= $vse1 & vse ¬= $vse2 & vse ¬= $vse3 then do
      vse = $vsedef
      parse upper arg fn ft fm . '(' opts
   end
   if fn = '' then fn = FNAME.1
   if ft = '' & fn ¬= 'SEL' then ft = FTYPE.1
   if fm = '' & fn ¬= 'SEL' then fm = FMODE.1
   if alt.1 > Ø then address xedit 'SAVE'
   end
else do
   if vse = '' then signal errparm
   if vse ¬= $vse1 & vse ¬= $vse2 & vse ¬= $vse3 then do
      vse = $vsedef
      parse upper arg fn ft fm . '(' opts
   end
end
if fn = 'SEL' & ft = '' then do
   fn = 'S$E$L$'
   ft = 'S$E$L$'
   fm = 'A'
end
if vse = '' | left(vse,2) = '(C' | fn = '' | left(fn,2) = '(C' | ,
   ft = '' | left(ft,2) = '(C' ,
   then signal errparm
if fm = '' | left(fm,2) = '(C' then fm = 'A'
parse upper arg . '(' opts
parse upper var opts 'CL' class .
if class = '' then class = 'D'
```

```
parse upper var opts 'CO' copies .
if copies = '' then copies = '1'

if vse = $vse1 then sysid = $sysid1
   else if vse = $vse2 then sysid = $sysid2
        else sysid = $sysid3

/* A file PRVSE JOB must exist that contains the VSE job skeleton */

'EXECIO 1 CP (STR Q SET'
pull . msg .
'EXECIO Ø CP (STR SET MSG OFF'
'VSECMD' vse 'L LST,CFNO=PRVS,CCLASS=X'
'EXEC EXXX' vse fn ft fm 'PRVSE CLASS' class '$SHIFT$ DUMMY' ,
            'JOBNAME' fn 'COPIES' copies 'SYSID' sysid '(5'
/* $SHIFT$ tells EXXX that parameter '(SHIFT' is appended to INCLUDE  */
/* Therefore INCLUDE XEDIT shifts the contents one column to the right*/

'EXECIO Ø CP (STR SET MSG' msg
exit
errparm:
'VMFCLEAR'
address cms 'type prvse exec * 1 19'
```

PRVSE JOB

```
* $$ LST CLASS=X,PRI=7,FNO=PRVS,DISP=L,SYSID=*SYSID*
// JOB PRVSE
// OPTION NOLOG
// UPSI 1Ø
// EXEC DTSRELST
/INCLUDE XXXXXXXX XXXXXXXX X
/*
* $$ LST CLASS=X,PRI=3,FNO=PRVS,DISP=L
// ASSGN SYSØØ1,DISK,VOL=DOSRES,SHR
// ASSGN SYSØØ2,DISK,VOL=SYSWK1,SHR
// EXEC CAPRUTLØ,SIZE=8ØK
INPUT  *JOBNAME*,CPRI=7,CFNO=PRVS,ENDISP='PRI=3',FCB=$$BFCB22
REPORT *JOBNAME*,SELECT=(1,ALL,EQ,'// JOB PRVSE '), -
               LST='CLASS=X,DISP=L,FNO=PRVS,SYSID=*SYSID*'
REPORT *JOBNAME*,SELECT=NOMATCH, -
               LST='CLASS=*CLASS*,DISP=L,COPY=*COPIES*,SYSID=*SYSID*'
/*
/&
```

PRTCP EXEC

Let us assume you have a VM system that has TCP/IP installed, then

PRTCP allows you to print a CMS file to a printer anywhere on the TCP/IP network.

The syntax for the call is:

```
PRTCP  printer fn ft <fm>
```

where:

- ‘printer’ is the printer name (from TCPIP LPD CONFIG). Available TCP/IP printers are shown.
- ‘fn’ is the filename.
- ‘ft’ is the filetype.
- ‘fm’ is the filemode (default A).

Prerequisite is a TCP/IP service machine in VM with LPD/LPR installed. You have to specify the destination printers in the LPD CONFIG file of this service.

You need a CMSBATCH machine in the VM system that contains TCP/IP.

Hardcoded values

The following are hardcoded values:

- ‘CMSBATCH’ is the CMS batch machine.
- ‘RSCS’ is the name of the RSCS machine.
- ‘SYS1’ is the name of the VM system that has TCP/IP installed.
- ‘TCPMAINT’ is the name of the TCP/IP maintenance machine.
- ‘592’ is the virtual address of the TCP/IP mini-disk with LPD/LPR.
- ‘VMTCPIP’ is the TCP/IP host name of the VM TCP/IP system (it must be defined in the HOSTS file and point to a TCP/IP address, unless DNS is used).

```
/*********************************************************************/
/* Print file on a printer on the TCP/IP network                     */
/*                                                                   */
```

```
/*********************************************************************/
/* Call:   PRTCP  printer fn ft <fm>                                 */
/*                printer = printer name (from TCPIP LPD CONFIG)     */
/*                          X = available TCP/IP printers are shown  */
/*                fn  = filename                                     */
/*                ft  = filetype                                     */
/*                fm  = filemode   (default A)                       */
/*********************************************************************/
trace off
'GLOBALV SELECT $$GLOB$$ GET $vse1    $vse2    $vse3    $vsedef',
                           '$sysid1  $sysid2  $sysid3         '
parse upper arg printer fn ft fm .
if printer = '?' | printer = '' then signal help
call showprt

if fm = '' then fm = 'A'
fn = strip(fn)

address command 'ESTATE' fn ft fm
if rc ¬= Ø then do
   say 'file to print ('fn ft fm') does not exist; RC='rc
   exit
end

'LISTFILE' fn ft fm '(DATE STACK'
pull . . . recfm lrecl .
/*********************************************************************/
/* Copy print file to A disk (unpack if necessary)                   */
/*********************************************************************/
address cms set cmstype ht
'COPY' fn ft fm 'P$R$T J$O$B A (UNPACK REPLACE OLDD'
if rc = 32  /* file is not packed */
   then 'COPY' fn ft fm 'P$R$T J$O$B A (REPLACE OLDD'
/*********************************************************************/
/* Send job to CMSBATCH of the machine that has TCP/IP (via RSCS)    */
/* The job MOVEs the file to print to a temporary file               */
/* on the A disk of the CMSBATCH machine and then prints this temp   */
/* file by LPR.                                                      */
/*********************************************************************/

'SPOOL PUN RSCS CONT'
'TAG   PUN SYS1 CMSBATCH'
queue '/JOB CMSBATCH PRTCP'
queue 'CP LINK TCPMAINT 592 592 RR ALL'
queue 'ACC 592 Z'
queue 'FILEDEF INMOVE  TERM (RECFM' recfm 'LRECL' lrecl 'BLOCK' lrecl
queue 'FILEDEF OUTMOVE DISK' fn ft 'A (RECFM' recfm 'LRECL' lrecl,
                                    'BLOCK' lrecl
queue 'MOVEFILE'
'EXECIO' queued() 'PUNCH'
```

```
'PUNCH P$R$T J$O$B A (NOH'
queue '/*'
queue 'LPR' fn ft' A (HOST VMTCPIP PRINTER' printer
queue '/*'
'EXECIO' queued() 'PUNCH'
'CP SET IMSG OFF'
'SPOOL PUN NOCONT CLOSE'
'CP TAG DEV PUN'
'SPOOL PUN FOR *'
'CP SET IMSG ON'
exit
/*********************************************************************/
/* Show available TCP/IP printers                                    */
/*********************************************************************/
showprt:
'SET CMSTYPE RT'
nrmax = 5

pr.1 = 'KYOES9'
pr.2 = 'KYOXP85'
pr.3 = 'EPSXP95'
pr.4 = 'TI'
pr.5 = 'PRT4234'

if printer ¬= 'X' then do
   do i = 1 to nrmax
      if printer = pr.i then return
   end
   say 'Please enter a valid printer name'
   exit
end

do until nr >= 1 & nr <= nrmax
   say 'The following TCP/IP printers are available'
   say ' '
   say ' 1 Kyocera F-800  at ES/9000'
   say ' 2 Kyocera F-3300 at PC-Server Dept. 1'
   say ' 3 Epson          at PC-Server Dept. 2'
   say ' 4 TI             at RS-6000   Dept. 3'
   say ' 5 IBM 4234       at ES/9000'
   say ' '
   say 'Please enter a number'
   pull nr
end

printer = pr.nr

'SET CMSTYPE HT'
return
/*********************************************************************/
```

```
/* Help                                                                */
/*********************************************************************/
help:
'VMFCLEAR'
address cms 'type prtcp exec * 1 11'
```

PR EXEC

When you want to see a menu that shows you all available printing methods, you can call PR EXEC. Please select one of the following:

- PRSNA – print file or spool entry to an SNA printer.
- PRINT – print file to VM Spool PRT queue.
- PRVSE – print file via VSE to the POWER LST queue.
- PRTCP – print file to TCP/IP printer.

When you select a number from this menu, you will get the help function of the associated procedure.

```
/*********************************************************************/
/* Printing Menu                                                     */
/*********************************************************************/
/*                                                                   */
/* Please select one of the following:                               */
/*                                                                   */
/*   1    PRSNA     Print file or spool entry to a SNA printer      */
/*   2    PRINT     Print file to VM Spool PRT queue                */
/*   3    PRVSE     Print file via VSE to the POWER LST queue       */
/*   4    PRTCP     Print file to TCP/IP printer                    */
/*                                                                   */
/*********************************************************************/
trace off
'VMFCLEAR'
address cms 'type pr exec * 1 12'
pull nr
select
   when nr = '1' then 'EXEC PRSNA  ?'
   when nr = '2' then 'HELP PRINT (ALL'
   when nr = '3' then 'EXEC PRVSE  ?'
   when nr = '4' then 'EXEC PRTCP  ?'
   otherwise nop
end
exit
```


An extended TIME function

Some time ago I wrote a REXX function for date manipulation, published in *VM Update*, Issue 143, July 1998. After that, I was asked to create the obvious follow-up, a similar function for time manipulation.

The result is TIMEFUNC. Like its predecessor, this EXEC should run with any REXX interpreter. It can be used as a function or as a command. As a function, the result is returned, and as a command it is 'said'. It has a quick on-line help, accessible by passing '?' or 'HELP' as argument.

TIMEFUNC has two parameters, both optional, separated by a comma. The first parameter is some kind of time input, and the second is a one-letter code that designates the desired output format. These codes and their output are shown in Figure 1. Note that B and D are variants of the civil format, one without minutes and the other with seconds added. The first parameter, the time input, can be a single time, specified in any of the formats shown, or a pair of 'times', separated by a '+' or '-' signal.

Code	*Output*	*Example*
N (Normal)	hh:mm:ss	17:35:01
C (Civil)	hh:mmxx	11:12pm
B	hhxx	11pm
D	hh:mm:ssxx	11:12:31pm
H	Any number of hours	74
M	Any number of minutes	6141
S	Any number of seconds	369

Figure 1: Codes and output

If a single time is specified, it will be converted to the requested output format. If two times are given, they will be added or subtracted and the result formatted as requested. The two times can have different formats. If the input time is not specified, the result of the standard REXX time() will be used as input (that is, the current time).

The detection of the input format is automatic, except for formats H, M, and S – because a single number is not self-evident. In these cases, add a letter (h, m, or s) to the number to indicate what it represents. In some cases, the result time falls outside the current day. When this happens, the output appears with a separate number (positive or negative) representing the shifted number of days.

A few examples should help to understand how TIMEFUNC works (see Figure 2). The quotes surrounding the arguments are not necessary, in most cases.

TIMEFUNC

```
/*  REXX  *============================================================*/
/*                                                                      */
/*  TIMEFUNC - An extended REXX time utility. Works as command or       */
/*             as function. The result is "said" or returned.           */
/*  Argument1: time_in          default: current time                    */
/*         or  time1-time2      difference between two times            */
/*  Argument2: output format (optional: default is default_fmt_out)     */
/*                                                                      */
/*======================================================================*/

default_fmt_out = "N"                          /* default output fmt */
signal on error                                /* avoid crash with   */
signal on syntax                               /* invalid input      */
arg aaa
aaa = translate(aaa,"","'")                    /* get rid of quotes  */
aaa = translate(aaa,"",'"')                    /* and separate args  */
parse var aaa timein","fmt_out                 /* by the comma       */
timein = space(timein,0)                       /* if no timein,      */
if timein ="" then timein = time()             /* get current time() */
if timein="?"|timein="HELP" then signal helpe
fmt_out = space(fmt_out,0)
if fmt_out ="" then fmt_out=default_fmt_out
parse source . calltype .                      /* find out how called*/
parse var timein time1"+"time2_more            /* separate times by  */
parse var time1  time1"-"time2_less            /* + or - sign        */
if time1 ="" then time1 = time()               /* get current time() */
time2 = ""
operand = ""
```

Input	*Result*	*Comments*
timefunc	09:28:19	Without arguments, same as time()
timefunc ‘,c’	09:28am	Current time in civil format
timefunc ‘,m’	568	Current time in minutes
timefunc ‘,s’	34099	Current time in seconds
timefunc ‘14:15:32,d’	2:15:32pm	Convert input hour to D format
timefunc ‘00:59:12,c’	12:59am	Convert input hour to C format
timefunc 545h	17:00:00 22	545 hours is 17 hours and 22 days
timefunc ‘27h,m’	1620	27 hours are 1620 minutes
timefunc ‘3602s,h’	1	3602 seconds is one hour (remainders are truncated)
timefunc ‘17:07:05-6:14’	10:53:05	Difference between two times
timefunc ‘11:15am-16:30’	18:45 -1	The resulting hour falls yesterday (-1)
timefunc ‘13:22+96h,c’	1:22pm 4	The resulting hour is 4 days ahead
timefunc ‘13:22+96h,m’	6562	Same thing, but in minutes
timefunc ‘-78m’	08:10:19	Current time (omitted) less 78 minutes

Figure 2: Examples of TIMEFUNC usage

```
if time2_more <>"" then do
   operand = "+"
   time2 = time2_more
end
if time2_less <>"" then do
```

```
   operand = "-"
   time2 = time2_less
end
seconds_1 = calculate_seconds(time1)          /* calculate seconds  */
if seconds_1 < 0 then do                      /* of time1           */
   erro = seconds_1
   signal error
end
if time2 <>"" then do                         /* if time2 specified */
   seconds_2 = calculate_seconds(time2)       /* calculate seconds  */
   if seconds_2 < 0 then do                   /* of time2           */
      erro = seconds_2
      signal error
   end
   interpret abs_secs "=" seconds_1 operand seconds_2
end
else do                                       /* calculate result   */
   abs_secs = seconds_1                       /* in seconds         */
end
parse var abs_secs absolute_secs"."lixo       /* get rid of decimals*/

/*====================================================================*/
/* Translate absolute seconds into days, hours, minutes, and seconds*/
/*====================================================================*/
days = 0
final_secs = absolute_secs
if final_secs < 0 then,
 do until final_secs > 0
    final_secs = final_secs + 86400
    days = days-1
end
if final_secs > 86400 then,
 do until final_secs < 86400
    final_secs = final_secs - 86400
    days = days+1
end
if days = 0 then days=""
hours = final_secs % 3600
final_secs = final_secs - 3600*hours
mins = final_secs % 60
secs = final_secs - 60*mins
hours = right(hours,2,"0")
mins  = right(mins,2,"0")
secs  = right(secs,2,"0")
/*====================================================================*/
/*                 Select output type and format it                   */
/*====================================================================*/

out = 0
select
```

```
 when fmt_out = "B" then do
   out = set_civil_hour(hours)
   output = space(out,Ø)
 end
 when fmt_out = "C" then do
   out = set_civil_hour(hours)
   output = word(out,1)":"mins||word(out,2)
 end
 when fmt_out = "D" then do
   out = set_civil_hour(hours)
   output = word(out,1)":"mins":"secs||word(out,2)
 end
 when fmt_out = "H" then do
   output = absolute_secs%36ØØ
   days = ""
 end
 when fmt_out = "M" then do
   output = absolute_secs%6Ø
   days = ""
 end
 when fmt_out = "N" then do
   output = hours":"mins":"secs
 end
 when fmt_out = "S" then do
   output = absolute_secs
   days = ""
 end
 otherwise do
   erro = -9Ø
   signal error
 end
end
if out < Ø then do
   erro = out
   signal error
end
if calltype = "COMMAND" then say output days
else return output days
exit

/*====================================================================*/
/*                        Subroutines                                 */
/*====================================================================*/
set_civil_hour: procedure
 arg valor .
 valor = strip(valor,"L","Ø")
 select
   when valor<1            then return 12 "am"
   when valor>Ø & valor<12 then return valor "am"
   when valor=12           then return 12 "pm"
```

```
   when valor>12           then return valor-12 "pm"
   otherwise                    return -98
 end
return follow_the_white_rabbit

/*====================================================================*/
/* Procedure to calculate seconds and check input format for errors */
/* Returns a negative value if some inconsistency was found         */
/*====================================================================*/
calculate_seconds:  procedure
arg mytime
civil_format = 0
if pos("AM",mytime) > 0 then do
   civil_format = 1
   mytime = left(mytime,length(mytime)-2)
end
if pos("PM",mytime) > 0 then do
   civil_format = 2
   mytime = left(mytime,length(mytime)-2)
end
if pos("H",mytime) > 0  then hours   = 1
if pos("M",mytime) > 0  then minutes = 1
if pos("S",mytime) > 0  then seconds = 1
parse var mytime hh":"mm":"ss
if (hours=1|minutes=1|seconds=1) then do
   if mm<>""|ss<>"" then return -99
   mytime = left(mytime,length(mytime)-1)
end
if hours   = 1 then return mytime*3600
if minutes = 1 then return mytime*60
if seconds = 1 then return mytime
if datatype(hh,"W") then do
   if hh > 23 then return -98
   if hh < 0  then return -98
   if hh > 12 & civil_format <>0 then return -98
   if hh = 0  & civil_format <>0 then return -98
   if civil_format = 1 & hh = 12 then hh = 0
   if civil_format = 2 & hh < 12 then hh = hh + 12
   final_secs = hh*3600
end
if datatype(mm,"W") then do
   if mm > 59 then return -96
   if mm < 0  then return -96
   final_secs = final_secs + mm*60
end
if datatype(ss,"W") then do
   if ss > 59 then return -95
   if ss < 0  then return -95
   final_secs = final_secs + ss
end
```

```
return final_secs

/*====================================================================*/
error:
syntax:
if calltype = "COMMAND" then,
 select
  when erro = -99 then say "Invalid format"
  when erro = -98 then say "Invalid hour"
  when erro = -96 then say "Invalid minutes"
  when erro = -95 then say "Invalid seconds"
  when erro = -90 then say "Invalid output format"
  otherwise say "Invalid parameters"
 end
exit -1

/*====================================================================*/

helpe:
say "TIMEFUNC 'parm1,parm2'"
say
say "parm1 can be one of the following:"
say "                 time1"
say "                 time1+time2"
say "                 time1-time2"
say
say "   where time1 and time2 are specified in any of the following"
say "   formats (they need not be the same for both):"
say
say "    Format N - hh:mm:ss    ( example: 15:23:09 "
say "    Format B - hhxx        ( example: 3pm        12am "
say "    Format C - hh:mmxx     ( example: 1:23pm     6:08am "
say "    Format D - hh:mm:ssxx  ( example: 7:30:09pm  9:12:04pm"
say "    Format H - hhh         ( example: 56h        8776h"
say "    Format M - mmm         ( example: 545m       95m "
say "    Format S - sss         ( example: 7441s      443s "
say
say "parm2 is the letter corresponding to the desired output format"
say
say " The output format has the form  xxxxx dd  where xxxxx is the"
say " time in the requested format and dd a number (positive or "
say " negative) that represents a number of days in case the difference"
say " between two times falls on a different day. If the difference"
say " falls on the same day, dd does not appear."
```

Luis Paulo Figueiredo Sousa Ribeiro
Systems Engineer
Edinfor (Portugal)

A Script-to-HTML translator

S2H is a VM application that converts Script-VS files into HTML. Its most direct use is to take text containing DCF and GML formatting directives suitable for input to Script-VS and produce from it text for display with an HTML browser, such as Netscape Communicator or Microsoft Internet Explorer. In some instances, such a document may be needed in both output forms; S2H allows document maintenance to be confined to the Script file.

A second application arises in connection with orphaned Script files, namely files that have been archived but for which the Script-VS processor is no longer available to produce a formatted print file. The remedy is to convert the Script file to HTML and print it using the browser's print function.

Thirdly, S2H provides a means for those familiar with Script to produce Web pages without needing any knowledge of HTML.

The program produces complete HTML documents including <html>, <head>, <title>, and <body> tags. It can also be invoked as a REXX filter whose function is to produce line-by-line translation of DCF and GML tags into HTML tags. All commonly used DCF/GML tags are recognized and handled, including headings, lists, bold and italic directives, and cross references (both to headings and to list items). This is discussed in more detail in the program comments.

S2H EXEC

```
/* S2H EXEC */
/*Ø
SCRIPT-to-HTML translator. Reads a SCRIPT file as input and produces
an HTML file as output.
Call:  S2H sfn [sft [sfm]] [,hfn [hft [hfm]]] Defaults are:
  sft:  SCRIPT
  sfm:  *
  hfn:  sfn
  hft:  HTML
  hfm:  A

The output file may sometimes require fine-tuning. The auxiliary output
file sfn CMSUT1 A3 which contains any ignored DCF control words should
also be examined.  Note that it is "read-once".
```

For DCF control words (.xx) and GML tags (:xx) to be processed properly, the SCRIPT file needs to conform to certain conventions:

1. If a line beginning with ".cm HTML" is found, all lines preceding it are ignored. This allows skipping of the DCF control words used for SCRIPT initialization. If there is no such line, the entire file is processed.

This line has a second purpose: any text found following ".cm HTML" will be used for the title of the HTML document. If the line is absent or contains no title text, the text associated with any :title. tag is used. Otherwise, the title will be the name of the SCRIPT file, followed by its creation date.

2. The following DCF control words are recognized and processed as indicated ("¬" means "not," "|" means "or"):

.br	 	.fo off	<PRE>
.bl	 	.fo ¬off	</PRE>
.ce ...	<CENTER>...</CENTER>	.nf off	</PRE>
.ce on	<CENTER>	.nf ¬off	<PRE>
.ce off	</CENTER>	.hr	<HR>
.bf b;...;.pf	<B>...</B>	.hn ...	<Hn>...</Hn> (n = 1-5)
.bf i;...;.pf	<I>...</I>	.sk|sp n	n+1 's; at most 4

The only font names recognized are "b" for bold; "i" for italic. Font changes must not be nested — .bf and .pf must strictly alternate. They need not appear on the same line. (See :hpn below for another method for emphasized text.)

Blank lines are treated as if .bl or .sp 1.

3. The following GML tags are recognized and processed as indicated:

:dl [compact].	<DL [COMPACT]>	:esl.	</UL>
:dt.	<DT>	:lq.	<BLOCKQUOTE>
:dd.	<DD>	:elq.	</BLOCKQUOTE>
:edl.	</DL>	:hp1.	<I>
:gl [compact].	<DL [COMPACT]>	:ehp1.	</I>
:gt.	<DT>	:hp2.	<B>
:gd.	<DD>	:ehp2.	</B>
:egl.	</DL>	:hp3.	<I><B>
:ol [compact].	<OL [COMPACT]>	:ehp3.	</B></I>
:eol.	</OL>	:p|pc.	<P>
:ul [compact].	<UL [COMPACT]>	:xmp.	<PRE>
:eul.	</UL>	:exmp.	</PRE>
:sl [compact].	<UL [COMPACT]>	:lp.	<P>

:hn [id=xxx]. ...	[<A NAME="xxx">]<Hn>...</Hn> (n = 1-5)
:hdref refid=xxx.	<A HREF=#xxx>?X-REF?</A>
:li [id=xxx].	[<A NAME="xxx">]<P><LI>

```
  :liref refid=xxx.    <A HREF=#xxx>?X-REF?</A>
  :dthdr xxx.          <DT><B>xxx</B>
  :ddhdr xxx.          <DD><B>xxx</B>

4.  If an input line begins ".cm#xx...x;yy...y", the string "xx...x" is
output verbatim while the strings ".cm#" and ";yy...y" are discarded.
This permits the input file to contain text xx...x solely for HTML
output and text yy...y solely for SCRIPT processing. Note that either
"xx...x" or ";yy...y" may be absent.

(Note:  this program has been tested using VM/ESA CMS Pipelines 3.Ø1Ø9
and CMS Pipelines Runtime Library Distribution 1.Ø11Ø. The latter
provides features not available in the former and the program takes
account of this at run time. Users encountering difficulties with other
versions of PIPEs should inform the author or, better, download a
current version from pucc.princeton.edu in directory anonymou.376 .)
Ø*/

/* Acknowledgements:
Many thanks to John Hartmann, Rob vd Heij, and Melinda Varian for help
and advice on some rough spots.
*/
/*1 Initial boilerplate:
<HTML>
<HEAD>
<META CONTENT="Auto-translated from $SOURCE$ by S2H.">
<TITLE>
$TITLE$
</TITLE>
</HEAD>
<BODY BGCOLOR="#FFDØDØ" TEXT="#ØØØØØØ" LINK="#ØØ8ØØØ"
      VLINK="#ØØØØDØ" ALINK="#FØØØØØ">
1*/
/*2 Concluding boilerplate:
</BODY>
</HTML>
2*/

PARSE SOURCE . . fn ft fm . how .                       /* This file. */
IF how = "?" THEN SIGNAL rexx                  /* Called as a filter. */
ARG args
IF args = "" | args = "?" THEN DO               /* Display help text. */
  "PIPE <" fn ft fm "| INSIDE '/*Ø' 'Ø*/' | >" fn "HELP A3"
  QUEUE "SUPERSET / FM S1 / PREF OFF / VER 1 79 / SCALE OFF"
  QUEUE "SET CURLINE ON 2"
  QUEUE ":1"
  "XEDIT" fn "HELP A (NOPROF"
  EXIT Ø
END

PARSE VAR args input","output
```

```
PARSE VAR input in it im
IF it = "" THEN it = "SCRIPT"
IF im = "" THEN im = "*"
input = in it im
PARSE VAR output on ot om
IF on = "" | on = "=" THEN on = in
IF ot = "" THEN ot = "HTML"
IF om = "" THEN om = "A"
output = on ot om
auxiliary = in "CMSUT1 A3"  /* Write discarded DCF cw's to this file. */
"PIPE (END | NAME prescan) CMS LISTFILE" input "(NOHEADER DATE",
  "| TAKE 1",
  "| a: FANOUT",
  "| SPEC W1-2 1",
  "| VAR source",                          /* Full name of input file. */
  "| a:",
  "| SPEC W8 1",
  "| VAR fdate0",                               /* Date of creation. */
  "| a:",
  "| SPEC W1-3 1",
  "| GETFILES",                    /* Preview the file to see whether */
  "| b: CASEI FIND .CM HTML"||,  /* input line ".CM HTML" is present. */
  "| TAKE 1",                             /* Abort the read if found. */
  "| SPEC W3-* 1",                /* Save anything else on line for  */
  "| VAR htmltitle",                              /*  document title. */
  "| COUNT LINES",
  "| VAR htmlstart",                               /* True if found. */
  "| b:",
  "| CASEI FIND :title."||,
  "| TAKE 1",
  "| SPEC 8-* 1",
  "| VAR doctitle",
  "| COUNT LINES",
  "| VAR isdoctitle"
IF RC ¬= 0 THEN EXIT RC
SELECT
  WHEN htmlstart THEN DO
    input = input "| FRLAB .cm HTML| DROP 1"
    IF htmltitle = "" & isdoctitle THEN htmltitle = doctitle
  END
  WHEN isdoctitle THEN htmltitle = doctitle
OTHERWISE
  htmltitle = source "("fdate0")"
END
PARSE VAR fdate0 mm "/" dd "/" yy
mm = WORD("January February March April May June July August September",
     "October November December", mm)
yy = 1900+yy; IF yy < 1964 THEN yy + yy + 100 /* No 2K problems here| */
fdate1 = mm dd+0","yy            /* File creation date - long form.  */
"PIPE (END | NAME main) <" fn ft fm,                    /* This file. */
  "| a: INSIDE '/*1' '1*/'",          /* Pick up initial boilerplate. */
```

```
  "| CHANGE '$SOURCE$'"source" ("fdateØ")'",/* (For <META> statement, */
  "| CHANGE" '1F'X"$TITLE$"'1F'X||htmltitle'1F'X,  /*  <TITLE> stmt.) */
  "| b: FANIN Ø 2 1",                        /* All output comes here. */
  "| >" output,
  "| a:",
  "| INSIDE '/*2' '2*/'",               /* Pick up final boilerplate. */
  "| elastic",               /* Buffer as necessary to prevent stall. */
  "| b:",
,
  "| <" input,                                        /* Raw input data. */
  "| c: REXX (" fn ft fm ")",                /* Translate SCRIPT file */
,                          /* using filter below starting at "rexx:". */
  "| b:",                                    /* Write to output file. */
  "| c:",                             /* Rejects from filter come here. */
  "| >" auxiliary          /* Ignored DCF control words to this file. */

EXIT RC

/*====================================================================*/
/* This filter processes all records from the input file, converting
DCF control words and GML tags to their corresponding HTML tags. Those
conversions that require merely a simple substitution are done using
LOOKUP against one or another table defined in comment lines below.
Those requiring more extensive manipulation are handled by one or
another of the pipe fragments immediately below.

Each of these fragments looks for a particular type of DCF control word
or GML tag.  If found, it is processed and the result sent to "out:".
If not, control falls through to the next fragment.  All the fragments
and look-ups are concatenated together in the CALLPIPE at the end. The
order in which they appear there is important — eg, one-line centres
must be disposed of before the look-up for .ce takes place.

DCF control words not recognized by this filter are written to the
secondary output at *.output.1:.  GML tags not recognized by this filter
are written to the primary output at *.output:.                      */

REXX:

"CALLPIPE QUERY VERSION | SPEC W5 1 | VAR version"
PARSE VAR version vers "." relmod
recent = vers = 1 & relmod > Ø1Ø9
          /* If true, recent release of runtime library distribution. */

center =,                                                    /* .ce. */
  "| c1: FIND .CE| SPEC '<CENTER>' 1 W2-* N '</CENTER>' N | out: | c1:"

font =,                                                   /* .bf, .pf. */
  "| f1: CASEI FIND .BF B"||,
  "| f2: FANINANY | XLATE UPPER | f3: FANOUT",
```

```
  "| SPEC '</' 1 W2 N '>' N | f4: JUXTAPOSE | out:",
  "| f3: | SPEC '<' 1 W2 N '>' N | out: | f1:",
  "| f5: CASEI FIND .BF I| f2: | f5:",
,
  "| f6: FIND .PF| CHOP Ø | f4:"
IF recent THEN font = font,     /* Use secondary output of JUXTAPOSE. */
  "| SPEC 'Warning:  unterminated .BF' 1 3 NW | CONSOLE | f6:"
ELSE font = font "| f6:"           /* No secondary output available. */

dcfhdr = ""                                          /* .h1 ... .h5. */
DO i = 1 TO 5
  dcfhdr = dcfhdr,
  "| h"i": FIND .H"i"| SPEC '<H"i">' 1 W2-* N '</H"i">' N",
  "| out: | h"i":"
END

right =,                                                    /* .ri. */
  "| r1: FIND .RI| SPEC '<H4 ALIGN=RIGHT>' 1 W2-* N '</H4>' N",
    "| out: | r1:"
xref =,                                          /* :liref., :hdref. */
  "| xr1: CASEI FIND :liref"||,
  "| xr2: FANINANY | SPLIT | CASEI FIND refid"||,
    "| SPEC '<A HREF=""#' 1 7-* N '"">?X-REF?</A>' N | out: | xr1:",
  "| xr3: CASEI FIND :hdref| xr2: | xr3:"
li =,
  "| li1: CASEI FIND :li id="||,                        /* :li id= */
    "| SPEC '<P><A NAME=""' 1 8-* N '""><LI>' N | out: | li1:"
hp =,                                               /* :hp1.-:hp3. */
  "| CHANGE ':hp1.'<I>' | CHANGE ':ehp1.'</I>'",
  "| CHANGE ':hp2.'<B>' | CHANGE ':ehp2.'</B>'",
  "| CHANGE ':hp3.'<I><B>' | CHANGE ':ehp3.'</B></I>'",
  "| CHANGE ':HP1.'<I>' | CHANGE ':EHP1.'</I>'",
  "| CHANGE ':HP2.'<B>' | CHANGE ':EHP2.'</B>'",
  "| CHANGE ':HP3.'<I><B>' | CHANGE ':EHP3.'</B></I>'"
gmlhdr = ""
DO i = 1 TO 5                                               /* :hn. */
  gmlhdr = gmlhdr,
  "| h"i"1: CASEI FIND :h"i" id=| SPEC 8-* 1 | CHANGE '.' ' 1",
  "| SPEC '<A NAME=""' 1 W1 N '"">' N '<H"i">' N W2-* N '</H"i">' N",
    "| out: | h"i"1:",
  "| h"i"2: CASEI FIND :h"i"| CHANGE '.' ' 1",
  "| SPEC '<H"i">' N W2-* N '</H"i">' N | out: | h"i"2:"
END

dlhdr =,                                          /* :dthd., :ddhd. */
  "| d1: CASEI FIND :dthd| CHANGE '.' '",
    "| SPEC '<DT><B>' 1 W2-* N '</B>' N | out: | d1:",
  "| d2: CASEI FIND :ddhd| CHANGE '.' '",
    "| SPEC '<DD><B>' 1 W2-* N '</B>' N | out: | d2:"
```

```
/*d3 3-character DCF tag lookup table.
.bl    <BR><BR>
.br    <BR>
.fo    </PRE>
.nf    <PRE>
.hr    <HR>
.sk    <BR><BR><BR><BR>
.sp    <BR><BR><BR><BR>
d3*/

/*d5 5-character DCF tag lookup table.
.sk 0  <BR>
.sk 1  <BR><BR>
.sk    <BR><BR>
.sk 2  <BR><BR><BR>
.sp 0  <BR>
.sp 1  <BR><BR>
.sp    <BR><BR>
.sp 2  <BR><BR><BR>
d5*/

/*d6 6-character DCF tag lookup table.
.ce of </CENTER>
.ce on <CENTER>
.fo of <PRE>
.nf of </PRE>
d6*/

dcflookup =,
  "| l1: LOOKUP PAD BLANK ANYCASE 1-6 MASTER",
  "| l2: FANINANY",              /* All successful LOOKUPs come here. */
  "| SPEC 8-* 1",                   /* Replace tag with table entry. */
  "| out:",
  "| <" fn ft fm,                                        /* This file. */
  "| l3: INSIDE '/*d6' 'd6*/'",   /* Pick up 6-char DCF lookup table. */
  "| l1:",
,                    /* Not found in 6-char table.  Try 5-char table. */
  "| l4: LOOKUP PAD BLANK ANYCASE 1-5 MASTER",
  "| l2:",
  "| l3:",
  "| l5: INSIDE '/*d5' 'd5*/'",
  "| l4:",
,
  "| l6: LOOKUP PAD BLANK ANYCASE 1-3 MASTER",
  "| l2:",
  "| l5:",
  "| l7: INSIDE '/*d3' 'd3*/'",
  "| l6:"

/*g4 4-character GML tag lookup table.
```

```
:p     <P>
:li    <P><LI>
:dd    <DD>
:dl    <DL>
:dt    <DT>
:gd    <DD>
:gl    <DL>
:gt    <DT>
:lp    <P>
:lq    <BLOCKQUOTE>
:ol    <OL>
:pc    <P>
:ul    <UL>
:ul    <UL>
:xmp   <PRE>
:edl   </DL>
:egl   </DL>
:eol   </OL>
:esl   </UL>
:eul   </UL>
:elq   </BLOCKQUOTE>
:exm   </PRE>
g4*/

/*g5 5-character GML tag lookup table
:dl c  <DL COMPACT>
:gl c  <DL COMPACT>
:ol c  <OL COMPACT>
:ul c  <UL COMPACT>
:ul c  <UL COMPACT>
g5*/

gmllookup =,
  "| l8: LOOKUP PAD BLANK ANYCASE 1-5 MASTER",
  "| l2:",
  "| l7:",
  "| l9: INSIDE '/*g5' 'g5*/'",
  "| l8:",
  "| l1Ø: LOOKUP PAD BLANK ANYCASE 1-4 MASTER",
  "| l2:",
  "| l9:",
  "| INSIDE '/*g4' 'g4*/'",
  "| l1Ø:"

TRACE OFF
"STREAMSTATE OUTPUT 1" /* Does calling pipeline expect 2ndary output? */
TRACE NORMAL
IF RC = -4 | RC = 12 THEN output1 = "" /* If not, don't generate any. */
ELSE output1 = "| *.output.1:" /* If so, implant secondary connector. */
```

```
"CALLPIPE (END | NAME s2h) *.input:",
  "| STRIP TRAILING",
  "| a: CASEI FIND .CM#| CHOP STRING /;/ | SPEC 5-* 1",
,              /* In ".cm#xxx;yyy" output xxx verbatim; discard yyy. */
  "| out: FANINANY",               /* All processed output goes here. */
  "| *.output:",
,
  "| a:",                              /* Ordinary input (not ".cm#"). */
  "| b: NLOCATE W1",                        /* Select all blank lines. */
  "| SPEC '<BR><BR>' 1",                           /* Treat as if .bl. */
  "| out:",
,
  "| b:",                                  /* Nonblank lines come here. */
  "| CHANGE ';.'"'1E'X".' | SPLIT AT 1E",  /* Deconcatenate all lines */
  "| c: FIND .| XLATE 2-3 UPPER | SPLIT at ';'",    /* containing DCF */
,     /* control words and translate DCF control words to upper case. */
  "| d: FANINANY",                     /* All input data end up here. */
    hp,                               /* Do highlighted phrases first. */
  "| dcftags: NFIND ."||,
  "| gmltags: NFIND :"||,
  "| out:",             /* If neither DCF nor GML, just write it out. */
,
  "| c:",           /* Input data not starting with DCF control word. */
  "| d:",                     /* Merge with deconcatenated input data. */
,
  "| dcftags:",                          /* Process DCF control words. */
    dcfhdr,
    font,
    right,
    dcflookup,
    center,                 /* One-line centres must follow look-up. */
    output1,                       /* Unrecognized DCF tags go here. */
,
  "| gmltags:",                                    /* Process GML tags. */
    gmlhdr,
    dlhdr,
  "| SPLIT '.' | e: NFIND :| out: | e:",
    xref,
    li,
    gmllookup,
  "| out:"                          /* Unrecognized GML tags go here. */
 EXIT
```

Editor's note: readers wishing to discuss the material in this article can contact the author at bec@nysernet.org.

Ben Chi
NYSERNet (USA)

Working with long REXX strings

Editor's note: in the first article in this series, the authors examined how parse can be used to improve performance. In this article they will discuss how to handle long strings.

When writing REXX applications, there is often the need to build long strings and/or decompose long strings. Performance may be adversely affected by the way long strings are manipulated.

BUILDING UP STRINGS

EXECs often build a growing string in a DO loop. In this situation, we made some surprising discoveries, namely, that the straightforward solution becomes costly when the string being built is long, and solutions that might appear 'silly' perform very well.

Using this knowledge, we can reduce the response time of an OS/2 REXX application from 2 to 0.3 seconds. The application did use more than one 'long' string, but the lengths were not that special – between 2,000 and 13,000 bytes.

Firstly, let's show there really is a problem with the following straightforward coding:

```
files=''
do n
   files=files fn ft fm
end
```

Our test was as follows: the DO loop was executed three times, resulting in a string of 78 bytes. We then ran the DO loop 6 times, and the length of the string became 156, and so on. The relative performance table for VM and OS/2 is shown in Figure 1. The last row in the table shows the results of our 'better' code (we won't tell you now what the improvement was, so read on).

It is clear that for both VM and OS/2, the time required to build the string is not always proportional to the length of the string. REXX on OS/2 starts to lose with strings of 600 characters and with a length of 2,498 the situation becomes really bad.

String length	78	156	312	624	1248	2498	4992	9984	19968
Ideal time	100	200	400	800	1600	3200	6400	12800	25600
VM time required	100	173	314	696	1874	5147	13418	37609	111971
Improved VM code	105	185	401	846	1776	3770	11178	32803	96767
OS/2 time required	100	200	400	1133	6666	24166	171466	845000	3240000
Improved OS/2 code	110	212	415	883	3967	13383	63950	271667	1036667

Figure 1: Relative performance for VM and OS/2

REXX on VM starts very well, but, at 2,400 bytes, degradation is high. Compared to OS/2, REXX on VM still handles strings of 20,000 bytes very well.

We tried to solve this problem by looking for other coding techniques. Some time ago, we learned that performanance on VM can be improved by ‘pre-allocating the variable’ that will get the growing string:

```
files=left('',10000) /* Place 10000 spaces in var "files" */
files=''             /* Make "files" empty again */
do ...
   files=files fn ft fm
end
```

On VM, this improves performance. An EXEC with DO 1000 extends a string so that it gets a length of 26,000; we get the following results:

```
Interpreted, without preallocation: 100%, with preallocation: -30%
Compiled,    without preallocation: -59%, with preallocation: -61%
```

Interpreted, the gains are considerable. You can also see that simply compiling the EXEC improves performance even more. But, in a compiled EXEC, preallocating variables does not help. Why is this faster? On VM, REXX will not free the storage occupied by a variable when the variable is shortened; on the other hand, when a variable grows, REXX must get new storage for it and move the content. Because we knew this, it was already applied in the code that we used

to produce the numbers in the table in Figure 1. On OS/2 this technique does not help.

Let's look at five alternatives that can be used to add elements to a string:

```
do ...  /* Straight forward */
   files=files fn ft fm
end
do ...  /* use an intermediate variable */
   tt=fn ft fm
   files=files tt
end
        /* and, a very bad one: */
do ...  /*    use concatenation */
   files=files fn||' '||ft||' '||fm
end
do ...  /* use a "nop" function */
   files=files space(fn ft fm)
end
do ...  /* use parenthesis */
   files=files (fn ft fm)
end
```

Probably only the first and second alternatives are used very often. Until a while ago, I would have said that the first one is the best, and all others are silly, creating extra overhead. However, this is only true when the string to build remains short.

Indeed, some of the alternatives above *are* silly, and tests reveal that using them degrades performance even more. But, when looking for miracles, all possible alternatives have to be tried. In the test, we started with an empty string and made it longer, with the code variations shown above, until the string reached a certain length. On VM we measured one more length than on the PC platforms (on VM a string of 780 characters does not suffer from the 'long string problem'; at 2KB, the problem starts to appear).

How the five alternatives shown above perform with different string lengths is shown in Figure 2. We can draw the following conclusions from the results:

- The needless concatenation creates much overhead, on VM as well as on PCs, with short and long strings.
- Using an interim variable is almost always 'cheap' on the PC platform.

Environment	*VM*	*VM*	*VM*	*VM*
String length	*78*	*780*	*2,028*	*26,000*
Simply append	100%	100%	100%	100%
With an interim variable	+33%	+27%	+11%	-14%
With concatenation	+51%	+61%	+56%	+25%
With SPACE()	+102%	+93%	+62%	-2%
With parenthesis	+4%	+9%	-6%	-19%

Figure 2a: Performance with different length strings

- With SPACE, the overhead for short strings is very high. For long strings though it helps, on PCs a lot.
- Our 'wonderful' solution is using parenthesis – little or no overhead for short strings, a great help for long strings.
- When comparing the straightforward solution with the best, and

Environment	*OS/2*	*OS/2*	*OS/2*	*W95*	*W95*	*W95*
String length	*78*	*780*	*26,000*	*78*	*780*	*26,000*
Simply append	100%	100%	100%	100%	100%	100%
With interim variable	+16%	-53%	-59%	-1%	-44%	-66%
With concatenation	+32%	+60%	+64%	+109%	+168%	+167%
With SPACE()	+132%	-13%	-66%	+92%	-11%	-65%
With parenthesis	+0%	-60%	-68%	+0%	-45%	-66%

Figure 2b: Performance with different length strings

worst one, on OS/2 the best one performs three times faster for long strings.

So, how did we come to our ‘best’ solution?

The ‘long strings’ overhead can be explained as follows. When you append three items to a string, REXX gets storage, appends the first item, then it finds another item to append, gets more storage, etc.

For short strings, this is no problem, but when the string becomes long, the overhead is high. So we searched for ways to have REXX treat the stuff to append as one item. An interim variable is the obvious path. This helps, but we kept searching because an extra variable surely adds some overhead. So we tested with the concatenation. I thought of using SPACE, and, during tests with long strings, SPACE helped a lot (but not with short strings). Explaining the SPACE solution to a colleague, I suddenly had the idea of using simple parenthesis – and, yes, that’s the best for long strings. With the parenthesis, REXX first joins the three items and appends that to the string.

As a conclusion for building strings, whenever you code an EXEC with growing strings, use the parenthesis technique.

Having seen how to build long strings, let’s discuss how to ‘walk through’ the elements of long strings.

HANDLING STRING ELEMENTS

Often, EXECs must ‘walk’ through strings and handle the elements. Various techniques exist and, once again, there are good and bad performers. Once more, a good technique for short strings may be bad for long strings. On VM, I found that ‘eating it’ with parse worked well, and surely faster than using WORD(string,i). An example of both styles follows:

```
do i=1 to words(ftypes)
   ftype=word(ftypes,i)
   ...
end
```

As mentioned in the first article in this series, WORD() becomes very costly to get the nth word when ‘n’ is high. With parse, the string held

in ‘ftypes’ can only be processed once. If more than one pass is required, we’d code:

```
ToEat=ftypes
do while ToEat<>''
   parse var ToEat ftype ToEat
   ...
end
```

This technique also works when the elements of the strings hold more than a single word, such as in:

```
ToEat=files
do while ToEat<>''
   parse var ToEat fname ftype fmode ToEat
   ...
end
```

With our OS/2 application, we found that the ‘eating’ technique no longer performed well when the string became long. As with building strings, on VM this effect exists as well, but the strings can be much longer before performance degrades.

The good news is that, with an appropriate technique, performance can be dramatically improved. In the test case we made to perform our measurements, the elements placed in the string contained three words – filename, filetype, and filemode – and we tested with strings of various lengths – 63 characters (3 elements), 630 characters, 6,300, and 63,000 characters (3,000 elements).

The code to perform our tests was as follows:

```
/* Test performance of "walking" through a string                    */
/* Format:  EATIT nbElements nbIterations                            */
/*  where:   nbElements  = number of elements to place in the string */
/*           nbIterations= number times we walk through the string   */
parse arg n m . ; if n='' then n=1000 ; if m='' then m=10
parse source OpSys .
b='PROFILE  EXEC     A2' /* An element to place in the string */
files=''              /* Build the string we'll "walk" though */
do n
   files=b files
end
say OpSys':' m 'Times through string with lng='length(files)

/*********************************************************************/
call CpuTime 'R'
```

```
do m                    /* maybe repeat our test a couple of times */
   ToEat=files
   do while ToEat<>''
      parse var toEat fn ft fm ToEat
   end
end
t2=CpuTime('E')
say format(t2,3,3) 'parse var toEat fn ft fm ToEat'
.... etc for the other techniques ......
```

How did this behave? On OS/2 (Pentium 133MHz) 'walking' through a string of 63,000 characters with the 'eat' technique cost 4.65 seconds, which we could reduce to 0.09 seconds. On VM (IBM 9672/R65), 'eating it' cost 0.33 CPU seconds, and could be reduced to 0.07 seconds.

Let's look at the possibilities we found to 'walk' through a string. For each alternative, we directly examined how it performed in relation to the 'eat' solution for a string of 63,000 characters. First, the two alternatives encountered most often:

```
/*-1- eat the string ******************************************/
/*    VM:  =100% OS/2:  =100% W95:  =100%                     */
   ToEat=files
   do while ToEat<>''
      parse var toEat fn ft fm ToEat
   end
/*-2- use WORD() ***********************************************/
/*    VM: +7988% OS/2:  +388% W95:  +419%                     */
   do i=1 by 3 to words(files)
      fn=word(files,i);ft=word(files,i+2) ; fm=word(files,i+2)
   end
```

The 'eat' technique works well, except with long strings. The second solution, based on WORD(), should be thrown away. It behaves very badly with long strings (on OS/2, 22 seconds for a 63,000 character string; on VM, 22 CPU seconds were required). This solution is included to convince you that it should not be used; in none of our tests was it faster than 'eating'.

To solve our problem, we started looking for faster techniques.

Below, we show several alternatives. On PCs, most are better for long strings, but they all require that the elements in the string have a fixed length. In our test case, an element counts 21 characters and contains filename, filetype, and filemode. Not all solutions produce exactly the

same result: the words extracted may or may not contain trailing blanks, and, depending on the application, these trailing blanks may or may not matter. For example:

```
fytpe='EXEC'
data.ftype='xyz'       /* Creates var "DATA.EXEC" */
ft='EXEC    '
if data.ft='xyz' then  /* References var "DATA.EXEC    " */
```

So, bear this difference in mind while reading our alternatives.

```
/*-3- parse with columns ************************************/
/*    VM:   +48% OS/2:   -98% W95:   -98%                   */
   do c=0 by 21 to length(files)-1
      parse var files +(c) fn +9 ft +9 fm +2
   end
/*-4- but, above "fn" gets 9 chars; take only 8 ***************/
+*    VM:   +48% OS/2:   -97% W95:   -98%                   */
   do c=0 by 21 to length(files)-1
      parse var files +(c) fn +8 +1 ft +8 +1 fm +2
   end
/*-5- but, now "fn" maybe have trailing spaces: remove them ****/
/*    VM:   +52% OS/2:   -97% W95:   -98%                   */
   do c=0 by 21 to length(files)-1
      parse var files +(c) fn . +9 ft . +9 fm +2
   end
/*-6- maybe this works fast ? *********************************/
/*    VM:   +39% OS/2:  +4.3% W95:   -98%                   */
   do c=0 by 21 to length(files)-1
      parse var files +(c) fn ft fm .
   end
/*-7- or is this thing fast ? *********************************/
/*    VM:   +42% OS/2:   -98% W95:   -98%                   */
   do c=0 by 21 to length(files)-1
      parse var files +(c) fn ft fm . +21
   end
/*-8- use SUBSTR() and STRIP() ********************************/
/*    VM:   -58% OS/2:   -94% W95:   -96%                   */
   do c=1 by 21 to length(files)
      fn=strip(substr(files,c,8),'T')
      ft=strip(substr(files,c+9,8),'T')
      fm=substr(files,c+18,2)
   end
/*-9- take a small piece and parse that ***********************/
/*    VM:   -79% OS/2:   -97% W95:   -97%                   */
   do c=1 by 21 to length(files)
      fid=substr(files,c,21); parse var fid fn ft fm .
   end
```

Some conclusions:

- On VM, all except for the last two solutions are worse than ‘eating’.
- On OS/2 with classic REXX, there must be some performance bug – solution ‘-6-’ is even slower than ‘eating’ the string (with OO-REXX on Windows 95 or OS/2, solution ‘-6-’ works very well too).

Surprisingly, the last solution with a temporary variable works well everywhere.

The drawback for these solutions is that all elements in the string need to have a fixed length. The following solutions don’t have this requirement:

```
/*-10- ********************************************************/
/*VM:   -42% OS/2:   -93% W95:   -95%                           */
   b=space(files) /* Avoid extraneous spaces */
   do c=1 to length(b)
      tt=subword(substr(b,c,50),1,3)
      fn=word(tt,1) ; ft=word(tt,2) ; fm=word(tt,3)
      c=c+length(tt)
   end
/*-11- ********************************************************/
/* VM:   -61% OS/2:   -94% W95:   -95%                          */
   b=space(files) /* Avoid extraneous spaces */
   do c=1 to length(b)
      tt=subword(substr(b,c,50),1,3)
      parse var tt fn ft fm .
      c=c+length(tt)
   end
/*-12********************************************************/
/* VM:   -64% OS/2:   -95% W95:   -95%                          */
   b=space(files) /* Avoid extraneous spaces */
   do c=1 to length(b)
      parse value substr(b,c,50) with fn ft fm .
      c=c+length(fn ft fm)
   end
```

These solutions also work well. How did we come to them? As we now know that REXX has problems with long strings, in these solutions we first extract a small piece and than unravel that further. The above code is general, the elements can have varying length, but no element must have more than 50 characters.

Note: we use LENGTH(), and warned you in the previous article

Environment	*VM*	*VM*	*VM*	*VM*
String length	*63*	*630*	*6300*	*63000*
-1-Eating	100%	100%	100%	100%
-2-WORD()	+206%	+720%	+3338%	+7988%
-3-parsefiles+(c)n+9t+9m+2	+41%	+33%	+50%	+48%
-4-parsefiles+(c)n+8+1t+8+1m+2	+51%	+43%	+50%	+48%
-5-parsefiles+(c)n.+9t.+9m+2	+49%	+39%	+50%	+52%
-6-parsefiles+(c)ntm.	+33%	+24%	+38%	+39%
-7-parsefiles+(c)ntm.	+21%	+37%	+28%	+50%
-8-n=strip(substr(files,c,8),T);t=strip(...)	+229%	+202%	+75%	-58%
-9-fid=substr(files,c,21);parsevarfidntm	+78%	+50%	-13%	-79%
-10-space();t=subword(substr(b,c,50),1,3);n=word(t,1);...;c=c+l(t)	+335%	+304%	+138%	-42%
-11-space();t=subword(substr(b,c,50),1,3);parsevartn...;c=c+l(t)	+222%	+191%	+75%	-61%
-12-space();parsesubstr(b,c,50)withntm;c=c+lng(ntm)	+180%	+150%	+38%	-64%

Figure 3: Performance figures for VM

against using WORDS() because that requires scanning. But, even on PCs, the LENGTH() function performs well – as opposed to C-style strings that are ended with an 'X00' character, REXX stores the length of its strings, so it is available without scanning.

As you can imagine, the length of the string often greatly influences the performance of the various techniques. Figure 3 shows different lengths and solutions for VM.

Remember that techniques 3 to 9 can be used only when the elements of the string have a fixed length. If your own EXEC is constructing this string, it may be possible to add elements with a fixed length. When you look back at the fastest way to build long strings you can see that coding:

```
files=files left(fn ft fm,21)
```

can be faster than what you probably code now:

```
files=files fn ft fm
```

CONCLUSION

For VM, use the 'eat' technique for strings that are not expected to become longer than about 6,000 characters. For longer strings, it becomes worthwhile using another technique (number 12) or, with fixed length elements, number 9.

On the PC platform, with classic REXX, use 'eat' for strings up to about 600 characters. With OO-REXX, even for shorter strings, the other techniques behave better than 'eat'.

But don't forget our 'Lesson 1' from the first article in this series. In our tests, an element is composed of three words. Based on this study, it should be clear that when elements have, for example, 10 words, the best technique may be one other than the 'best' one listed in the table above. You should use an EXEC as we did to compare the techniques.

Kris Buelens and Guy De Ceulaer
Advisory Systems Engineers
IBM (Belgium)

EXCEL in REXX

INTRODUCTION

EXCELRXX is a CMS EXEC that combines SQL query partial results into an EXCEL table and calculates totals. Specification of width and title is supported for each dynamically generated column.

EXCELRXX can process any number of input files and creates the resulting table with no limitation on its size. Each file can be either formatted or not formatted.

EXCELRXX EXEC USAGE

EXCELRXX EXEC does not have set parameters.

Input files and the parameters of the output table are defined during interactive dialog. Input files must reside on the A mini-disk. They should be grouped together by filename and filetype.

The group of files contains data for the corresponding column in the generated table and is identified by its filename. The filename is the same for all files in a given group.

Inside the group, files differ by filetype, which is an integer number, starting at 1, with an increment of 1.

For each group of files, an index and a value field are defined. The index field controls the generation of a row in the resulting table. The content of the value field is accumulated in a given column, selected by index row. This means that the value field must always contain a number, not text.

For formatted files, index and value fields are declared as usual – by first and last position in the record, or by first position in the record and length.

For unformatted files (eg print files), fields are processed as words. In this case, index and value fields are declared by the number of corresponding words in the record.

The following conditions must be taken into consideration before using EXCELRXX:

- All input files must be sorted in ascending order by the selected index field.
- As a temporary area, EXCELRXX uses CMS files named $EXCEL$ <numbr> A. These files are erased during EXCELRXX execution.

Example of input files, divided into two groups to create two columns in the generated table, are:

- Group WEEKLY contains 2 files (WEEKLY 1 A and WEEKLY 2 A). 'NAME' is selected as the index field and 'SALARY WEEKLY' as the value field:

```
file WEEKLY 1 A

EMPNO NAME JOB SALARY WEEKLY
──────────────────────────────
1     AA    2       3
1     AA    2       4
2     BB    2       5

file WEEKLY 2 A

EMPNO NAME JOB SALARY WEEKLY
──────────────────────────────
2     BB    2       6
```

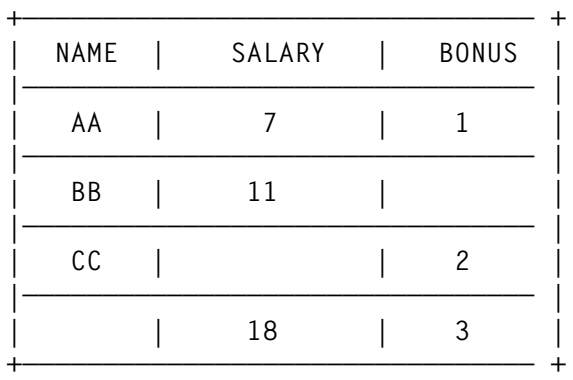
- Group BONUS contains 1 file (BONUS 1 A). NAME is selected as the index field and BONUS as the value field.

```
file  1 A

DEPT NAME  BONUS
────────────────
1     AA     1
1     CC     2
```

When the following declarations are made, EXCELRXX will generate the table shown in Figure 1:

- NUMBER of columns in result table – 2.
- INDEX and VALUE fields definition – by word, sequential number.



NAME	SALARY	BONUS
AA	7	1
BB	11	
CC		2
	18	3

Figure 1: Example of table generated by EXCELRXX

- NAME and WIDTH for INDEX column – NAME, 10.
- NAME and WIDTH for column 1 – SALARY, 14.
- FILE NAME and NUMBER of files for column 1 – WEEKLY, 2.
- WORD NUMBER of INDEX and VALUE for column 1 – 2, 4.
- NAME and WIDTH for column 2 – BONUS, 8.
- FILE NAME and NUMBER of files for column 2 – BONUS, 1.
- WORD NUMBER of INDEX and VALUE for column 2 – 2, 3.

EXCELRXX EXEC

```
/*********************************************************************/
/***                                                    ***         ***/
/*** EXCELRXX              EXCEL in REXX                 ***         ***/
/***                                                    ***         ***/
/*********************************************************************/
  SIGNAL ON SYNTAX
  SET CMSTYPE HT
  ERASE $EXCEL$ '*' A
  STATE TABLE GEN A
  RC_SAVE = RC
  SET CMSTYPE RT
  CLRSCRN
  DO 9
```

```
  SAY
END
IF RC_SAVE = Ø THEN
DO
  SAY '— found' TABLE GEN A
  SAY '— Enter 1/Yes/ to erase'
  PULL ANS
  IF ANS ¬= 1 THEN
  EXIT
  ERASE TABLE GEN A
END
DO UNTIL DATATYPE(N_COLS, 'W')
  SAY '— Enter NUMBER of columns in result table, Ø-exit'
  PULL N_COLS
  IF N_COLS = Ø THEN
  EXIT
END
DO UNTIL VERIFY(FMT, '123') = Ø
  SAY '— INDEX and VALUE fields definition -'
  SAY '    1 - by word seq number in file records'
  SAY '    2 - by starting and ending column'
  SAY '    3 - by starting column and length'
  SAY '— Enter 1/2/3 or Ø-exit'
  PULL FMT
  IF FMT = Ø THEN
  EXIT
  FMT = LEFT(FMT, 1)
END
TAB_WIDTH = Ø
DO N_COL = 1 TO N_COLS
  IF N_COL = 1 THEN
  DO
    DO UNTIL DATATYPE(COL_WIDTH.Ø, 'W')
      SAY '— Enter NAME and WIDTH for INDEX column, Ø-exit'
      PULL COL_NAME.Ø COL_WIDTH.Ø
      IF COL_NAME.Ø = Ø THEN
      EXIT
    END
    TAB_WIDTH = TAB_WIDTH + COL_WIDTH.Ø
  END
  CLRSCRN
  DO 1Ø
    SAY
  END
  SAY '                    >>> Column' N_COL 'processing <<<'
  DO UNTIL DATATYPE(COL_WIDTH.N_COL, 'W')
    SAY '— Enter NAME and WIDTH  for' N_COL 'column, Ø-exit'
    PULL COL_NAME.N_COL COL_WIDTH.N_COL
    IF COL_NAME.N_COL = Ø THEN
    EXIT
```

```
END
TAB_WIDTH = TAB_WIDTH + COL_WIDTH.N_COL + 1
DO UNTIL DATATYPE(N_FILES, 'W')
  SAY '- Enter FILE NAME and NUMBER of files for' N_COL        ,
      'column, 0-exit'
  PULL FN N_FILES
  IF FN = 0 THEN
  EXIT
  IF DATATYPE(N_FILES, 'W') THEN
  DO I = 1 TO N_FILES
    SET CMSTYPE HT
    STATE FN I A
    RC_SAVE = RC
    SET CMSTYPE RT
    IF RC_SAVE = 0 THEN
    ITERATE
    SAY '- not found ' FN I A
    N_FILES = A
    LEAVE
  END
END
DO UNTIL DATATYPE(N_INDEX, 'W') & DATATYPE(N_VALUE, 'W')
  IF FMT = '1' THEN
  SAY '- Enter WORD NUMBER of INDEX and VALUE in record'       ,
      'for' N_COL 'col, 0-exit'
  ELSE
  IF FMT = '2' THEN
  SAY '-Enter starting and ending column of INDEX'             ,
      'and VALUE for' N_COL 'col, 0-exit'
  ELSE
  SAY '-Enter starting column and length of INDEX'             ,
      'and VALUE for' N_COL 'col, 0-exit'
  IF FMT = '1' THEN
  PULL N_INDEX N_VALUE
  ELSE
  DO
    PULL N_INDEX N_VALUE S_INDEX S_VALUE
    IF ¬ (DATATYPE(S_INDEX, 'W') & DATATYPE(S_VALUE, 'W')) THEN
    N_VALUE = A
  END
  IF N_INDEX = 0 THEN
  EXIT
END
IF FMT = '2' THEN
DO
  N_VALUE = N_VALUE - N_INDEX + 1
  S_VALUE = S_VALUE - S_INDEX + 1
END
SAY '- Processing of files for' N_COL 'column   at' TIME(L)
DO I = 1 TO N_FILES
```

```
    EXECIO '*'  DISKR FN I A '(FINI STE ' BUF.I.
    I.I = 1
  END
  DO FOREVER
    PROC_INDEX = 'FFFFFFFFFFFFFFFFFFFF'X
    DO I = 1 TO N_FILES
      IF I.I <= BUF.I.Ø THEN
      DO
        J = I.I
        IF FMT = '1' THEN
        INDEX.I = WORD(BUF.I.J, N_INDEX)
        ELSE
        INDEX.I = SUBSTR(BUF.I.J, N_INDEX, N_VALUE)
        IF PROC_INDEX > INDEX.I THEN
        PROC_INDEX = INDEX.I
      END
      ELSE
      INDEX.I = 'FFFFFFFFFFFFFFFFFFFF'X
    END
    IF PROC_INDEX = 'FFFFFFFFFFFFFFFFFFFF'X THEN
    LEAVE
    PART_SUM = Ø
    DO I = 1 TO N_FILES
      IF PROC_INDEX = INDEX.I THEN
      DO FOREVER
        J = I.I
        IF FMT = '1' THEN
        VALUE = WORD(BUF.I.J, N_VALUE)
        ELSE
        VALUE = SUBSTR(BUF.I.J, S_INDEX, S_VALUE)
        PART_SUM = PART_SUM + VALUE
        J = J + 1
        I.I = J
        IF J > BUF.I.Ø THEN
        LEAVE
        IF FMT = '1' THEN
          IF PROC_INDEX ¬= WORD(BUF.I.J, N_INDEX) THEN
        LEAVE
        IF FMT ¬= '1' THEN
          IF PROC_INDEX ¬= SUBSTR(BUF.I.J,N_INDEX,N_VALUE) THEN
        LEAVE
      END
    END
    EXECIO 1 DISKW $EXCEL$ N_COL A '(' STR PROC_INDEX PART_SUM
  END
  DROP BUF.
  SAY '— Input files processed              at' TIME(L)
  PULL
END
CLRSCRN
```

```
DO 19
  SAY
END
SAY '— Begin table generation             at' TIME(L)
EXECIO 1 DISKW TABLE GEN A '(' STR '+' ||                          ,
                          COPIES('-', VALUE(TAB_WIDTH)) || '+'
TITLE = '|' || CENTER(COL_NAME.Ø, COL_WIDTH.Ø) || '|'
DO N_COL = 1 TO N_COLS
  EXECIO '*'  DISKR $EXCEL$ N_COL A '(FINI STE ' BUF.N_COL.
  TITLE = TITLE || CENTER(COL_NAME.N_COL, COL_WIDTH.N_COL) || '|'
  I.N_COL = 1
  SUM.N_COL = Ø
END
ERASE $EXCEL$ '*' A
EXECIO 1 DISKW TABLE GEN A '(VAR TITLE'
EXECIO 1 DISKW TABLE GEN A '(' STR '|' ||                           ,
                          COPIES('-', VALUE(TAB_WIDTH)) || '|'
DO FOREVER
  PROC_VALUE = 'FFFFFFFFFFFFFFFFFFFF'X
  DO N_COL = 1 TO N_COLS
    IF I.N_COL ¬> BUF.N_COL.Ø THEN
    DO
      I = I.N_COL
      VALUE.N_COL = WORD(BUF.N_COL.I, 1)
      IF PROC_VALUE > VALUE.N_COL THEN
      PROC_VALUE = VALUE.N_COL
    END
    ELSE
    VALUE.N_COL = ''
  END
  IF PROC_VALUE = 'FFFFFFFFFFFFFFFFFFFF'X THEN
  DO
    EXECIO 1 DISKW TABLE GEN A '(' STR '|' ||                       ,
                          COPIES('-', VALUE(TAB_WIDTH)) || '|'
    ROW = '|' || COPIES(' ', COL_WIDTH.Ø) || '|'
    DO N_COL = 1 TO N_COLS
      ROW = ROW || RIGHT(SUM.N_COL, COL_WIDTH.N_COL) || '|'
    END
    EXECIO 1 DISKW TABLE GEN A '(' STR ROW
    EXECIO 1 DISKW TABLE GEN A '(' STR '+' ||                       ,
                             COPIES('-', VALUE(TAB_WIDTH)) || '+'
    ROW = '|' || RIGHT(PROC_VALUE, COL_WIDTH.Ø) || '|'
    SAY '— Table generation successful        at' TIME(L)
    EXIT
  END
  ROW = '|' || RIGHT(PROC_VALUE, COL_WIDTH.Ø) || '|'
  DO N_COL = 1 TO N_COLS
    IF PROC_VALUE = VALUE.N_COL THEN
    DO
      I = I.N_COL
```

```
        VALUE = WORD(BUF.N_COL.I, 2)
        I.N_COL = I.N_COL + 1
        SUM.N_COL = SUM.N_COL + VALUE
        ROW = ROW || RIGHT(VALUE, COL_WIDTH.N_COL) || '|'
      END
      ELSE
      ROW = ROW || RIGHT(' ', COL_WIDTH.N_COL) || '|'
    END
    EXECIO 1 DISKW TABLE GEN A '(' STR ROW
  END
SYNTAX:
  SAY '+++ EXELRXX abend due to data check    at' TIME(L)
  SAY '>>> bad number follows                 at' TIME(L)
  SAY VALUE
```

Dobrin Goranov
Information Services Co (Bulgaria)

A full screen console interface – part 17

Editor's note: the following article is an extensive piece of work which will be published over several issues of VM Update. *It was felt that readers could benefit from the entire article and from the individual sections. Any comments or recommendations would be welcomed and should be addressed either to Xephon or directly to the author at fernando_duarte@vnet.ibm.com.*

BACKWARD HELPCSC

```
.cm VM Software Services
.cm
.cs Ø on
                                []CSC Tool[%
[%
[%
 Use the BACKWARD command to scroll backward the CSC log file.
   EXAMPLE: BACKWARD 3
            Scroll backward 3 screens of data.

.cs Ø off
.cs 1 on
                                []CSC Tool[%
[]Purpose[%
```

```
 Use the BACKWARD command to scroll backward the CSC log file.
.cs 1 off
.cs 2 on
[]Format[%
                    .-1-.
 >>-.'Backward-.-+-+-------------------------------><
     '-BWD---'   '-n-'

[]Class[%

 3
.cs 2 off
.cs 3 on
[]Operands[%

 n
     number of screens to scroll. The default is one.
.cs 3 off
.cs 4 on
.cs 4 off
.cs 5 on
[]Usage Notes[%

   1.   PFØ7 and PF19 perform the command BACKWARD 1.

   2.   BWD is a synonym for BACKWARD.

   3.   The reference line for the BACKWARD command is the first line on
        screen. This may not be the expected result if you are on the
        current screen and have messages with the HOLD attribute. Using
        the BOTTOM command before solves this problem.

.cs 5 off
.cs 6 on
[]Messages[%
 Ø311E  Invalid BACKWARD operand: operand
 Ø312E  Unexpected BACKWARD operand: operand
.cs 6 off
```

BOTTOM HELPCSC

```
.cm VM Software Services
.cm
.cs Ø on
                                  []CSC Tool[%
[%
[%
 Use the BOTTOM command to show the last screen of data.
   EXAMPLE: BOTTOM
            Show last screen of data.
.cs Ø off
.cs 1 on
```

```
                                []CSC Tool[%
[]Purpose[%

 Use the BOTTOM command to show the last screen of data.
.cs 1 off
.cs 2 on
[]Format[%

 >>─BOTtom──────────────────────────────────><

[]Class[%

 3
.cs 2 off
.cs 3 on
.cs 3 off
.cs 4 on
.cs 4 off
.cs 5 on
[]Usage Notes[%

   1.   PFØ5 and PF17 are assigned to the BOTTOM command.

   1.   BOTTOM is a Browse command. The screen is not refreshed when new
        messages are received by the service machine.

.cs 5 off
.cs 6 on
[]Messages[%
 Ø312E  Unexpected BOTTOM operand: operand
.cs 6 off
```

BWD HELPCSC

```
.cm VM Software Services
.cm
.cs Ø on
                                []CSC Tool[%
[%
[%
 Use the BWD command to scroll backward the CSC log file.
   EXAMPLE: BWD 3
            Scroll backward 3 screens of data.

.cs Ø off
.cs 1 on
                                []CSC Tool[%
[]Purpose[%

 Use the BWD command to scroll backward the CSC log file.
.cs 1 off
```

```
.cs 2 on

[]Format[%

          .-1-.
 >>-BWD-+-+-----------------------------------><
          '-n-'

[]Class[%

 3
.cs 2 off
.cs 3 on
[]Operands[%

 n
     number of screens to scroll. The default is one.

.cs 3 off
.cs 4 on
.cs 4 off
.cs 5 on
[]Usage Notes[%

   1.   PFØ7 and PF19 perform the command BWD 1.

   2.   BWD is a synonym for BACKWARD.

   3.   The reference line for the BACKWARD command is the first line on
        screen. This may not be the expected result if you are on the
        current screen and have messages with the HOLD attribute. Using
        the BOTTOM command before solves this problem.

.cs 5 off
.cs 6 on
[]Messages[%
 Ø311E  Invalid BWD operand: operand
 Ø312E  Unexpected BWD operand: operand
.cs 6 off
```

CLEAR HELPCSC

```
.cm VM Software Services
.cm
.cs Ø on
                                  []CSC Tool[%
[%
[%
 Use the CLEAR command to remove all scrollable lines from the screen.
```

```
   EXAMPLE: CLEAR
            Clear scrollable lines.

.cs Ø off
.cs 1 on
                                  []CSC Tool[%
[]Purpose[%

 Use the CLEAR command to remove all scrollable lines from the screen.
.cs 1 off
.cs 2 on

[]Format[%
 >>—Clear———————————————————————————><

[]Class[%

 2
.cs 2 off
.cs 3 on
.cs 3 off
.cs 4 on
.cs 4 off
.cs 5 on
[]Usage Notes[%

   1.   CLEAR is only valid from a Current screen when the CMS switch is
        ON.

.cs 5 off
.cs 6 on
[]Messages[%
 Ø312E  Unexpected CLEAR operand: operand
 Ø335E  CLEAR is valid only in Refresh mode with CMS scroll ON
 Ø336E  CLEAR only works if CMS scroll is active
.cs 6 off
```

CMS HELPCSC

```
.cm VM Software Services
.cm
.cs Ø on
                                  []CSC Tool[%
[%
[%
 Use the CMS command to execute a local CMS command.
   EXAMPLE: CMS QUERY DISK

            Execute CMS command QUERY DISK.
```

```
.cs Ø off
.cs 1 on

                                []CSC Tool[%
[]Purpose[%

 Use the CMS command to execute a local CMS command.
.cs 1 off
.cs 2 on

[]Format[%

 >>─CMS─.────.──────────────────────────────><
          '-command-'

[]Class[%

 Ø

.cs 2 off
.cs 3 on
[]Operands[%

 command
     any CMS command. If omitted CMS subset environment is entered.

.cs 3 off
.cs 4 on
.cs 4 off
.cs 5 on
[]Usage Notes[%

   1.   This is a local command, executed by the user program (CSCUSR).
        The service machine is not informed.

.cs 5 off
.cs 6 on
[]Messages[%

 Ø284W  CMS command ended with non-zero return code
.cs 6 off
```

Editor's note: subscribers can download the remaining code for this article from www.sdsusa.com/vmupdate.html.

Fernando Duarte
Analyst (Canada)

IBM has announced Version 2.2 of its VisualAge PL/I family for OS/390, replacing the PL/I for VM host compiler product.

There's been an easing of previous restrictions in PL/I VM, plus enhancements introduced in the workstation PL/I version, as well as an automated date windowing technique via the Millennium Language Extensions.

A debug tool is included that supports COBOL for VM and PL/I for VM.

For further information contact your local IBM representative.

* * *

Tivoli has announced Version 3.7 of its Storage Manager, S/390 Edition, which replaces Version 3.1 of ADSM for VM/ESA.

OS/390 servers get better usability, performance, and LAN-free data movement.

There's now rapid recovery using portable back-up sets, instant archiving, full filesystem and raw logical volume back-ups, exploitation of back-up-archive client multi-threading, and dynamic self-tuning of performance parameters.

Full filesystem or raw logical volume images can be backed up and are managed as a single object like any other object on the server. The function will be implemented on AIX 4.3, HP-UX 11.0, and Solaris 2.6 back-up-archive clients. A new option allows the user to perform an incremental by-image-date back-up.

Storage Manager supports tape library sharing in a SAN for SCSI libraries on NT-based servers with IBM 3570 and STK PowerVault 130T servers for LAN-free back-up and recovery.

It uses an adaptive algorithm said to optimize performance for individual customer environments while minimizing administrative intervention.

For further information contact your local IBM representative.

* * *

IBM has announced plans to migrate VM users to OS/390 with a free upgrade to the operating system and middleware software currently installed on the new Multiprise 3000s.

The OS/390 Workload Transition Offering for VM and VSE deal lasts for 36 months from the installation date.

It's aimed at migrating VM and VSE users to the e-business capabilities of OS/390, including electronic transaction processing, ERP, and business intelligence applications not available on VM and VSE.

More generally, it's also designed to make it easier and cheaper to shift application workloads from VM and/or VSE to OS/390. The scheme covers all Multiprise 3000 models and any other processor eligible for Growth Opportunity Licence Charge (GOLC).

For further information contact your local IBM representative.