OS-9 USERS GROUP MOTD

"DEDICATED TO EXCELLENCE IN OS-9 COMPUTING"

Officers:
President: Boisy G. Pitre
Vice President: Carl Kreider
Secretary/Tres: Debi Kreider
Librarian: Scott McGee
MOTD Editor: Alan Sheltra

Early Fall 1992 | The Official Voice of the OS-9 Users Group | Volume 1, Number 2

Bigger and Better...and More!

Editorial by Alan Sheltra

The MOTD now has a new look and a new schedule. It also has a new editor, Alan Sheltra. That's me. Scott McGee, the former editor, has taken over as the Group's Librarian as he is in a better position to handle the varied OS-9 formats than I was (including OS-9000).

Originally, the MOTD was to be a quarterly newsletter, but it was decided that with news breaking as fast as it does, it would be better to publish with more frequency than before. The MOTD will now be published 6 times a year (bi-monthly) starting with this issue. Better for you, and better to let us tell you what's happening in the World of OS-9.

I'm not new to publishing and have been involved with advertising and graphic design since 1975. Some of you may also recognize my name as the publisher of the "**OS9 Underground**".

This issue's submissions has us "Awk-ing" and "Bawk-ing", with a tutorial on the OSK "AWK" utility by Zack Sessions and an "AWK-like" utility in ASM by Boisy Pitre (Our President). Boisy also shares a C utility, "Park.c" which will compile under any flavor of OS-9, including OS-9000.

Just a reminder that you can submit your programs or articles for publication in the MOTD. Please send your submissions on a 3.5" or 5.25" format diskette (Coco or any OSK format) to:

Alan Sheltra - MOTD Editor
OS-9 Users Group
4650 Cahuenga Blvd. Ste. #7
Toluca Lake, Ca. 91602

You can also e-mail your submission direct to "zog!motd@abode.ttank.com", either case gets your submission direct to me.

Next issue should have a report (and pix) of the Atlanta Coco Fest. The OS-9 Users Group will be there, come and say "Hello" to the guys!

Until next time...!

-Alan Sheltra - MOTD Editor

An Introduction to AWK by Zack C. Sessions

In the February 1992 issue of "**68xxx Machines**" (Now the **OS9 Underground**), in his column Bob van der Poel talks about solving complex problems with a few standard commands along with one which is readily available. I was pleasently surprised to see that Bob was talking about GAWK. I recently was required to get more "intimate" with AWK for a college course, and had already put some work in this artical since I, too had gotten a copy of GAWK for my MM/1. So, after reading Bob's article, I thought I'd finish this article.

First, some history. The AWK programming language first came into being in 1977, being the brainchild of Alfred V. Aho, Peter Weinberger, and Brian W. Kernighan, all of Bell Labs. Knowing this, it is now obvious how the authors thought of AWK as the name of the language. [Hint: *Look at their last name's only.*] The authors saw the power, yet shortcomings of the UNIX utilities **grep**, **fgrep** and **sed**. They adopted a goal of developing a pattern-scanning language that would understand fields, one with patterns to match field and action to manipulate them.

The current version of AWK was released in 1985, and is documented in a book by Aho, Kernighan and Weinberger called "The AWK Programming Language", ISBN 0-201-0798-X. This version as implemented on most UNIX systems is called "nawk" for "new awk", and the original 1977 version of AWK remains just "awk".

The Free Software Foundation, as expected, released their version of AWK called GNU AWK, or GAWK in 1986. The most recent version, V2.11 was released only recently. It supposedly supports all the features and functions of AWK as descibed in the aforementioned book. (I have found a few situations where GAWK does not fully function as descibed in the book, but nothing major.) This version of GAWK is available for OSK from several sources. I can send you a copy for the cost of the disk and mailing charges. Write to me if interested. (P.O. Box 540, Castle Hayne, NC 28429)

Okay, do I have your undivided attention? I now will address the central and most impotant issue of our discussion on AWK. Just what the heck is it, what does it do and what good is it to me? Here on out, I will refer to AWK as GAWK, since all example commands are depicted how I have used them on my MM/1. But essentially, GAWK is AWK. I will be using the dollar sign ($) to indicate the SHELL prompt in the example commands.

Directory of MOTD_EARLY_FALL :

From the Desk of Boisy... (Editorial)

User Group Prsident Boisy G. Pitre

OS-9000: Power to the Masses

Most, if not all OS-9 users have heard of OS-9000. In short, it is an enhanced version of OS-9/68K with some major differences:

∞ OS-9000 is written in C, whereas OS-9/68K is written in 68000 assemble language

∞ Because OS-9000 is written in C, it is portable to a number of hardware platforms, including CISC and RISC processors.

∞ OS-9000 is available for 68010 and higher processors as well as 386/486 computers.

Prices of Intel-based computers have dropped rapidly in the past few years. With the introduction of the Intel 586, prices on the 386 and 486 are dropping even more. From a standpoint of cost, OS-9000 represents a tremendous value: a powerful operating system for real-time and development applications which runs on widely-available and relitively inexpensive hardware.

Microware's recent release of OS-9000 version 1.3 includes the Microware C Compiler and VPC (Virtual PC) software. VPC allows one to run DOS applications under OS-9000 while maintaining full multi-tasking capabilities. And, OS-9000come with an easy to use installation program which takes care of partitioning your hard drive device and installing the OS-9000 system.

Personally, I am surprised that OS-9 hardware vendors have not taken advantage of the tremendous market potentioal here. Imagine if you will, a complete 386 computer system bundled with OS-9000! An excellent sales oportunity exists. Not only could such a n investment be a financial success, it would give OS-9 users a base on which to grow and expand.

OS-9000 is the perfect example of a state-of-the-art operating system that can adapt to the constant change of he computer industry. The **OS-9 Users Group** is committed to the establishing support for OS-9000 and VPC. It's an important committment.

by Boisy G. Pitre

GAWK is a very intelligent file processor. It can process several files with a single invocation. A typical GAWK command would look like:

```
$ gawk 'awk program' [file(s)]
```

If no files are listed, GAWK read from the Standard Input Path, making it perfect for receiving it's data from from a pipe, as you will see later. If more than one file is listed, all files are processed, but only one at a time, from left to right.

If the awk program is a multi-line program, or if you plan to use it often, you would want to create the awk program in a separate file and invoke the gawk command:

```
$ gawk -f awk.program {file(s)]
```

Where awk.program is the filename of the awk program file.

The key part of either format is the "awk program". Thjat is where all of the intelligence is. An awk program is one or more of awk program statements. Each statement has two parts, apattern match, and an action. Both parts are optional, but one has to be there. So, the structuree of the awk program would be:

```
[pattern] [{action}]
```

The square brackets indicate each part is optional, and actually, if omitted there is a default value for each of them. The action portion of an awk program statement is distinguished by being surrounded by curly braces. Each awk program is applied to each record in the ASCII files. Here is a simple awk program:

```
{print}
```

This awk program displays each record read to the Standard Output path. Since awk awk sends it's output to the standard output path, it is perfect for piping to a child process. Here there was no pattern, thus the default pattern "all records" was used.

GAWK does some "pre-processing" of the records read from the input stream before passing the data to the awk program, in fact this is one of gawk's most impressive and useful features. Each record is parsed and the "fields" are identified. Each field in the input is a string of characters which does not contain a blank. So the following line of data:

```
Kathy 4.00 10
```

has three fields. The values of these three fields can be used by referencing special variables awk sets up for you. The format for a field variable is $n where 'n' is the field number. In the above record, $1 would contain the character string "Kathy", $2 would contain the number 4.00, and $3 would contain the number 10. Yes, awk determines if the field is alphanumeric or numeric. There is another special variable name you can use to reference the input line. Variable $0 refers to the entire line. In this record, variable $0 would contain the character string, "Kathy 4.00 10". The default action for an awk program is equivalent to {Kathy 4.00 10"}.

The "n" portion of the field variable can be variable itself. For example, the following awk program will print each field of each record in a separate line.

```
{for (i = 1; i < NF; ++i)
print $i
}
```

Note again that the field's variable names start with 1. $1 is the first field, $0 is not. $0 is the ENTIRE current input record, I'll talk about what NF is in just a second. You might also notice the similarity of awk action statements to C programming statements.

There is also a few other special built-in variables. NR contains the number of the current input record. Since an END pattern's action is executed after all input data has been processed, in an end pattern's action the built-in variable NR contains the number of records read from the input.

NF is the number of fields in the current record. So, the variable $NF would represent the LAST field in the current input record. There are several other built-in variables, all of which are beyond the scope of this article. The awk programmer can also use user defined variables simply by referencing them, as I used the variable "i" in the action example above. The data type, string or numeric, is determined by the context. Arrarys of numbers and strings are even supported.

All user defined variables also are automatically given an initial value the very first time they are referenced in an awk program statement. The initial value for numeric variables is 0, and for string variables "". Datatype is interpeted from the context it is used in.

The scope of variables is also important to realize, but is also beyond the scope of this article. (No pun intended!) For now, consider field variables as "global", known to all patterns and actions, and variables referenced in an action are local to that action.

The pattern portion is a little more difficult to fully understand especially the regular expressions, unless you are familiar with the concepts of regular expressions a la UNIX. The pattern may be any one of the following:

1) The string BEGIN.
2) The string END.
3) An expression.
4) A regular expression.
5) A compound pattern.
6) A pattern.

I'm not going into the last two at all, and the fourth one only breifly. I will devote an entire artical on the last three types of patterns.

BEGIN and END are special patterns. The indicate that their asscoiated actions (it is of little use to have a BEGIN and END with no action!), are only performed at special times. The BEGIN action is always processed BEORE ANY data has been read. The END action is ALWAYS processed AFTER ALL data has been processed.

A pattern which is in the format of an expression is a comparison between two expressions. All standard comparison operators used in the C programming language are recognized, !=, ==,>=, >, plus two others, > means "is matched by" and !+ means "is not matched by". I'll talk about these last two later. A typical expression pattern would be:

```
$2 >= $3
```

The pattern would be true if the value of field #2 is less than or equal to the value of field #3. For each record in the input stream that the pattern is true, the pattern's associated action is executed. Since the default action is { print }, if the above pattern were the entire awk program, then for each record which the pattern were true, the entire record would be written to standard output. Since expressions are allowed in the two items to compare, the following is a valid expression pattern:

```
$1 / 2 >= $4 * $5
```

In this case, field 1 is divided by a constant 2. That value is compared to the product of the values of fields 4 and 5, and the pattern is true if it is less than or equal.

A pattern can also be what is called a "string matching pattern". In most cases, this is usually in the form is a single regular expression. To signify that a pattern is a regular expression, it must be enclosed in slasheds. Here's an example:

```
/Mary/
```

In this case, the pattern is true for any record which contains the substring "Mary", and thus the pattern's associated action would be performed.

Now, on to some actions! An action is one or more valid awk action statements. These look much like C programming statements, and indeed, some are identical and function the same. Perhaps the most common action of the awk action clauses is to output something with either a print or a printf function. "printf" works exactly like the C function does. The print statement command:

```
print("%s %s %s/n",$1,$2,$3)
```

The fields are automatically separated by a space. This example assumes that the three fields are all strings. Awk is smart enough to know the difference. Note that the print also does an implied new line at the end of it's data. Awk program actions can also "if" (with optional "else"), "while", "for", "do while", "break", "continue", "next" and "exit" statements.

I'll go deeper in subsequent articles. Now at risk of making Bob mad at me, I'm going to analyze his awk programming skills using his February, 1992 article as a guide. His first awk program is a simple one:

```
$1 < /bsr/ { print $2 }
```

Well, nothing I can say about that. Short, sweet and fuctional. But looking ahead I see that this awk program is intended to be run on several different files and since it's short, it is being run supplied on the command line itself. In a UNIX environment, that is fine, but on my MM/1 using GAWK, there is something to consider. Each time you run a command like:

```
$ gawk '$0 < /Asia/ { print $3, $4 * $5 }' countries
```

there is a file created in the /dd/TMP directory which contains the awk program. Each time it is run, a SEPARATE file is created, even if the awk program being executed over and over and over is exactly the same program. So, every so often, you need to clear out the /DD/TMP directory. So, in Bob's procedure, for this reason, I would have extracted the awk program for both the gawk commands to go out into an external file and use the -f option.

Next program, I got a few observations. Bob uses a BEGIN pattern merely to initialize a variable to 0. Since all variables are given an initial value the very first time they are ever referenced, and if they are used in the context of numerical expression, they assume the value of 0. So, the BEGIN pattern and it's action are redundant and not needed. I can't reall improve on the rest of the second awk program.

I will finish up with an awk success story which strengthens Bob's premise in his article, that is to use the tools you have. I have just downloaded several files and I wanted to set the attributes of the files to public read/write. Thhese files were the only files in the directory which were created on that date, but there were other file in that same directory I didn't want to mess with. So consider the following command:

```
$ dir -e ! gawk '$2 == "92/03/11" { print $7 }' ! attr -z -pr -pw
```

Actually, that is not how I first did it. My first attempt was even cruder:

```
$ dir -e ! grep 92/03/11 ! gawk '{ print $7 }' ! attr -z -pr -pw
```

Actually, if grep is already in memory, the second version would probably run faster, um, nope, I was thinking we would save a lot by having grep do the pattern matching, but gawk will still parse the entire record before even processing the pattern, in this case, process all records. But, the overhead of creating the fourth process needs to be considered also. It was pretty to watch in a procs display in another window!

Next time, more on the complex pattern types and regular expressions.

Zack C. Seesions

Park Those Drives!

by Scott McGee & Boisy G. Pitre

Park, a utility by Scott McGee and Boisy G. Pitre for parking not only your hard drives, but your Disk and Tape drives as well.

The C listing below should be able to compile on the 6809 C Compiler or those used on os9-68K and OS-9000.

Happy Parking!

```
/*
 * Park - Resets an RBF device's read/write heads .
 *
 *  This utility uses the _ss_rest() call to restore an RBF device's
 *  read/write head to track 0.  Park works on most RBF devices,
 *  including floppy disk drives.  For SBF devices, park rewinds the
 *   tape.
 *
 *    For 6809 C compilers, the Kreider library must be linked to
 *    reslove the _ss_rest() function.
 *
 *    Written by:  Scott McGee & Boisy G. Pitre
 *    (C) 1992 - The OS-9 Users Group
 */

extern int errno;

#include <stdio.h>
#include <errno.h>
#include <modes.h>
#ifdef _OS9000
#include <sg_codes.h>
#else
#include <sgstat.h>
#endif

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

main(argc, argv)
int argc;
char *argv[];
{
    int quiet = FALSE;

    if (argc == 1) ShowHelp();
    while (argv[1]) {
         if (argv[1][0] == '-')
              switch (toupper(argv[1][1])) {
                   case 'Q' : quiet = TRUE;
                              break;
                   default  : ShowHelp();
              }
         else ParkDev(argv[1], quiet);
    ++argv;
    }
}
```