A Supplemental Document For AWK

- or -

Things Al, Pete, And Brian Didn't Mention Much

John W. Pierce

Department of Chemistry
University of California, San Diego
La Jolla, California 92093
jwp%chem@sdcsvax.ucsd.edu

ABSTRACT

As **awk** and its documentation are distributed with *4.2 BSD UNIX** there are a number of bugs, undocumented features, and features that are touched on so briefly in the documentation that the casual user may not realize their full significance. While this document applies primarily to the *4.2 BSD* version of *UNIX*, it is known that the *4.3 BSD* version does not have all of the bugs fixed, and that it does not have updated documentation. The situation with respect to the versions of **awk** disitributed with other versions *UNIX* and similar systems is unknown to the author.

May 28, 1986

*UNIX is a trademark of AT&T

A Supplemental Document For AWK

- or -

Things Al, Pete, And Brian Didn't Mention Much

John W. Pierce

Department of Chemistry
University of California, San Diego
La Jolla, California 92093
jwp%chem@sdcsvax.ucsd.edu

In this document references to "the user manual" mean *Awk - A Pattern Scanning and Processing Language (Second Edition)* by Aho, Kernighan, and Weinberger. References to "awk(1)" mean the entry for **awk** in the *UNIX Programmer's Manual, 4th Berkeley Distribution.* References to "the documentation" mean both of those.

In most examples, the outermost set of braces ('{ }') have been ommitted. They would, of course, be necessary in real scripts.

1. Known Bugs

There are three main bugs known to me. They involve:

Assignment to input fields.

Piping output to a program from within an **awk** script.

Using '*' in *printf* field width and precision specifications.

1.1. Assignment to Input Fields

[This problem is partially fixed in *4.3BSD*; see the last paragraph of this section regarding the unfixed portion.]

The user manual states that input fields may be objects of assignment statements. Given the input line

```
field_one field_two field_three
```

the script

```
$2 = "new_field_2"
print $0
```

should print

```
field_one new_field_2 field_three
```

This does not work; it will print

```
field_one field_two field_three
```

That is, the script will behave as if the assignment to $2 had not been made. However, explicitly referencing an "assigned to" field *does* recognize that the assignment has been made. If the script

```
$2 = "new_field_2"
print $1, $2, $3
```

*UNIX is a trademark of AT&T

is given the same input it will [properly] print

```
field_one new_field_2 field_three
```

Therefore, you can get around this bug with, e.g.,

```
$2 = "new_field_2"
output = $1                  # Concatenate output fields
for(i = 2; i <= NF; ++i)     # into a single output line
        output = output OFS $i    # with OFS between fields
print output
```

In *4.3BSD*, this bug has been fixed to the extent that the failing example above works correctly. However, a script like

```
$2 = "new_field_2"
var = $0
print var
```

still gives incorrect output. This problem can be bypassed by using

```
var = sprintf("%s", $0)
```

instead of "*var* = $0"; *var* will have the correct value.

1.2. Piping Output to a Program

[This problem appears to have been fixed in *4.3BSD*, but that has not been exhaustively tested.]

The user manual states that *print* and *printf* statements may write to a program using, e.g.,

```
print | "command"
```

This would pipe the output into *command*, and it does work. However, you should be aware that this causes **awk** to spawn a child process (*command*), and that it *does not* wait for the child to exit before it exits itself. In the case of a "slow" command like **sort, awk** may exit before *command* has finished.

This can cause problems in, for example, a shell script that depends on everything done by **awk** being finished before the next shell command is executed. Consider the shell script

```
awk -f awk_script input_file
mv sorted_output somewhere_else
```

and the **awk** script

```
print output_line | "sort -o sorted_output"
```

If *input_file* is large **awk** will exit long before **sort** is finished. That means that the **mv** command will be executed before **sort** is finished, and the result is unlikely to be what you wanted. Other than fixing the source, there is no way to avoid this problem except to handle such pipes outside of the awk script, e.g.

```
awk -f awk_file input_file | sort -o sorted_output
mv sorted_output somewhere_else
```

which is not wholly satisfactory.

See *Sketchily Documented Features* below for other considerations in redirecting output from within an **awk** script.

1.3. Printf Field Width and Precision Specification With '*'

The document says that the *printf* function provided is identical to the *printf* provided by the *C* language **stdio** package. This is not true for the case of using '*' to specify a field width or precision. The command

```
printf("%*.s", len, string)
```

will cause a core dump. Given **awk**'s age, it is likely that its *printf* was written well before the use of '*' for specifying field width and precision appeared in the **stdio** library's *printf.* Another possibility is that it wasn't implemented because it isn't really needed to achieve the same effect.

To accomplish this effect, you can utilize the fact that **awk** concatenates variables before it does any other processing on them. For example, assume a script has two variables *wid* and *prec* which control the width and precision used for printing another variable *val:*

```
[code to set "wid", "prec", and "val"]

printf("%" wid "." prec "d\n", val)
```

If, for example, *wid* is 8 and *prec* is 3, then /fBawk will concatenate everything to the left of the comma in the *printf* statement, and the statement will really be

```
printf(%8.3d\n, val)
```

These could, of course, been assigned to some variable *fmt* before being used:

```
fmt = "%" wid "." prec "d"

printf(fmt "\n", val)
```

Note, however, that the newline ("\n") in the second form *cannot* be included in the assignment to *fmt*.

2. Undocumented Features

There are several undocumented features:

Variable values may be established on the command line.

A **getline** function exists that reads the next input line and starts processing it immediately.

Regular expressions accept octal representations of characters.

A **-d** flag argument produces debugging output if **awk** was compiled with "DEBUG" defined.

Scripts may be "compiled" and run later (providing the installer did what is necessary to make this work).

2.1. Defining Variables On The Command Line

To pass variable values into a script at run time, you may use

variable=value

(as many as you like) between any "**-f** *scriptname*" or *program* and the names of any files to be processed. For example,

```
awk -f awkscript today=\"‘date‘\" infile
```

would establish for *awkscript* a variable named **today** that had as its value the output of the **date** command.

There are a number of caveats:

Such assignments may appear only between **-f** *awkscript* (or *program* or [see below] **-R***awk.out*) and the name of any input file (or '-').

Each *variable=value* combination must be a single argument (i.e. there must not be spaces around the '=' sign); *value* may be either a numeric value or a string. If it is a string, it must be enclosed in double quotes at the time **awk** reads the argument. That means that the double quotes enclosing *value* on the command line must be protected from the shell as in the example above or it will remove them.

Variable is not available for use within the script until after the first record has been read and parsed, but it is available as soon as that has occurred so that it may be used before any other processing begins. It does not exist at the time the **BEGIN** block is executed, and if there was no input it will not exist in the **END** block (if any).

2.2. Getline Function

Getline immediately reads the next input line (which is parsed into *$1*, *$2*, etc) and starts processing it at the location of the call (as opposed to **next** which immediately reads the next input line but starts processing from the start of the script).

Getline facilitates performing some types of tasks such as processing files with multiline records and merging information from several files. To use the latter as an example, consider a case where two files, whose lines do not share a common format, must be processed together. Shell and **awk** scripts to do this might look something like

In the shell script

```
( echo DATA1; cat datafile1; echo ENDdata1 \
  echo DATA2; cat datafile2; echo ENDdata2 \
) | \
   awk -f awkscript - > awk_output_file
```

In the **awk** script

```
/^DATA1/  {       # Next input line starts datafile1
        while (getline && $1 !~ /^ENDdata1$/)
              {
              [processing for data1 lines]
              }
        }

/^DATA2/  {       # Next input line starts datafile2
        while (getline && $1 !~ /^ENDdata2$/)
              {
              [processing for data2 lines]
              }
        }
```

There are, of course, other ways of accomplishing this particular task (primarily using **sed** to preprocess the information), but they are generally more difficult to write and more subject to logic errors. Many cases arising in practice are significantly more difficult, if not impossible, to handle without **getline**.

2.3. Regular Expressions

The sequence "*ddd*" (where 'd' is a digit) may be used to include explicit octal values in regular expressions. This is often useful if "nonprinting" characters have been used as "markers" in a file. It has not been tested for ASCII values outside the range 01 through 0127.

2.4. Debugging output

[This is unlikely to be of interest to the casual user.]

If **awk** was compiled with "DEBUG" defined, then giving it a **-d** flag argument will cause it to produce debugging output when it is run. This is sometimes useful in finding obscure problems in scripts, though it is primarily intended for tracking down problems with **awk** itself.

2.5. Script "Compilation"

[It is likely that this does not work at most sites. If it does not, the following will probably not be of interest to the casual user.]

The command

```
awk -S -f script.awk
```

produces a file named **awk.out.** This is a core image of **awk** after parsing the file *script.awk.* The command

```
awk -Rawk.out datafile
```

causes **awk.out** to be applied to *datafile* (or the standard input if no input file is given). This avoids having to reparse large scripts each time they are used. Unfortunately, the way this is implemented requires some special action on the part of the person installing **awk**.

As **awk** is delivered with *4.2 BSD* (and *4.3 BSD*), *awk.out* is created by the **awk -S ...** process by calling **sbrk()** with '0', writing out the returned value, then writing out the core image from location 0 to the returned address. The **awk -R...** process reads the first word of *awk.out* to get the length of the image, calls **brk()** with that length, and then reads the image into itself starting at location 0. For this to work, **awk** must have been loaded with its text segment writeable. Unfortunately, the *BSD* default for **ld** is to load with the text read-only and shareable. Thus, the installer must remember to take special action (e.g. "cc -N ..." [equivalently "ld -N ..."] for *4BSD*) if these flags are to work.

[Personally, I don't think it is a very good idea to give **awk** the opportunity to write on its text segment; I changed it so that only the data segment is overwritten.]

Also, due to what appears to be a lapse in logic, the first non-flag argument following **-R***awk.out* is discarded. [Disliking that behavior, the I changed it so that the **-R** flag is treated like the **-f** flag: no flag arguments may follow it.]

3. Sketchily Documented Features

3.1. Exit

The user manual says that using the **exit** function causes the script to behave as if end-of-input has been reached. Not menitoned explicitly is the fact that this will cause the **END** block to be executed if it exists. Also, two things are ommitted:

exit(*expr*) causes the script's exit status to be set to the value of *expr*.

If **exit** is called within the **END** block, the script exits immediately.

3.2. Mathematical Functions

The following builtin functions exist and are mentioned in *awk(1)* but not in the user manual.

int(*x*) *x* trunctated to an integer.

sqrt(*x*) the square root of *x* for *x* >= 0, otherwise zero.

exp(*x*) **e**-to-the-*x* for -88 <= *x* <= 88, zero for *x* < -88, and dumps core for *x* > 88.

log(*x*) the natural log of *x*.

3.3. OFMT Variable

The variable **OFMT** may be set to, e.g. "%.2f", and purely numerical output will be bound by that restriction in **print** statements. The default value is "%.6g". Again, this is mentioned in *awk(1)* but not in the user manual.

3.4. Array Elements

The user manual states that "Array elements ... spring into existence by being mentioned." This is literally true; *any* reference to an array element causes it to exist. ("I was thought about, therefore I am.") Take, for example,

```
if(array[$1] == "blah")
      {
      [process blah lines]
      }
```

If there is not an existing element of **array** whose subscript is the same as the contents of the current line's first field, *one is created* and its value (null, of course) is then compared with "blah". This can be a bit disconcerting, particularly when later processing is using

```
for (i in array)
     {
     [do something with result of processing
       "blah" lines]
     }
```

to walk the array and expects all the elements to be non-null. Succinct practical examples are difficult to construct, but when this happens in a 500 line script it can be difficult to determine what has gone wrong.

3.5. FS and Input Fields

By default any number of spaces or tabs can separate fields (i.e. there are no null input fields) and trailing spaces and tabs are ignored. However, if **FS** is explicitly set to any character other than a space (e.g., a tab: **FS = "\t"**), then a field is defined by each such character and trailing field separator characters are not ignored. For example, if '>' represents a tab then

```
one>>three>>five>
```

defines six fields, with fields two, four, and six being empty.

If **FS** is explicitly set to a space (**FS** = " "), then the default behavior obtains (this may be a bug); that is, both spaces and tabs are taken as field separators, there can be no null input fields, and trailing spaces and tabs are ignored.

3.6. RS and Input Records

If **RS** is explicitly set to the null string (**RS** = ""), then the input record separator becomes a blank line, and the newlines at the end of input lines is a field separator. This facilitates handling multiline records.

3.7. "Fall Through"

This is mentioned in the user manual, but it is important enough that it is worth pointing out here, also.

In the script

```
/pattern_1/ {
        [do something]
        }

/pattern_2/ {
        [do something]
        }
```

all input lines will be compared with both *pattern_1* and *pattern_2* unless the **next** function is used before the closing '}' in the *pattern_1* portion.

3.8. Output Redirection

Once a file (or pipe) is opened by **awk** it is not closed until **awk** exits. This can occassionally cause problems. For example, it means that a script that sorts its input lines into output files named by the contents of their first fields (similar to an example in the user manual)

```
{ print $0 > $1 }
```

is going to fail if the number of different first fields exceeds about 10. This problem *cannot* be avoided by using something like

```
{
command = "cat >> " $1
print $0 | command
}
```

as the value of the variable **command** is different for each different value of *$1* and is therefore treated as a different output "file".

[I have not been able to create a truly satisfactory fix for this that doesn't involve having **awk** treat output redirection to pipes differently from output to files; I would greatly appreciate hearing of one.]

3.9. Field and Variable Types, Values, and Comparisons

The following is a synopsis of notes included with **awk**'s source code.

3.9.1. Types

Variables and fields can be strings or numbers or both.

3.9.1.1. Variable Types

When a variable is set by the assignment

var = expr

its type is set to the type of *expr* (this includes +=, ++, etc). An arithmetic expression is of type *number,* a concatenation is of type *string,* etc. If the assignment is a simple copy, e.g.

var1 = var2

then the type of *var1* becomes that of *var2*.

Type is determined by context; rarely, but always very inconveniently, this context-determined type is incorrect. As mentioned in *awk(1)* the type of an expression can be coerced to that desired. E.g.

{
expr1 + 0

expr2 "" # Concatenate with a null string
}

coerces *expr1* to numeric type and *expr2* to string type.

3.9.1.2. Field Types

As with variables, the type of a field is determined by context when possible, e.g.

$1++ clearly implies that *$1* is to be numeric, and

$1 = $1 "," $2 implies that $1 and $2 are both to be strings.

Coercion is done as needed. In contexts where types cannot be reliably determined, e.g.,

if($1 == $2) ...

the type of each field is determined on input by inspection. All fields are strings; in addition, each field that contains only a number is also considered numeric. Thus, the test

if($1 == $2) ...

will succeed on the inputs

0 0.0
100 1e2
+100 100
1e-3 1e-3

and fail on the inputs

(null) 0
(null) 0.0
2E-518 6E-427

"only a number" in this case means matching the regular expression

^[+-]?[0-9]*\.?[0-9]+(e[+-]?[0-9]+)?$

3.9.2. Values

Uninitialized variables have the numeric value 0 and the string value "". Therefore, if *x* is uninitialized,

if(x) ...
if (x == "0") ...

are false, and

if(!x) ...
if(x == 0) ...
if(x == "") ...

are true.

Fields which are explicitly null have the string value "", and are not numeric. Non-existent fields (i.e., fields past **NF**) are also treated this way.

3.9.3. Types of Comparisons

If both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to type string if necessary, and the comparison is made on strings.

3.9.4. Array Elements

Array elements created by **split** are treated in the same way as fields.