awk

User's Manual

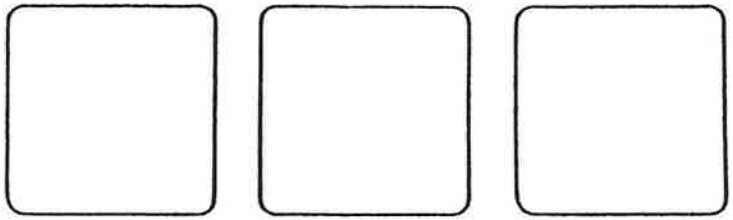

Table of Contents

1. Introduction

awk is a general-purpose pattern scanning language within the COHERENT operating system. **awk** performs pattern matching, string manipulation, record processing, and report generation.

The syntax for **awk** is simple. It uses only one kind of statement, consisting of one or both of two elements: a *pattern* and an *action*. *Patterns* select the data to be processed, and *actions* specify the function to be performed on the selected data.

This manual explains how to write programs for processing input with **awk**. Including this introduction, there are five sections in this manual.

Section 2 explains how to use the **awk** interpreter and how to create an **awk** program. In addition, the section discusses the principles of data selection and the structure of command line arguments.

Section 3 describes the basic function of printing and the specification of input and output field and record separators.

Section 4 explains the pattern scanning capabilities of **awk**.

Section 5 describes the actions **awk** performs in addition to printing, such as assigning variables, defining arrays, and controlling the flow of data.

For further information, about related COHERENT commands, see the *COHERENT Command Manual*. If you are not familiar with the operation of COHERENT, see the *Introduction to the COHERENT System*.

2. Using awk

Like many other COHERENT utilities, awk is a filter. awk reads input from the standard input (entered from your terminal or from a file you specify), processes each input line according to a specified awk program, and writes output to the standard output. This section explains the structure of an awk program and the syntax of awk command lines.

Program Structure

The basic element of an awk program is a statement in the form:

```
pattern {action}
```

A program may contain as many sets of *patterns* and *actions* as you need to accomplish your purposes.

awk checks each line of input with the *patterns* specified for a match, one pattern at a time. Each time the line matches a *pattern*, awk performs the corresponding *action*. After awk has compared the line with each *pattern* in the program, awk tests the next input line against the *patterns*.

An awk program may specify an *action* without a *pattern*. When awk processes an *action* which has no *pattern*, each input line matches. Therefore, awk performs the *action* on every line of the input.

An awk program may also specify a *pattern* without an *action*. In this case, when an input line matches the *pattern*, awk copies (or prints) the line to the standard output.

One of the special *patterns* that awk recognizes is the word FILENAME. This *pattern* causes awk to print the name of the file that it is currently using as a source for data. Other special *patterns* are discussed below.

Records and Fields

awk divides its input into separate records, and subdivides each record into separate fields. Records are separated by a character called the input record separator (RS), and fields are separated by the input field separator (FS).

The default input record separator is the newline character, so **awk** normally regards each input line as a separate record. Because the default input field separator is either the space or the tab character, white space normally separates fields.

In addition to input record and field separators, **awk** provides output record and field separators (**ORS** and **OFS**) which it prints between output records and fields. The default output record separator is the newline character; **awk** normally prints each output record as a separate line. The default output field separator is the space character.

To process input with a record separator other than the newline character, use the special **BEGIN** pattern (fully described in Section 4) with an *action* assigning the desired record separator to the variable **RS**. For example,

```
BEGIN {RS = ":"}
```

changes the record separator to a colon. You may specify any one character as the record separator. Specifying the null string (**RS** = **""**) makes two consecutive newlines the record separator. If you include more than one character within double quotes, **awk** ignores all characters after the first one.

To change the output record separator, assign the desired character to the variable **ORS**. The output record separator may be a single character or a string. For example, the following program assigns the string ***record end*** to **ORS**:

```
BEGIN {ORS = "***record end***"}
```

The variable **NR** gives you the number of the current record. In the following program, **awk** prints this number at the beginning of each record to make editing easier:

```
{print NR, $0}
```

Here is a program that prints the total number of records in the input file.

```
END {print NR}
```


awk can also use the record number in relational expressions. To select a particular record for printing (for example, line 6), use the following program:

```
NR == 6 {print $0}
```

which tells **awk** to print the whole record when the number of the record is equivalent to 6.

Each record is subdivided into fields. Within the record, you may refer to each field separately by the name **$***n*, where *n* is the field number. For example, the fourth field is called **$4**. The entire current record is called **$0**.

Like records, fields have a default separator. For fields, the default separator is white space for both input and output fields (usually spaces or tabs; newlines can separate fields when **RS** is null).

You may change the field separator (variable **FS**) in two ways. The first way is to specify the change within the **awk** program, as follows:

```
BEGIN  {FS = ":"}
```

The sample statement changes the field separator to a colon. When you specify several characters within quotes, each character becomes a field separator, and all separators have equal precedence. For example, you can specify commas, colons, and periods to separate fields. In the following program, **awk** looks for any of these separators, and breaks the record into fields at each occurrence of each character:

```
BEGIN  {FS = ",:."}
```

The second method of changing the field separator is to use a command line argument. The command line method enables you to declare the field separator at the time you invoke **awk**. For more information, see "Command Line Arguments" below. To show how changing the input field separator affects the ouput, consider the following record from the file "now":

```
Now is the time for all good men
```

and the **awk** statement:

```
{print $1,$2}
```

When the input field separator is the default, the result of the awk program is:

```
Now 1s
```

When using the same statement but setting FS = "i", awk prints the following:

```
Now s the t
```

As the input field separator, "i" is not printed; however, in its place a blank separates the two output fields. The first field consists of uppercase "N", lowercase "o" and "w", and a space. The second field consists of the "s", a space, the word "the", and the "t" of time.

When you use an input field separator other than the default, the printed output can look confusing, as in the example above. However, you can change the output field separator by assigning a character or string to the variable OFS.

To indicate where fields are divided when the output is printed, you can assign a character such as * to OFS as follows:

```
BEGIN {OFS = "*"}
      {FS = "i" ; print, $1, $2}
```

This program prints the following:

```
Now *s the t
```

Notice that a semicolon (;) separates two statements on the same line. For more information about the use of the semicolon, see "Printing with awk" in Section 3.

The variable NF contains the number of fields in the current record. In the following program, awk prints the number of fields at the beginning of each output record, telling you the number of elements in the record:

```
{print  NF,$0}
```

awk can also use the variable NF in relational expressions. For example, to print all records with 10 or more fields, you could use this program:

```
NF >= 10 {print $0}
```


Command Line Arguments

As with any COHERENT program or command, you invoke awk by typing the lowercase letters awk. To process files with awk, you must include some additional elements on the command line, called arguments.

The complete form for the awk command line is:

awk [**−y**] [**−F***c*] [**−f** *progfile*] [*prog*] [*file1*] [*file2*] ...

Each of the command line arguments is explained below.

The **−y** option enables you to name *patterns* in lowercase characters, which awk matches to both uppercase and lowercase characters in the input file. This option is similar to its counterpart in the COHERENT regular expression pattern-matching utility, **grep**.

The following programs show how the **−y** option works on the file named **the**, which contains the following two lines:

```
The time is right.
Now is the time.
```

Command	Output
`awk -y '/the/' the`	`The time is right.` `Now is the time.`
`awk '/the/' the`	`Now is the time.`

The **−F***c* option is the command line version of **FS = "c"**, an assignment like the one described earlier in "Records and Fields". This option changes the input field separator from the default (white space) to the character *c*. You may include any characters you want awk to use as field separators after the **−F** flag.

The **−f** *progfile* option enables you to use a file *progfile* containing awk commands as an awk program. The option flag (**−f**) must precede the name of the file to be used as a program.

If you do not use the **−f** *progfile* option, you must use the *prog* option. This option specifies the awk program on the command line. When writing a command-line awk program, use a single quotation mark before the first statement (*pattern*, *action*, or both);

then enter the subsequent lines of the program. After the last statement of the program, type another single quotation mark followed by the file or files to be processed. Note that COHERENT prompts you to enter more information by displaying the '>' at the beginning of each line until you enter the closing single quote and newline character.

The following program is an awk command-line program. It prints a heading before awk reads the input file "test", and then prints the entire file with each line preceded by its line number.

```
$ awk 'BEGIN {print "sample output file"}
>           {print NR, $0}' test
```

The *file1 file2* ... option enables you to process existing files. When you want to process more than one file, separate the file names with white space. If you do not specify a file name in the command line, awk takes input from the standard input.

The following program prints the files **test1** and **test2**. Each line is preceded by its record number.

```
$ awk '{print NR, $0}' test1 test2
```


3. Printing with awk

Printing is an awk *action*. In fact, it is the action most often used, because it is the simplest to use. The following short awk program prints its entire input:

```
{print}
```

When you specify awk actions, you may include several actions within one set of braces; however, each action must be separated from the others by semicolons (;) or newlines.

Printing Individual Fields

Using awk, you can print output fields in a different order from the input fields.

You can print fields in any order you desire. For example, you can print the second and third fields in reverse order:

```
{print $3,$2}
```

When this program processes the input file now containing the sample record used in Section 2, the printed result is:

```
the ls
```

Because the field names are separated by a comma, awk inserts an output field separator between the fields when printing them.

If you do not separate field names by commas in the print statement, awk concatenates the fields when printing them. For example, the following program prints the second and third fields:

```
{print $2 $3}
```

The result is:

```
isthe
```

Changing the Output Field and Record Separators

You may change the output field separator by assigning your desired separator to the variable **OFS**. To use the same field separator for the entire input, make the assignment before the first print statement. For example, to make the colon your output field separator, use a statement like this:

```
{OFS=":"; print $2,$3,$4}
```

Which means that you will receive this output:

```
1s:the:time
```

To change the separator for the first line only, use the statement:

```
NR ==1  OFS=":";print $2,$3,$4}
```

To change the output record separator from the default newline, assign required separator to the variable **ORS** in the same manner.

Printing Predefined Variables

As discussed in Section 2, you can print either or both of the NF (number of fields) or NR (number of records) predefined variables. To print a predefined variable, simply name it in the print statement. For example, to include the NF variable before the other output in the previous example, edit the program to read as follows:

```
{OFS = ":"; print NF,$2,$3,$4}
```

The output resulting from this statement is:

```
8:1s:the:time
```

You can specify the **NR** variable in the same way. When you add the name of the variable to the desired place in the list of fields to be printed, **awk** prints the record number in that place in the output.

Redirecting Output

In addition to printing to the standard output, you also may redirect output to a file or files of your choosing. This ability to direct output to any file enables you to extract information from a given file and construct new documents.

Suppose you have a file named **accounts** with accounting information stored in it. The first column of the file contains payroll information, the second column shows income for the year, and the third column reports accounts payable information. You are to make an income report for the year containing text and tables.

To extract the income information from the accounts file and put it into a separate file named **income**, you can use the following awk program:

```
{print $2 > "income"}
```

With this program, **awk** creates the file **income** if it does not already exist, and enters the second column of the accounts file as the contents of the new file. If a file named **income** already exists, **awk** replaces the current contents of the file with the second column of the accounts file.

If you need the first two columns for two separate reports, you can redirect both columns to separate files using one statement.

```
{print $2 > "income"; print $1 > "payroll"}
```

You can specify a maximum of 10 files for output.

If text for your report is already contained in the file **report**, you can append the second column of the accounts file to the end of your report using this awk program:

```
{print $2 >> "report"}
```

Appending enables you to complete your report without retyping a column of numbers that exists in another file.

Formatting Output

When you use **awk** to process a column of text or numbers as in the example above, you may want to specify a consistent format for the output. The statement for formatting a column of numbers follows this *pattern*:

```
{printf "format", expression}
```

where *format* is prescribed by the format control characters and separators defined below. *expression* specifies the fields for **awk** to print.

The following table shows the names and meanings of the most frequently used **awk** format control characters. To be recognized as format control characters by awk, these characters must be preceded by the percent sign '%' and a number in the form of *n* or *n.m*.

Format Control Characters	Meaning
%nd	Decimal number
%n.mf	Floating point number
%n.ms	String of characters or digits

When you call the **printf** function through **awk** to format the output, you must specify the output separators you want to use.

Output Separator Character	Meaning
\n	Newline
\t	Tab
\f	Form feed
\r	Carriage return
\"	Quotation mark

For example, if you wish to print a column of numbers with up to 9 places to the left of the decimal and 2 to the right (for a total of 12 places, including the decimal), and you want a new entry for each line, use a format like this:

```
{printf "%12.2s\n", $2}
```

Piping Output

You can pipe the output of your **awk** program to another process. The pipe connects the standard output of **awk** to the standard input of another process, program, or utility.

For example, you can pipe output to the **mail** utility with the following program, which mails the output to **name**:

```
{print | "mail name"}
```

The pipe operator is the vertical bar character between the print and mail commands in this statement.

4. awk pattern scanning

The previous section described printing in terms of fields. Fields are generally the best way to select single elements from columnar input files. In addition to names of fields, the **awk** interpreter is capable of scanning records for the following:

- Two special *patterns*: **BEGIN** and **END**
- Regular expressions
- Arithmetic relational expressions
- Boolean combinations of expressions
- Pattern ranges

Special Patterns: BEGIN and END

BEGIN is a special *pattern* which matches the beinning of the input, before **awk** processes any of the input. As mentioned in Section 2, **BEGIN** is the best place to set the field and record separators if you want the same separators for the entire input. **BEGIN** is also a good place to perform the *action* of assigning values to variables when the values are known.

Actions that require **awk** to compare input with the variable NR may not produce the results you expect from a **BEGIN** *pattern*, because all **BEGIN** processing is finished before NR = 1. Also, **awk** does not permit field references in **BEGIN** or **END** statements.

END is a special *pattern* which matches the end of **awk** input. The **END** *pattern* enables you to request an *action* to occur when all processing is finished. A common use of **END** is printing the value of variables. For example:

```
END {print NR}
```

tells **awk** to print the value of NR after processing is finished, giving the total number of records processed. When you reach the **END** *pattern*, you may not return for further processing.

You may make **awk** into a calculator by using **END** with no *action*. At the end of the input, you may enter any arithmetic equation or **awk** function and have the result automatically printed on the standard output. When you are finished using **awk** as a calculator, type <ctrl-D>.

Patterns

You can enclose strings of characters in slashes '/' for awk to match, as ed (the COHERENT text editor) and grep (the COHERENT text pattern matching command) do. For example, take this pattern:

```
/ted/
```

When a statement contains this expression, awk prints every record with the string ted, whether ted occurs as a word or as part of a word. For example:

```
Interested
busted
tedious
```

In addition to specific strings, you can scan for classes and types of characters. To do so, enclose the characters within brackets, and place the bracketed characters between the slashes. For example, to specify a range of lowercase letters, enclose the range of letters within brackets:

```
/[a-z]/
```

You can specify ranges of uppercase letters or numerals the same way.

In addition, you can use the following special characters for further flexibility:

Character	Meaning
[]	Class of characters
()	Grouping subexpressions
|	Alternatives among expressions
+	One or more occurrences of the expression
?	Zero or one occurrences of the expression
*	Zero, one, or more occurrences of the expression
.	Any non-newline character

When adding one of the special characters to a pattern, enclose the special character as well as the rest of the pattern within slashes.

To search for a string that contains one of the special characters, you must precede the character with a backslash. For example, if you are looking for the string "today?", use the following *pattern*:

```
/today\?/
```

When you need to find an expression in a particular field, not just anywhere in the record, you can use one of these operators:

Character	Meaning
~	Contains
!~	Does not contain

For example, if you need to find the characters jam in the fourth field of the input, you can use the following statement:

```
$4~/[JJ]am/
```

This statement prints all lines where the fourth field contains Jam or jam. The statement also prints lines where the fourth field contains words like James, jammed, and pajamas. To prevent the awk program from selecting lines with characters other than separators on either side of the required expression, use the following special characters:

Character	Meaning
^	Beginning of the record or field
$	End of the record or field

With these characters, you can be still more specific about which field or record you want printed. For example, to allow James to be printed, but not pajamas, use the following statement:

```
$4~/^[JJ]am/
```

To allow only Jam or jam, use this statement:

```
$4~/^[JJ]am$/
```

Arithmetic Relational Expressions

An awk *pattern* may consist of relational expressions using the following operators:

Operator	Meaning
<	Less than
<=	Less than or equal to
=	Equivalent
!=	Not equivalent
>=	Greater than or equal to
>	Greater than

With these operators, you may select fields according to their relation to one another. For example, if you want to print the first field only when it does not equal the second field, use this statement:

```
$1 != $2 {print $1}
```

You also can establish relationships among records. If you want to print no more than the first ten records, use the following statement:

```
NR <= 10
```

Because this example specifies no action, the statement prints all the records whose record number is 10 or less.

Relational tests default to string comparison if either operand is nonnumeric. Thus, if one operand is numeric and the other is a string, awk makes a string comparison. The following example shows how awk compares one field to part of the alphabet:

```
$1 <= "C"
```

This statement selects all lines beginning with an ASCII value less than or equal to that of "C" (octal 103).

When you compare fields that have numeric values to one another, awk performs a numeric comparison. Consider the comparison in this example:

```
$2 < $1 + 100 {print $2}
```

This statement causes field 2 to be printed only when the value of field 2 does not exceed the value of field 1 by 100. If field 2 is alphabetic, it always matches in this comparison because strings evaluate to 0 in numeric comparisons.

Boolean Combinations of Expressions

awk tests logical combinations of expresssions in its pattern-scanning process. Use the following operators for combining expressions.

Operator	Meaning
||	Or
&&	And
!	Not

The following example tests for records that begin field 1 with a character that is less than u, greater than or equal to t, and begin field 1 with a string other than the.

```
$1 < "u" && $1 >= "t" && $1 != "the"
```

The effect of this *pattern* is to select records that have a t as the first character in field 1 but do not begin field 1 with the letters the.

Pattern Ranges

awk may cause an *action* to be performed on all records between two specified *patterns*. For example, to print all records between the *patterns* April 10 and April 19 inclusive, enclose the strings in slashes and separate them with a comma; then indicate the print action, as follows:

```
/April 10/,/April 19/ {print}
```

You also may specify a range of record numbers using a statement such as this:

```
NR == 5, NR == 17 {print}
```

This statement specifies that records 5 through 17 of the input are to be printed.

5. Specifying awk actions

This section describes awk *actions* other than printing *actions*. In addition to printing, awk is capable of:

- Performing functions
- Assigning variables
- Using fields as variables
- Concatenating strings
- Defining arrays
- Using control statements

Functions

awk includes functions that enable you to perform specific calculations with input information. You may assign these functions to any variable and use them in patterns. The following list shows the functions and their definitions; an argument can be any expression.

Function	Meaning
length	Returns the length of the current record
length(*argument*)	Returns the length of *argument*
sqrt(*argument*)	Returns the square root of *argument*
exp(*argument*)	Returns e to the power of *argument*
log(*argument*)	Returns the natural logarithm of *argument*
int(*argument*)	Returns the integer part of *argument*
abs(*argument*)	Returns the absolute value of *argument*
substr(*str,beg,len*)	Returns the substring of str that is *len* characters long beginning at position *beg*
index(*s1,s2*)	Returns the position of *s2* within *s1*, or 0 if *s2* does not occur in *s1*
sprintf(*f,e1,e2*)	Returns strings *e1* and *e2* in the printf format *f*
split(*str,array,fs*)	Divides *str* into fields associated with *array* (an array is a collection of fields listed under a single name) that are separated by *fs* or the default field separator

The **length**, **sqrt**, **exp**, **log**, **int**, **abs**, and **index** functions are self-explanatory.

When **substr** (*str,beg,len*) occurs in a statement, **awk** scans the argument string *str* for the position *beg* within the string. When **awk** finds *beg*, it prints a substring *len* characters long starting at *beg*. If *len* is not included in the argument, the substring includes everything from *beg* to the end of the record.

The **sprintf** (*f,e1,e2*) function enables you to format expressions *e1* and *e2* according to format specification *f*. The following example demonstrates the operation of the **sprintf** (*f,e1,e2*) function.

```
$ awk 'x = sprintf("%7.2s",$1)
> {print $1}
> END {print x}'
```

When you run this sample program, **awk** accepts input data from the keyboard of the terminal. The first line of the program begins the **awk** program and sets variable **x** so that it contains five blank spaces and the first two characters of the first input field. The second line causes **awk** to print the first field as it was received. The third line ends the program by printing **x**, the formatted version of the first input field.

If you enter the word **chicago** as the first input field for this program, **awk** prints:

```
chicago
     ch
```

The **split** (*str,array,fs*) function divides fields into subfields, breaking *str* into elements of *array* separated by *fs*, or white space when *fs* is not specified. In the following example, **awk** splits the first field of the record into subfields. If the record has a single colon in the first field, **awk** splits the field into two subfields. These subfields become the first and second fields of the array named **time** (see "Arrays" later in this section).

```
{split ($1,time,":")}
```

At this point, you may manipulate the information stored in the array **time** or simply print the subfields.

Assignment of Variables

In addition to the intrinsic variables, such as **NR** which contains the number of the current input record, and **FILENAME** which contains the name of the current file, you may assign other variables as described below.

Variables in **awk** may be string or numeric variables, depending on the context. By default, variables are set to the null string (numeric value 0) on start-up of the **awk** program. To set the variable **x** to the numeric value 1, you can use the following assignment statement:

```
x = 1
```

To set **x** to the string **ted**, use the following statement:

```
x = "ted"
```

When the context demands it, **awk** converts strings to numbers or numbers to strings. For example, the statement:

```
x = "3"
```

assigns to **x** the string 3. When an expression contains an arithmetic operator such as the '−', **awk** interprets the expression as numeric. (Alphabetic strings evaluate to 0.) Therefore,

```
x = "3" - "1"
```

assigns the value 2 to variable **x**.

When the operator is included within the quotes, **awk** treats the operator as a character in the string. In the following example,

```
x = "3 - 1"
```

assigns the string

```
"3 - 1"
```

to **x**.

You also can perform numeric calculations on fields. For example, you can calculate the sum of the fourth field in the following manner:

```
{sum += $4}
END {print sum}
```

The following table includes all the available operators for awk.

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo division
++	Increment
--	Decrement
+=	Add and assign value
-=	Subtract and assign value
*=	Multiply and assign value
/=	Divide and assign value
%=	Divide modulo and assign value

You may use any of these operators in awk expressions.

Field Variables

In awk, fields may receive assignments, be used in arithmetic, and be manipulated in string operations. The following awk statements show some of the available uses of fields as variables.

Statement	Meaning
{$1 = NF; print}	The first field is assigned the number of fields in the record; the resulting record is printed.
{$1 = $3 - $2; print $0}	The value of field 2 is subtracted from the value of field 3 and assigned to field 1; the resulting record is printed.
{if (length ($2) > 11) $2 = "large field" print }	If the length of field 2 is greater than the numeric value 11, the statement assigns the string "large field" to the field, and then prints the record.
{print $1, $(1+1), $(1+n)}	Using numeric expressions to refer to fields, this statement prints fields 1, 1+1, and 1+n.

String Concatenation

As mentioned in Section 3, you may concatenate strings by omitting comma separators in printing actions. The following example shows a print statement that concatenates the first two fields by inserting a new connecting string:

```
{print $1 " telephones "  $2}
```

If $1 contains "Tom" and $2 contains "John", this statement prints:

```
Tom telephones John
```

Arrays

In awk programs, arrays are collections of values labelled with the name of the array. Each element has at least one named index. The array is implicitly declared because awk creates the array when you name it. Also, you can name the individual indices with any legal string or numeric value.

Because the indices for any array may have any value, the ordering of array elements is arbitrary. However, when you use numeric index names exclusively, awk follows an ascending numeric sequence.

You should specify the array element using an identifier followed by the array index, an arbitrary expression enclosed in brackets ([]). For example, consider an array called **surname**. This example uses array indices named **tom**, **van**, and **gordon**. The following action assigns a value to each of these indices:

```
BEGIN  {surname ["tom"] = "Jones"
surname ["van"] = "Johnson"
surname ["gordon"] = "smith"}
```

You can print the contents of the array by naming the array in a **print** statement. awk also enables you to print the name of the index by associating another variable with the index, using a special form of the **for** statement. This form of **for** is:

```
for (index in array)
```

To retrieve the index names of the array **surname**, you may use the following statement:

```
END  {for (person in surname)
print person, surname[person]}
```

This statement yields the following output:

```
tom  Jones
van  Johnson
gordon  smith
```

In addition to being a generic term for the indices in the array **surname**, awk creates an array of names called **person**, to which you can make further associations as needed.

To store the number of occurrences of a pattern , you may use the associative array capabilities of awk. For example, if you want to determine the number of occurrences of **mark** and **test**, and print the number next to its respective word, you can use the following program:

```
/[Mm]ark/  {n["mark"]++}
/[Tt]est/  {n["test"]++}
END        {for (word in n)
           print word, n[word]}
```

With each occurrence of **Mark** or **mark**, awk increments the variable **n["mark"]**. (awk automatically initializes **n["mark"]** and **n["test"]** to 0 at the start of execution.) After awk processes the last line of the input, the program prints each word and the number of occurrences of that word as stored in **n[word]**.

Control Statements

awk has seven defined control statements. The following section explains the statements and gives examples of their use.

if (condition) else

If the *condition* within the parentheses is true, the statement following the **if** is executed. If there is a clear alternative, the **else** precedes the action to be performed when the condition is false. The **else** is optional. If awk does not perform the *action* of the **if** statement and there is no **else** statement, awk continues with the next statement. Example:

```
{
if (NR % 2 == 1)
        print "odd-numbered record"
else
        print "even-numbered record"
}
```

Index

User Reaction Report

To keep this manual and COHERENT free of bugs and facilitate future improvements, we would appreciate receiving your reactions. Please fill in the appropriate sections below and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name: _____

Company: _____

Address: _____

_____

Phone: _____ Date: _____

Version and hardware used: _____

Did you find any errors in the manual? _____

_____

_____

Can you suggest any improvements to the manual? _____

_____

_____

Did you find any bugs in the software? _____

_____

_____

Can you suggest improvements or enhancements to the software?

_____

_____

_____

Additional comments: (Please use other side.)