OS-9 USERS GROUP MOTD
"DEDICATED TO EXCELLENCE IN OS-9 COMPUTING"

Officers:
President: Boisy Pitre
Vice President: Carl Kreider
Secretary/Tres: Debi Kreider
Librarian: Zack Sessions
MOTD Editor: Alan Sheltra

Winter Issue 1992 | The Official Voice of the OS-9 Users Group | Volume 1, Number 3

from the OS-9 Users Group

Editorial by Alan Sheltra

It's the holiday season already! Where did the time go? Well, we've got some goodies for you this month.

Hope you haven't stuffed your "guts" with too much turkey this past Thanksgiving, 'cause we have some desert for your OS-9 box.

A hands-on review of the new UltraScience port of OS-9 for the Macintosh. This is good news for Mac owners, who can now have their favorite operating system on their Macs.

We continue our articles from last month on Awk and Bawk.

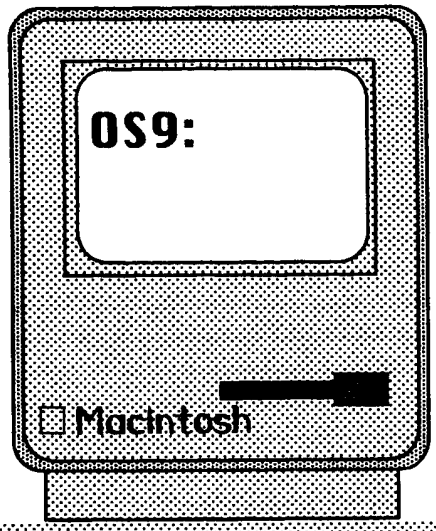
The OS-9 Users Group has a new Librarian. Scott McGee had to step down due to personal reasons. Welcome Zack Sessions to fill that role now. **Welcome aboard Zack!**

I should also mention that the OS-9 Users Group now has a new (and Final!) mailing address:

P.O. Box 71131, Des Moine, IA 50325

Until next time I hope you have a Happy and Safe Holiday Season!

Alan Sheltra - MOTD Editor



REVIEW OF ULTRASCIENCE'S OS-9 FOR THE MAC

by Mark Heilpern

Recently, I had the pleasure of examining a port of OS-9, for the Macintosh computer. This port is released by UltraScience/Gibbs Laboratories. My review is based on my experiences with OS-9 on a Mac Powerbook 140, System 7.0.1, and OS-9 V2.4.

The port seems to be put together rather well. The basic scheme of things is that to start up OS-9, you merely launch an application, which does some preliminary setup to place OS-9 in control of the CPU, and still allows the Finder (or, Multi-Finder) to run in the background. Also, any

(Continued Page Three)

Directory of MOTD_WINTER :

From the Desk of Carl...

An Editorial

by User Group Vice President
Carl Kreider

Many of you will recall that I was the assistant librarian and then librarian for the now defucnct **OS-9 User Group**. I was Founding Member #39. I noted the passing of the group with more than a bit of sadness. I had put a lot of time and energy into the group. But the pain wasn't from only selfish reasons, I felt like the UG provided a useful service. The library was the best repository of public domain software for OS-9. Later the online services accumulated quite a bit (including that of the UG), but in the beginning, the UG was all there was. We had a newsletter (admittedly sparodic) that helped keep us up on events. Later the 68 Micro Journal began to cover a bit of OS-9, as did Rainbow, but in the beginning, the UG was all there was.

And now we seem to have come full circle. Rainbow is all but gone. Don Williams (and apparently 68 Micro Journal) is gone. Tandy has orphaned the CoCo. Compuserve, for one, has cut back support for OS-9 in favor of DOS. So it seemed like there was a need for an OS-9 User Group again.

Sure, the guys with the OS9CN were trying to fill the void, but there are a lot of folks out there who aren't on FIDO. So when Boisy asked me to help, I agreed to become Vice President. It has been a bit difficult to get going, but we are moving forward. We now have about 116 members (including overseas members), and continue to grow. The **MOTD** looks better than it ever has. The library is beginning to shape up nicely.

Contributions are starting to roll in. We plan to continue to provide a way for the remaining OS-9 lovers to stay in contact with each other and with vendors. We plan to continue to provide a way for the faithful to share their software creations. And if CDI should take off, or OS-9000 become a cost effective option for 486 mavens, we could see a great swelling of the ranks. All in all, I see a great future for the OS-9 in general and the OS-9 User Group in particular.

Carl Kreider

Usenet: uunet!rde!gator!syscon!carl
CIS: 71076,76
Internet : carl@syscon.rn.com

(Continued from Page One)
application that is Multi-Finder tolerant can run as well.

Rather than require an external disk or physical disk partitioning, UltraScience took an interesting approach where you use pseudo-disks as hard devices, which are seen as files on the Mac's disk. You may have up to eight separate logical disks for OS-9, each of which can be any size. One of them must be used as a bootable disk, containing a valid OS9Boot file on it.

As for device drivers, there exists support for using the Mac's devices with what appears to be some type of "pass-through" driver. I had no problems using the serial ports on my Mac. Also, included with the package is a driver to allow use of a Teac floppy drive for OS-9. (However, there is no support for using the Mac's SuperDrive in Universal format.) Also, OS-9 has a special device, /m0, which allows access to any files on a Mac format disk. (However, since file name conventions differ between the two os's, there is often problems getting to files this way.)

As advertised, this version of OS-9 should provide a MacToolBox library which allows programming of applications that use the Mac's built-in hardware (Quick Draw, etc.), however, the version I tested did not have this feature. After calling the company, I learned that they had this working, however, "any technical Mac user could unravel what we have done, removing the mystery", i.e., they don't want anyone reverse engineering their product. They also told me that they were planning a release that would be more secure and just as functional. I am curious as to how their modifications will affect the speed of ToolBox calls.

One abnormality I found, whenever you fork (from the shell) a new process, with redirection of all paths to /term, the process appears in a new window. This can be helpful, but they are both device /term! I'm sure this can cause great confusion under normal circumstances.

I discovered an annoyance with their terminal emulation, which runs under the TERM type of "Mac", they have no support for inverse characters!

Overall, I did enjoy using OS-9 on the Macintosh, and I foresee UltraScience improving their software in the not-too-distant future.

Mark Heilpern

OS-9 User Group Tee-Shirts

It appears that demand for our "Kick Butt In Real-Time" t-shirts have surged. To appeal the masses, we have ordered another run of the popular in-your-face shirts. Only 30 have been ordered, and some have been spoken for already. All shirts are made of 50% cotton, 50% polyester and feature the OS-9 Users Group Logo on the back.

Size	Member Price	Non-Member Price
L, XL	$13.00	$15.00
XXL	$14.50	$16.50

Send your Check or M.O. to The OS-9 Users Group. Please state quantity and size. Send to:
P.O. Box 71131, Des Moine, IA 50325

Using AWK, Complex Patterns and Regular Expressions

by Zack C. Sessions

Last time, we had an introduction to the AWK Programming Language, specifically, the GNU version of AWK for the OSK Operating System, gawk.

The GNU AWK is "fully" compliant with the formal definition of AWK as described in "The AWK Programming Language" by Aho, Kernighan and Weinberger. (See my comments on this in the previous article published last time. Z.)

We talked about gawk's command line syntax and options, what an awk program is and what it's basic structure is. We learned that each awk program statement has two parts, a pattern and an action. We also discussed a couple of different types of patterns. We talked about the special patterns BEGIN and END, about expressions as patterns, and simple regular expressions.

This time we'll first talk about some of the more complex pattern types and later we'll get into a more detailed discussion of regular expressions.

The first of the more complex patterns I will discuss are known as Compound Patterns. These are expressions which combine other expressions with logical ANDs, ORs, and NOTs. For example, you can have:

```
$1 == "Mary" && $3 > 100
```

Again, standard C operators are used, &&, || and ! for and, or and not. In this case, the pattern is true for any record where the first field contains exactly the string value "Mary" and the third field is greater than 100, when considered as a numeric variable. This reminds me of something I glossed over in part one of this series on AWK. Let me digress for a second. While a field which contains a "pure" numeric value is considered as a numeric field, it can be referenced in the context of a string variable. When used so, the value of the variable is converted to a string variable before the value of the variable is referenced. This is also true with alphanumeric variables, in fact. That is, a string variable refernenced in a numeric context, it's ASCII values are converted to an numeric value. The effect is analogous to an atoi() or an atof() function call in C. Let me finish this digression by commenting that it should be obvious that variables are referenced more efficiently in the context of which the are defined as.

Oh yes, let me finish describing the previous awk programming statement by saying that for all records which the complex pattern is true, the entire record will be displayed to the standard input. Actually, since no action is specified, the default action is processed for all records which match the patter. That action is:

```
{print $0}
```

So, the previous example is a shorthand form of the following awk programming statement:

```
$1 == "Mary" && $3 > 100 {print $0}
```

Range Patterns are two patterns separated by commas. The range pattern is true for all records starting with a record for which the first pattern is true and then continuing sequentially through the file up and including a record, if found, for which the second pattern is true. If the first pattern is never true, no records will be processed. If the first pattern is ever true, and the second pattern is then never true, all records starting with the one which matched the first pattern through to the end of the input are processed by the range pattern's associated action. Each of the two patterns may be of any of the different types of patterns. For example:

```
/Alfred/, /Karen/ {print $1,$2}
```

This is two regular expressions as the first and second pattern. In this case, the first record found which contains the substring "Alfred" and all records after it up to and including a record, if found, which contains the substring "Karen" are processed by the patterns' associated action.

Also, a valid range pattern is:

```
NR > 3, NR > 10
```

This shows that the two a parts of a range pattern can be expressions as well as regular expressions. (Remember the difference between an expression and a regular expression?) In this case the 4th through the 10th record are all processed by the patterns' action. I had to think about this one for a second. Remember, after the first pattern is true all records are processed UNTIL the second pattern is TRUE.

Here's something similar:

```
$3 > 100, $3 < 200
```

In this case, if a record is found which has its third field greater than 100 then that record and all subsequent records will be processed by the action, until a record is come upon which has the third field less than 200. That record will be processed, too, but none after it. This one deserves a second thought also, but I'll let you handle that. Another interesting use for a range pattern would be this:

```
$1 == "Mary", $NF == 0
```

In this case, the first record found to have the first field equal to the character string "Mary" and all records after that will be processed until a record is found whose LAST field contains a value of zero. That will be also processed, but none after. Note that using the NF variable to denote
the LAST field in a record, a file with variable numbers of fields in the records would be no problem.

A range pattern may not be part of another pattern.

Last time I talked about simple regular expressions. Let's get back into them. The type of pattern which contains a regular expression is called a "string matching pattern". A pattern can contain more than one regular expression. It will contain either a single regular expression, or a regular expression used in conjunction with an expression. To indicate a regular expression, it is surrounded by slashes. This string matching pattern must fit one of the following three general formats:

1) /regexp/ - Matches with the current input l ine contains a substring matched by the regular expression regexp.

2) expression ~ /regexp/ - Matches if the string value of expression contains a substring matched by the regular expression regexp.

3) expression !~ /regexp/ - Matches if the string value of the expression does not contain a substring matched by the regular expression regexp.

Any expression may be used in place of /regexp/ in the context of ~ and !~. Here's examples of more complex string matching patterns:

```
$4 ~ /Mary/
```

This is an example of type 2 above. In this case, field #4 of the input record must contain the substring "Mary" for the pattern to be true. Consider the following:

```
$1 ~ $3
```

In this case the /regexp/ is replaced by an expression, in this case, the field variable $3. This can only be done in the context of ~ and !~. The ! operator is use as a NOT modifier. Here is an example of its use:

```
$5 !~ /Phil/
```

This pattern would be true for all records which did NOT contain the substring "Phil". But, we have really only touched on what came come between the slashes for a regular expression. So far, all examples with regular expressions contained only a character string. There are many special characters called "metacharacters" which can be used to indicate special processing.

For example, the "^" character matches the beginning of a string and the "$"character matches the end of the string. These metacharacters may appear alone or in combination in a pattern. Consider the string matching pattern:

```
$1 ~ /^Chicago$/
```

In this case, the regular expression says to match with a string which starts with the C, and ends with the o, and has an hicag in between. So, only records whose first field is the string "Chicago" (not just contains the substring) will match and the pattern be true. The "*" character matches any size string of any characters, and the "?" character matches zero or one occurances of the previous character. Example:

```
$2 ~ /^Z*/
```

This pattern would be true for all records in which the second field BEGAN with the character "Z" (UPPERCASE Z), followed by zero or more of any character. Consider this example:

```
$4 ~ /ing$/
```

This pattern would be true for all records whose fourth field ENDS with the substring "ing". Here's another:

```
$5 ~ /A?/
```

This pattern would be true for any record whose fifth field contained either the value "A" or "AA". The last metacharacter I will discuss is the [] pair. The [] contains one or more individually considered characters in it. It can also specify a range. For the pattern to be true, the string must match only the characters listed within the []'s. For example:

```
$1 ~ /^[ABC]/
```

This pattern is true for all records whose first field starts with one of the characters, "A", "B", or "C", and is followed by zero or more of any characters. Note that the comparison is VERY case sensitive! Here's an example of a range:

```
$2 ~ /^[a-zA-Z]/
```

This pattern is true for all records which has a second field which has as its first character a letter, upper or lower case. It may have zero or more characters after the initial letter. Be careful when using combinations of metacharacters! Consider the following example:

```
$2 ~ /^[a-zA-Z]*/
```

Now, at first glance, you might think that this does the same thing as the previous pattern. Uh, uh, it doesn't!! In fact, it will match on field two no matter what field two contains! You see, the first metacharacter is the dual character range which matches only a single character. Let's say that the range does match the first character. Then, no matter what is next, the "*" metacharacter will match it. But, let's say that the range does not match. Then no matter what is next the "*" metacharacter will match it!

So, this pattern matches anything, which nullifies the reason to even attempt the first character verification. So, use the "*" metacharacter carefully!!

Multiple ranges may also be specified. For example:

```
$2 ~ /^[0-9][A-Z]$/
```

In this case, only records in which the second field starts with a numeric character and ends with an upper case alphabetic character.

This is as deep as I want to get with regular expressions. I'll end this time with a quickie awk program which may help to illustrate a technique.

You want to know how many total bytes are used by the files in the current directory. Consider the command:

```
$ dir -e ! gawk 'NR > 3 { tot += $6 } END
  { print "Total bytes", tot }'
```

While the actual use of this command is a waste of time if you have a copy of the ls command which can supply file size totals, it illustrates how you can interpet system function displays by awk programs. In this case, the first three lines are ignored, and all subsequent lines, the 6th field, the size in bytes, is summed to the variable tot. At the end of file, the total is displayed. An equivalent command line would be:

```
$ dir -eu ! gawk '{ tot += $6 } END
  { print "Total bytes", tot }'
```

This example also shows an arithmetic expression I haven't mentioned, ie, the use of the "+=" arithmetic operator. As you might expect, all C type arithmetic expressions are supported in awk programs, including the auto pre or post increment.

Next time, we'll concentrate more on the action part of AWK programming statements.

–

Zack Sessions
sessions@seq.uncwil.edu
University of North Carolina at Wilmington
"Good health is merely the slowest form of dying."

NOTHING TO BAWK AT...

(Part 2 of 2)

by Boisy G. Pitre

```
**************************************
* Strips leading spaces
*
* Entry: X - Address of line
*
* Exit:  X - Points to first non-space character

EatSpace        pshs    a
Eat2    lda     x+
        cmpa    #$20
        beq     Eat2
        leax    -1,x
        puls    a
        rts

.....................................
* Entry of program

Start   decb                    any params?
        beq     Help            nope, exit w/ error

        clr     Path            assume stdin upon entry
        clr     IncFlag         Clear (OFF) inclusion flag
        clr     FileFlag        Clear printing of  filenames
        clr     Anchor          Anchor to first column
        clr     FEFlag          Clear Fork/Echo flag
        clr     ForkFlag                Clear Fork flag
        lda     #$20            put space as extra delimiter
        sta     Delm

.....................................
* Command line parsing is done here

Parse   bsr     EatSpace
        lda     x+
        cmpa    #$0d
        beq     Help
        cmpa    #'-
        bne     IsItQ
* Dash options parsed here
        lda     ,x+             load A with char
        cmpa    #'a             is it the anchor option?
        bne     IsItF
        bsr     Str2Byte
        stb     Anchor
        bra     Parse
IsItF   cmpa    #'f
        bne     IsItUpF
        lda     #$ff
        sta     ForkFlag
        bra     Parse
IsItUpF cmpa    #'F
        bne     IsItL
        da      #$ff
        sta     FEFlag
        bra     Parse
IsItL   cmpa    #'l
```

```
          bne     IsItI
          lda     #$FF
          sta     FileFlag
          bra     Parse
IsItI     cmpa    #'i         is it the inclusion option?
          bne     IsItD
          lda     #$ff        set Inclusion Flag
          sta     IncFlag
          bra     Parse
IsItD     cmpa    #'d         delimiter?
          bne     Help        bad option -- error out
          lda     x+          else load character after
the 'D'
          sta     Delim       save it...
          bra     Parse       then go back to parsing the
line
* Format String detected here
IsItQ     cmpa    #'"         Is it a '"' format string?
          bne     Help        nope, must be an error

* Save the format string
SaveFmt leay      Format,u
SaveFmt2          lda     x+          Point to char after
first '"'
          cmpa    #$0d
          beq     Help
          cmpa    #'"         is it the second '"'?
          bne     SaveFmt3            no, save char
          lda     #$0d
          sta     ,y
          bra     ChkFile
SaveFmt3  sta     ,y+         else save char
          bra     SaveFmt2
ChkFile   lbsr    EatSpace Check after last '"' for a
filename
          lda     ,x
          cmpa    #$0d        if no filename, execute from
StdIn
          beq     MainLine
          bra     OpenFile

*****************************************
* Help Routine
*

Help      leax    HelpMess,pcr        Show Help message
          lda     #2
          os9     I$WritLn
          bra     Done

*****************************************
* Check for EOF
*

EOF       cmpb    #E$EOF
          bne     Error
          lda     path
          os9     I$Close     Close path
          puls    x           and restore the cmd line
pointer
          tst     Path
          beq     Done
          bra     FilePrs

*****************************************
* Exit Here
*

Done      clrb
Error     os9     F$Exit

*****************************************
* BAWK goes here if files are on the cmd line
*

FilePrs   lbsr    EatSpace        eat spaces
          lda     ,x          check char
          cmpa    #$0d        if CR,
          beq     Done

OpenFile          lbsr    SaveFile
          lda     #read.      else assume a file name
          os9     I$Open      and try to open it
          bcs     Error
          sta     Path
          tst     FileFlag
          beq     MainLine
          lbsr    PrnFile

*****************************************
* The following lines are the "heart" of BAWK's processing
*

MainLine  pshs    x           save pointer to cmd line

*****************************************
* The line of input is read from here.
*

ReadLine          lda     Path        get path
          ldy     #250        max chars per line
          leax    Line,u      point to line buffer
          os9     I$ReadLn            and read the line
          bcs     EOF         check EOF if error

*****************************************
* The Process of Expansion starts here.
*

ProcLine          leax    Format,u
          leay    ExpLine,u           Position Y to
expansion line

ParseFmt  lda     ,x+
          cmpa    #'$         Is it the '$' field character?
          beq     FieldPar    Check Field Parameter
PFmt2     sta     ,y+
          cmpa    #$0d
          bne     ParseFmt
          tst     ForkFlag
          bne     PFmt3
          bsr     Print
          tst     FEFlag      see if the fork/echo flag is
set
          beq     ReadLine
PFmt3     lbsr    Fork
          bra     ReadLine
FieldPar  lda     ,x+         get char after '$'
          cmpa    #'$         Is it another?
          beq     PFmt2       yep, store it
          leax    -1,x
FieldP2   lbsr    Str2Byte            convert the number
          tstb    check the number to see if it's 0
          bne     FieldI

*****************************************
* The entire line is copied at the direction of $0
*

          pshs    x
          leax    Line,u      at this point we copy the
entire...
          lbsr    AncLine     Anchor the line
```

```
CopyAll  lda     ,x+         and transfer the rest of the
line
         cmpa    #$0d        line since we've encountered a
$0
         beq     Field3      and continue parsing
         sta     ,y+
         bra     CopyAll

Field1   pshs    x           save position in format string
         bsr     SetField    Position to the proper field
         tstb                was there an error?
         beq     Field2      no, continue with expansion
         tst     IncFlag     is the inclusion flag set?
         bne     Field2
         puls    x
         bra     ReadLine
Field2   bsr     Expand
Field3   puls    x           get position in format string
         bra     ParseFmt    and continue expanding...


*****************************************
* SETFIELD - This routine positions the X pointer to the
correct field
*
* Entry:  B - Number of the field
*
* Exit:   B - clear if field was found, set if it wasn't found
*         X - Address of Bth field (Points to EOLN if B is
set)
*

SetField leax   Line,u
         lbsr   AncLine     Anchor the line
Skip     lda    ,x+
         cmpa   #$20
         beq    Skip
         cmpa   Delim
         beq    Skip
         cmpa   #$0d
         beq    Leave2
         decb
         beq    Leave
EatField lda    ,x+
         cmpa   #$20
         beq    Skip
         cmpa   Delim
         beq    Skip
         cmpa   #$0d
         beq    Leave2
         bra    EatField
Leave    clrb
Leave2   leax   1,x
         rts


*****************************************
* EXPAND - This routine "expands" the field into the
expansion buffer
*

Expand   lda     ,x+
         cmpa    #$20
         beq     ExExit
         cmpa    Delim
         beq     ExExit
         cmpa    #$0d
         beq     ExExit
         sta     ,y+
         bra     Expand
ExExit   rts


*****************************************
* The expanded line is printed to StdOut here
*

Print    leax    ExpLine,u          Point X to the
expanded line buffer
         ldy     #500       max chars 500
         lda     #1
         os9     I$WritLn   write to stdout
         lbcs    Error
         rts

*****************************************
* The expanded line is used as a paramter to a shell
*

Fork     pshs    x,u
         lda     #Prgrm+Objct
         ldb     #16        Use 16 pages (4K) of data
         leax    Shell,pcr          Point to name of
Shell
         ldy     #1096
         leau    ExpLine,u          Point X to the
expanded line buffer
         os9     F$Fork     Fork it!
         lbcs    Error
         os9     F$Wait
         puls    x,u
         rts

         emod
Size     equ     *
         end
```