

Chapter 15: Using the TAWK Debugger.

The TAWK Debugger is used to help you find problems in your TAWK programs.

Starting the Debugger.

The debugger itself is written in TAWK in a file called "debug.awk". To use the debugger you compile this file with your TAWK program. The debug.awk file should appear before any of your own program source files, to ensure that the debugger starts before your program does. The debug.awk file is installed in the same directory as the TAWK compiler. You do not need to specify a pathname for debug.awk because TAWK looks in this directory for program source files that are not found in the current directory.

Suppose your program is in a file named "myprog.awk". To debug your program using the "awk" program, type this:

```
awk -f debug.awk -f myprog.awk
```

To debug using the TAWK Compiler you would type the following. Since debug.awk is first on the TAWK Compiler command line the output executable name defaults to "debug" (UNIX) or "debug.exe" (DOS, OS/2, Win32.) Use the -o option if you want to select a different executable filename. To run the debugger you simply type "debug", or whatever name you specified with the -o option.

```
awkc debug.awk myprog.awk  
debug
```

Basic Debugger Concepts

The TAWK Debugger has two primary operational modes that correspond to the state of your program: your program can be either running or stopped. When your program is stopped the debugger has control and displays your program source code and allows you to select debugger commands from the debugger's menu-bar.

The debugger uses two virtual screens corresponding to the two primary modes. One of the screens shows the normal input and output of your program and is displayed while your program is running. The other screen shows the main debugger window and is displayed when your program is stopped. Only one of the screens can be shown on your display at a time. The debugger switches the display back and forth between these two screens while you are debugging your program.

When your program is stopped, the main debugger screen is displayed. This screen has a menu-bar at the top, a main panel displaying the source code of your program, and a blank line at the bottom that is used for user input. The debugger highlights the program statement that will be executed next.

In order to see your program output when your program is stopped, you can switch from the debugger screen to your program output screen using the "Show_output" option from the debugger menu-bar. Just press the "s" key when the debugger screen is displayed. This option switches temporarily to your program output screen until you press any key to return to the debugger screen.

If the debugger screen does not come up when you start the debugger, make sure that you put "debug.awk" first on the command line. If that's not the problem, try pressing the Enter key. This may be necessary because the debugger stops your program just before each statement in the program is executed. If your program does not include any BEGIN or INIT blocks but does have an Automatic Input Loop, then TAWK will read the first line of program input before executing any statements in your program, so the debugger screen will not appear until after the first line of program input is read in.

You communicate with the TAWK Debugger using the keyboard to select items from the menu-bar. You can't use the mouse. The capitalized letter in each item indicates the key to press to select that item. You do not need to press the Alt or Shift keys, just the letter key. Most of the items in the menu-bar bring up a sub-menu. To get rid of the sub-menu without selecting anything press the Esc key.

Use the arrow keys and/or PageUp and PageDown keys to scroll through any window. For example, if your program source file does not fit on one screen, you can use these keys to move up or down to look at the rest of the program source file, or left or right to see program lines that are longer than 80 characters.

The bottom of the debugger screen shows function key definitions. These commands can also be selected using the menu-bar, but it is faster to use the function keys. For example, the “Trace” command can be selected from the “Run” menu or can be selected by pressing function key F1.

Tracing Your Program

There are several ways to watch your program execute one statement at a time:

- Trace** This command executes exactly one statement from your program. It will trace into functions. If you are stopped on the pattern part of a pattern-action block, and the pattern matches, this command will cause the debugger to trace into the actions in the block.
- Step** This command executes the next statement from your program, but skips over function calls. If the statement to be executed calls a function, the debugger will execute the entire function in a single step without showing you each statement executed (unless a “Stop Point” is encountered.) If you are stopped on the pattern part of a pattern-action block, this command will go on to the next pattern, rather than tracing into the actions in the block.
- Forward** This command is like “Step”, but it will not step backwards in your program, only forwards. In other words, it will continue execution until it encounters a statement that is forward (or below) the current program location. This is useful when you are debugging a loop and you want to get out of the loop. The Forward command can be used to continue execution until the loop is finished and the statement after the loop is about to be executed.
- Animate** This command “animates” your program by executing one statement, waiting a second, then executing the next statement, waiting another second, etc. It continues until you press any key to stop it. You can change the delay time between statements in the “Options” menu.
- Go** When you select “Go” your program starts running. It will continue to run until a “Stop Point” is encountered or until the program ends.

The **Trace** and **Step** commands have a special meaning when debugging a pattern-action statement. For example, consider the following program:

```
/pattern/ {
    statements
}
```

The debugger will stop with the line containing `/pattern/` high-lighted. To debug the *statements*, choose the **Trace** option. To continue to the next pattern-action block, choose the **Step** option.

Stop Points

Sometimes you want to debug only a small part of your program and do not want to trace through every statement. To do this start the debugger and place a “Stop Point” at or near the place you want to debug, then select “Go” to let your program execute until the Stop Point is encountered. To place a Stop Point select the “View Stop Points” option from the “Run” menu. This brings up a window showing all the currently set Stop Points. You can set as many Stop Points as you want. There are two kinds of Stop Points. If a Stop Point is set on a function, then the debugger will stop before any statement in the function is executed. The debugger will continue to stop before each and every statement in the function until you delete the Stop Point. You can also place a Stop Point on any particular statement in your program source code, and TAWK will stop before executing that statement. Be careful because TAWK does not check that the line number is valid when you enter it. If you enter a Stop Point on a line that does not have any statements on it then the Stop Point will have no effect.

Viewing Variables

There are several ways to view or modify the contents of variables in your program.

- 1) You can view any specified variable using the “Variables” menu “View” option. This allows you to enter the name of any variable, including local variables in the current function, and to view or modify the contents of the variable.
- 2) You can view a list of all global variables, both global and module local variables, or only variables defined in the current source file (module) by selecting the appropriate option from the “Variables” menu. Notice that these lists of variables do not include local variables defined in the current function.

- 3) You can watch variables by selecting the "Variables" menu "View" option. Watched variables are displayed in a separate window that is displayed whenever your program is stopped. You can watch any type of variable, including function local variables. If you specify a function local variable name the TAWK debugger will display the variable's value whenever you are in that function, and will display "<not available>" when the variable is not defined in the currently executing function.
- 4) You can see a list of the function local variables in any particular function by using the "Calls" menu. This brings up a list of the currently active functions. Select the function whose variables you want to view by using the arrow keys to highlight the function and the Enter key to select.

Module local variables are given unique names in the TAWK Debugger by prepending the file name in which they are defined, separated by a colon. For example, local variable x defined in file: myfile will be displayed in the list of all variables as "myfile:x".

TAWK's display of variables has some useful features:

Integers are displayed in both decimal and hexadecimal format. If a string is one character long, then the ASCII integer code is displayed along with the string. If a string is too long to fit on the screen, use the left and right arrow keys to shift the display left or right to see the rest of the string. The Ctrl-left and Ctrl-right arrow keys move faster. If the variable holds an array, then "<array>" is displayed. To see the contents of the array, move the highlighted bar to that variable using the arrow keys, and then choose "Expand_array" (press the "e" key.) This will show you the contents of the array. If a variable is uninitialized, the value "<uninitialized>" is displayed. If a variable is a special type, for example, a file descriptor, the special type is included after the value, for example, "<fileid>". See the **typeof** function for a list of the special types that may be shown.

Leaving the Debugger

When your program ends the debugger quits. If you want to do some more debugging, you must restart the debugger again.

You can press the Control-C key to quit the debugger if you lose control of your program. You lose control of your program if you use the "Go" option and it never hits a Stop Point, in which case your program just runs to completion.

You can also leave the debugger using the "Quit" option from the debugger menu.

Chapter 16: Calling External Functions

TAWK programs can call functions written in other languages. These functions are called "external" functions, because they are external to the TAWK program. These functions are declared using an **extern** declaration. Once they are declared, external functions can be used just like any other functions in TAWK. Note that your TAWK program can call functions written in another language, but not vice versa.

TAWK uses two different mechanisms for linking to external functions. Which mechanism you will use depends on the operating system you are using:

1) Dynamic Linking:

This is used for all versions except DOS. The external functions are located in a Dynamic Link Library, or DLL, which is located and linked in when your TAWK program runs. All you have to do is provide the appropriate "extern" function declarations in your TAWK program. You do not need to own a C or C++ compiler, unless you want to write your own DLL functions. Only 32-bit DLLs are supported. Older Windows 3.1 DLLs will not work.

In general, you can call any functions defined anywhere in the operating system. This gives you access to vast resources. Under Win32, you can invoke almost any function in the Win32 API (Application Programming Interface), including dialog box creation. For examples, look in the examples subdirectory, or look at some of the TAWK libraries, such as socket support, that are implemented by calling DLLs. Unfortunately, the documentation for these resources is also vast, and well beyond the scope of this book. Additional information is available direct from operating system vendors. As of this writing, the Win32 API is available on CD-ROM as the "Microsoft Developer Network". Basic UNIX documentation is available in many bookstores: ask for a "UNIX Programmer's Reference Manual".

2) Static Linking:

This is used for the 16-bit DOS version of TAWK. To perform static linking, you must own a C compiler that is supported by TAWK for static linking. The TAWK Compiler uses the linker supplied with your C compiler to statically link the C object or library files that you provide with object code and libraries that are provided with the TAWK Compiler.

Notes: The OS/2 version of TAWK includes both 16-bit and 32-bit runtime programs. The 32-bit version supports dynamic linking. The OS/2 16-bit runtime does not support external functions. The 32-bit DOS version of TAWK also does not support external functions.

Declaring External Functions in TAWK

To call an external function from TAWK, you must include an **extern** declaration for the external function in your TAWK program. An **extern** declaration tells the TAWK compiler the name of the external function, and the number and types of the arguments required for the function. Extern declarations may appear in any TAWK source file, and become globally defined names for the entire TAWK program. Extern declarations can also be placed in the TAWK library, to be used if needed. Each global name in TAWK can have only one meaning, so predefined TAWK function names can not be declared extern, nor can an extern name also be used as a global TAWK function or global variable. The same extern declaration may be duplicated in multiple files, as long as all such declarations are identical.

Extern declarations appear outside of any function or program block. They may appear each on a separate line, or may be separated or terminated by semi-colons. The syntax for an extern declaration is:

```
extern [ret_type] fun_name([type [name][, ...]])
```

The **extern** keyword is required.

The *ret_type* specifies the type of the return value from the function. If no *ret_type* is specified, **int** is assumed.

The *types* and *names* within parentheses are optional, and specify the types and names of the function parameters. The *name* is ignored, except that it may not be a TAWK reserved word. The *ret_type* and function parameter *types* may be any of the following types: (Brackets indicate optional stuff, and | indicates multiple possibilities.)

External Function Parameter and Return Types

```

[ signed | unsigned ] char [*][*]
[ signed | unsigned ] short [*][*]
[ signed | unsigned ] int [*][*]
[ signed | unsigned ] long [*][*]
float [*][*]
double [*][*]
struct anything [*][*]
string *
void [*][*]

```

The meanings of the base *types* are as follows:

<u>Type</u>	<u>Meaning</u>
char	8 bit character.
short	16 bit integer.
int	Either 16 bit or 32 bit integer depending on the operating system.
long	32 bit integer.
float	Single precision floating point number. Note that some C compilers automatically use double even when a function is declared as requiring a float. If this is the case, you must use double, not float.
double	Double precision floating point number.
string	Special TAWK data type allowing you to pass binary data, possibly containing 0 characters, between TAWK and the external function. The string format changed between TAWK version 4.1 and version 5.0, to support strings longer than 64K bytes. The documentation for the string data type can be found in the file "errata.txt" included with the TAWK compiler.
void	If this appears as a <i>ret_type</i> , it means the function has no return value. As a special case, the entire parameter list may be specified as "void", for example: <pre>extern foo(void)</pre> means that function foo has no arguments, and is like: <pre>extern foo()</pre>

The signed and unsigned keywords govern whether the corresponding value is signed or unsigned. The default is signed. When an external function parameter is a char or short, the value must be widened to the size of an int before the function is called. If the parameter is signed, the value is sign-extended, meaning the original sign of the value is preserved. If the parameter is unsigned, the value is zero-extended, meaning that additional zero bits are added on the left as necessary to widen the value to the size of an int. The signed/unsigned characteristic affects the range of values representable by each data type. The ranges are:

<u>Parameter Type</u>	<u>Range of Values</u>
signed char	-128 to 127
unsigned char	0 to 255
signed short	-32768 to 32767
unsigned short	0 to 65535
signed long	-2147483648 to 2147483647
unsigned long	0 to 4294967295

The optional trailing * indicates a pointer. The interpretation of what a pointer means depends on the type of thing pointed to.

<u>Type</u>	<u>Meaning</u>
<code>char*</code>	Pointer to an array of characters terminated by a 0 character. If an external function returns a <code>char*</code> , then it must return either a NULL (0) value, or a valid character pointer. If the value is non-NULL, TAWK copies the character data immediately after the function returns, so the data MUST be 0 terminated, so that TAWK can find where it ends.
<code>short *</code> <code>int *</code> <code>long *</code> <code>float *</code> <code>double *</code>	These may appear only as function parameters, not as a <i>ret_type</i> . The corresponding function parameter is assumed to be pass-by-reference. TAWK will pass a pointer to an appropriately sized argument. The external function can modify this value before it returns. If the argument to the function call was a simple variable (not a constant, array, field, or built-in variable) then the changes will be applied to that TAWK variable. If the argument to the function call was not a simple variable, TAWK passes a pointer to a temporary location containing the value specified in the function call, but changes made by the external function are ignored. NOTE: An alternate interpretation of this syntax is "pointer to array of type". For example, <code>short*</code> could mean a pointer to an array of short values. If this is the correct interpretation, you must use <code>void*</code> instead, and create the array to be passed to the external function using the TAWK pack function.
<code>void *</code> <code>struct name *</code> <code>type **</code>	These are used to signify pointers of any other type. The <i>type</i> may be any of the other types, above. All three syntaxes mean the same thing to TAWK: a generic pointer. TAWK does not have an automatic way of dealing with these types of pointers, so it is the user's responsibility to pass a pointer of the appropriate type. The pack function is typically used to create the appropriate structure.

Automatic Conversions

Values in TAWK can be numbers or strings (or other types, like arrays.) However, the parameters to external functions must have a specific type, as specified in the **extern** function declaration. Therefore, TAWK must handle the problem where the value passed to an external function is not of an appropriate type. The following table specifies how TAWK handles this.

Declared parameter type	If the actual parameter value is a string	If the actual parameter value is a number
<code>char</code>	passes the first character of the string.	passes the number, truncated to 8 bits.

int, short, long	passes the value of the string interpreted as a number, for example "10" simply passes 10.	passes the number, truncated if necessary.
char *	passes a pointer to the string.	If the value is 0, passes NULL. Otherwise, an error message is printed.
void *	passes a pointer to the string.	passes the number, which is assumed to be a pointer to the appropriate type. Your system will probably crash if it is not.

In all cases above, if the parameter is an array, an error message is printed. In all cases above, if the parameter is an uninitialized value, a 0 value is passed.

If you are using dynamic linking, the *ret_type* may additionally contain linkage keywords chosen from the following:

<u>Type</u>	<u>Meaning</u>
cdecl	Use C calling sequence. (the default) This calling convention pushes parameters in right-to-left order, and the calling function cleans up the stack. The function names are decorated in order to distinguish different types of calling conventions. The cdecl convention prepends an underbar ("_") to the name.
stdcall	Use Win32 "Standard Call" calling sequence. This calling convention pushes parameters in left-to-right order, and the called function cleans up the stack. This convention decorates the function name by prepending an underbar and appending an at-sign ("@") followed by the number of bytes of arguments required by the function.
winapi	Use for Win32 API functions. The winapi calling convention is similar to stdcall, but does not use stdcall function name decoration. Additionally, if the function can not be found in the DLL at runtime, TAWK appends an "A" to the function name and looks for that function in the DLL. The "A" is tried because many functions in the Win32 API have two versions to support two different character sets: The "ANSI" character set and the "UNICODE" character set. The ANSI versions of the functions all have an "A" appended to their name.
system	Use OS/2 "_System" calling sequence. Most OS/2 API functions use this convention.

<code>dll "dllname"</code>	Specifies the DLL filename. The <code>dll</code> keyword is followed by a string enclosed in double quotes that specifies the DLL filename. If the <code>dll</code> keyword is not provided, TAWK searches for the function in all the DLL filenames specified in the <code>DLLS</code> variable, which is initialized to the list of the most common DLLs used on each operating system.
<code>name "xxx"</code>	Specifies an alternate function name. This allows you to map the function name as known in the TAWK program to a different function name in the DLL. For example: <pre>extern name "xyz" foo()</pre> When the TAWK program uses function <code>foo()</code> , the actual external function called will be <code>xyz()</code> . This can be useful when an external function name conflicts with a TAWK built-in function name. The name specified in quotes is used as-is, without any checks for naming conflicts or validity. As a special case, in OS/2, the name can be of the form <code>"#nnn"</code> , where <code>nnn</code> is the decimal ordinal number of the function in the library.

Here are examples of dynamically linked extern functions. If the Dynamic Link Library "mydll.dll" contains function "foo", it could be declared as any of:

```
extern cdecl dll "mydll" foo()
extern cdecl dll "mydll" name "foo" alias()
```

In the above example, the TAWK function "alias" will actually call the C function "foo" in DLL: "mydll.dll".

Examples of TAWK statically linked extern declarations, to be used with DOS. A *ret_type* is not specified, because `cdecl` is assumed:

```
extern int max(int,int)
extern double log10(double);
extern int rename( char * ,char*)
```

Incorrect Examples:

```
extern print(char*) # "print" is a TAWK function
extern if() # "if" is a TAWK keyword.
```

The following will not work as you might expect. TAWK attempts to make a copy of the return value of `malloc` as soon as the C `malloc` function returns. Since the data `malloc` returns is usually uninitialized, this ends up returning a random length string containing garbage, depending on where (and whether) a nul character was found in the data. If no nul character is found, this will crash.

```
extern char *malloc(int) # WRONG!
```

Using Dynamic Linking

Dynamic linking is used by the Win32 (Windows NT or Windows 95), OS/2 (32-bit) and UNIX versions of TAWK. You don't have to do anything special to use dynamic linking. If you call an external functions that you declared with an **extern** declaration, TAWK will use dynamic linking. You don't even have to compile your program: the `awk` interpreter program can also call external functions dynamically.

Using Static Linking (DOS Version)

To perform static linking, you must own and install one of the C or C++ compilers supported by the TAWK compiler. You must do this even if you are using third party libraries and not writing any C code yourself, because when building combined TAWK/C programs, TAWK uses the linker and the C libraries provided with your C compiler. You must choose a C compiler from the list of C compilers supported by TAWK because there are subtle differences in the underlying C libraries from different C compiler vendors, so they are not interchangeable. TAWK comes with customized libraries for each C compiler that is supported.

The steps in building a statically linked combined TAWK/C program are as follows:

- 1) Prepare your TAWK code. You must place an "extern" declaration in your TAWK program for each function you plan to call that is external to TAWK, either in a C library or an object module.
- 2) Verify that the external functions you plan to call use appropriate linkage and model. For static linking, the external functions must be large model ("far") and use the "C" calling convention.
- 3) Invoke the TAWK Compiler to create the executable program. For static linking, the TAWK Compiler will compile your TAWK code and link your TAWK program with your object and/or library files.

Preparing your C code for Static Linking.

You may compile C source files into object (.obj) files or use library files containing compiled object modules. There are three requirements:

- 1) You must use the proper memory model when you compile your code or select third-party libraries. This varies with the compiler vendor and operating system. For 16-bit compilers (used for DOS) we currently support the "large" or "huge" memory models.
- 2) The TAWK libraries were compiled using the default floating point options. If your object code uses any floating point, then it must use a compatible floating point option. Simply not specifying a special floating point option is the best idea.
- 3) The functions you wish to call from TAWK must use the C calling convention. Modern C compilers support a variety of different calling conventions, including "C", "C++", "pascal", "fastcall", "stdcall", etc. The calling convention used by the C or C++ compiler can be affected both by keywords in the C or C++ source code, and by command line options that select a particular calling convention for the entire program.

ANSI Compatible C++ compilers support a compiler independent method to specify the linkage convention using the **extern "C"** keywords. Note that this method is new in C++ and will not be recognized by an older C compiler. Here is an example written in C++:

```
/* Here is the C++ function prototype: */
extern "C" char * testit(void);

/* And here is the function: */
extern "C" char * testit(void) {
    return "hello world"
}
```

If you have an older Borland or Microsoft C (not C++) compiler, you can probably use the **_cdecl** keyword to ensure C linkage. There may be either one or two underbars in the "cdecl" keyword, depending on your compiler. For example, here is a function written in C (not C++) that can be called from TAWK:

```
/* Here is the C function prototype: */
extern char * _cdecl testit(void);

/* And here is the function: */
char * _cdecl testit(void) {
    return "hello world"
}
```

Calling the TAWK Compiler for Static Linking.

The TAWK Compiler will compile your TAWK code and link your TAWK program in a single step. You must specify the following on the TAWK Compiler command line:

- a) The names of the TAWK source files, the object files, and the library files that you want to be compiled together. The filename extension must be specified with object and library files, so they will not be confused with TAWK source files. For example: mfile.obj or graphics.lib.
- b) An appropriate `-x` option to indicate that you want to create a combined TAWK/C program. Note that the `-x` option is optional if you specified object or library files on the command line, because if you did so, the TAWK compiler knows it has to call an external linker.
- c) An appropriate `-c` option to specify the C compiler you are using.

The TAWK Compiler automatically performs these steps:

- 1) The TAWK source files are compiled, and three files are created: an object file; a linker response (.lnk extension) file that is the input to the linker in step two; and a TAWK executable (.ae) file that contains additional TAWK code needed in step three below.
- 2) The TAWK Compiler calls your linker using the linker response (.lnk) file, which combines:
 - a) The TAWK object file;
 - b) Your object and library files, if any;
 - c) The C libraries supplied with your C compiler.
- 3) The TAWK Compiler then calls the awkbnd program to combine the TAWK executable (.ae) file with the regular executable (.exe) file created by the linker, to create the end product: a stand-alone executable file. The awkbnd program is included with the DOS version of the TAWK Compiler.

Specifying the Compile Mode with the `-x` Option.

There are two ways the TAWK Compiler can create combined TAWK/C programs, controlled by the `-x` option as follows:

- `-xl` Causes the TAWK Compiler to automatically call the linker. This option is the default case if you specify an object or library file on the TAWK command line and no other `-x` options. Note that you need not specify any object or library files on the command line if the C functions you wish to call are defined in the default C library supplied with your compiler.
- `-xo` Causes TAWK to create an object file that you must link yourself. The name of the object file is the same as the TAWK output file, but with the .obj filename extension. The TAWK compiler places additional TAWK code in a separate file with the same name but with a .ae extension.

After running your linker to create an .exe file, you can call the awkbnd program yourself to combine the .ae file with your executable file to create a stand-alone executable file. The awkbnd program takes one argument, which is the basename (without extension) of both the executable and .ae files. Both the executable and .ae files must be present in the same directory.

Specifying a C Compiler with the `-c` Option:

The TAWK Compiler `-c` option specifies which C compiler to use. The TAWK Compiler uses libraries tailored for the specific C compiler and attempts to call the linker supplied with that C compiler. The possibilities are:

- `-cm` Microsoft C or Quick-C version 7 or higher, or Microsoft Visual C or C++ (This is the default)
- `-cm6` Microsoft C version 6;
- `-cb` Borland or Turbo C or C++ version 3 or higher.

The above list was up-to-date at the time this manual was printed. Different C Compilers have different specific oddities, which are listed below. See the on-line readme file that accompanied your TAWK compiler for the latest information. Thompson Automation Software also sometimes does custom ports using other C compilers or operating systems on request.

Using Microsoft C

You can use Microsoft C version 6 or higher, or Microsoft Visual C or C++. When you install the Microsoft C compiler you must install the large model libraries and must build the "combined libraries". The Microsoft combined library name is "liblec.lib", and this library must be installed properly to create combined TAWK/C programs.

The Microsoft linker is creatively named: "link.exe". Former versions of DOS included a linker with the same name which is obsolete. Make SURE your PATH is set so you are getting the Microsoft linker.

The Microsoft linker requires the LIB environment variable be set to the directory containing the Microsoft libraries, so you must set it before calling the TAWK Compiler. You do not need to worry about the Microsoft linker finding the TAWK libraries: the TAWK Compiler furnishes their full path name to the Microsoft linker.

Using Borland or Turbo C:

The TAWK Compiler supports Borland C and Turbo C for DOS version 3 or higher. Specify the -cb option to the TAWK Compiler.

Note: We had a problem report that Borland C version 3.0 will not link TAWK programs when 386MAX is installed. To solve this problem uninstall 386MAX or upgrade your Borland compiler.

Debugging with Microsoft CodeView:

This discussion applies to statically linked programs for DOS or OS/2. You can not use Microsoft codeview on a TAWK program created by the awkbind program, or created with the -xl options which automatically call the awkbind program. The reason for this is that the awkbind program places the executable TAWK code in the same position in the executable file that codeview normally uses to store debugging information. To use codeview on a compiled TAWK program, you must follow these steps:

- 1) Compile the TAWK program using -xo, which creates an output .obj file, a .ae file containing TAWK executable code, and a .lnk linker response file.
- 2) Run the Microsoft linker by hand.
- 3) The resulting executable program can be run by specifying the name of the .ae file as the first argument to the program. The other program arguments come after the name of the .ae file. The name of the .ae file is removed from the program's ARGV array, so the program executes normally.

For example, to debug the combined program consisting of doit.awk and cfile.c:

```
# Create cfile.obj:
cl -c -AL -Zi cfile.c
# Create doit.obj, doit.ae, and doit.lnk:
awkc -xo doit.awk cfile.obj
# Run the linker with codeview support enabled:
link /CODEVIEW @doit.lnk
# Invoke the codeview debugger:
cv doit.exe doit.ae
```

Using RAWMODE:

The TAWK RAWMODE variable works differently in a statically linked combined TAWK/C program. In normal TAWK programs that are not linked with C, TAWK does all text mode translation, and RAWMODE works perfectly. In a combined TAWK/C program, the underlying C library is used to perform text mode translation. Specifically: changing the RAWMODE variable in a combined TAWK/C program causes TAWK to make an appropriate setmode() call to the underlying C library for stdin, stdout, stderr, and all files opened in TAWK, except those that were opened with fopen() with a specific text ("t") or binary ("b") mode.

The problems with text mode translation in a combined TAWK/C program (or any C program) are as follows:

- 1) When changing the mode translation for an input file, the new mode does not take effect until any currently buffered input for the specified file has been read. You can, however, reliably change the mode before you read any data from the file.
- 2) The least significant bit of the RAWMODE variable is ignored. This is the bit that controls how Control-Z is handled for input files. The C libraries do not support this feature.

Memory Allocation:

In the DOS operating system, low memory (the first 640K or so of memory) is a limited resource that is shared by all programs. In a combined TAWK/C program, both the TAWK and the C parts of the program require some low memory. By default, TAWK uses malloc() to allocate memory. When malloc() starts returning NULL, TAWK starts using XMS, EMS or disk space.

The Alternative Allocation Method:

TAWK has an option to allocate memory directly from DOS. This is what it does in normal TAWK programs that are not linked with any C code. Using this method has the following advantage: the TAWK system() and spawn() function calls can release this memory back to DOS during the system call to make more low memory available for the program you are attempting to execute, and this is often the difference between being able to call system() successfully or not.

Using the Alternative Allocation Method:

To make AWK allocate memory directly from DOS, simply place the following anywhere in your C code:

```
int awk_xmalloc = 1;
```

Borland C users take note: This alternative method is not recommended if your program needs to use malloc(), calloc() or etc. The Borland malloc() function does not coexist peacefully with programs that allocate memory directly from DOS. The Borland malloc() function is not able to allocate any new memory from DOS, after the program allocates any memory directly from DOS. This is a documented feature of Borland C and therefore not a bug.

Reserving memory for your C program's use:

If TAWK uses up all of low memory using malloc, there will be none available for your C program's use. However, TAWK allocates low memory only as it needs it, so for most programs with modest memory requirements, you do not need to worry about it.

You can tell TAWK to reserve a portion of low memory for use by your C program or for use by the system() or spawn() functions. To do so you set the C variable awk_heapkreserve to the amount of memory in Kilobytes that you want reserved. For example, including the following line in your C program:

```
int awk_heapkreserve = 100;
```

The above will cause the TAWK memory allocator to try to leave the last 100K bytes of main memory alone, thus reserving it for use by your C program or for spawning external programs. This number is advisory only: if TAWK can not continue without using this memory, it will allocate it anyway, but it will try to use XMS/EMS memory or disk space first. If you reserve too much heap space, the TAWK program will thrash trying not to use the memory you want reserved and your program may slow down spectacularly.

Using the Near Heap

In the Microsoft C versions 6 and 7, TAWK normally calls the _nheapmin() function to eliminate the near heap. To keep it from doing this define the variable awk_nheap in your C program and initialize it to 1 as follows:

```
int awk_nheap = 1;
```

Advanced TAWK Programming Topics.**What is the TAWK Executable File?**

The TAWK executable file contains the compiled TAWK code and the unified symbol table for your entire TAWK program. The single unified symbol table is what allows you to use global variables and functions in any TAWK source file without having to explicitly declare them in every file. The unified symbol table also makes very fast incremental compilations possible. The object file generated by TAWK contains only function entry points, no code. Under DOS the TAWK executable code is virtualized, that is, it can be read from the executable file on demand and swapped out if additional memory space is required. At runtime the DOS version of TAWK determines the optimal use of memory by allowing compiled TAWK code and your program data to share the same memory space. This technique, combined with the incremental linking feature, allows gigantic TAWK programs to be developed quite easily. Programs with more than 640K of compiled code have been developed under DOS using TAWK.

Win32 Call-Back Functions

Call-back functions are supported only in the Win32 version of TAWK. In order to create a window or dialog box using the Win32 API for Windows NT or Windows 95, you must provide a call-back function. Typically, the Win32 call-back function must use stdcall calling sequence, and must have exactly four parameters. If the call-back function meets these criteria, you can use a TAWK function with exactly four parameters as a Win32 call-back function.

TAWK provides two functions to support Win32 call-back functions:

```
registercallback("funname")
```

The TAWK registercallback() function takes as a parameter the name of a TAWK function (as a string) and returns a function pointer that can be passed to a Win32 API function as a call-back function pointer, or 0 if it fails.

```
unregistercallback("funname")
```

You can only use a limited number of TAWK functions (currently, three) as callback functions simultaneously. This function tells TAWK that the specified TAWK function is no longer being used as a call-back function.

Example:

```
# Declare the Win32 functions we will use.
extern winapi int GetModuleHandle(void*);
extern winapi int GetForegroundWindow();
# DialogBoxIndirect is a macro for
# DialogBoxIndirectParamA
extern winapi int DialogBoxIndirectParamA(int,void*,int,int,int);

# This is the call-back function written in TAWK
function dialogproc(hdlg,msg,wparam,lparam)
{
    print "The Call Back Function was Called!"
}

BEGIN {
    local hinst = GetModuleHandle(0);
    local hwnd = GetForegroundWindow()
    local callback = registercallback("dialogproc")
    if (callback == 0) {
        print "All callback functions are in use!";          abort(2);
    }
    # dialog_temp must contain a dialog box template.      # In the examples
    directory there is a real
    # program that shows how to do this.
    ret = DialogBoxIndirectParamA(hinst,
        dialog_temp,hwnd, callback,0)
    unregistercallback("dialogproc")
}
```

Sharing File Handles Between TAWK and C.

In a TAWK program, the file handles returned by fopen(), and the pre-defined file handles stdin, stdout, and stderr can be passed to C functions by declaring the corresponding argument in the extern declaration as type "long". For example, the C fputs() function can be called from TAWK as follows:

```
# Second argument is actually a FILE*
extern fputs(char*,long)

BEGIN { fputs("hello world\n",stdout); }
```

You can also determine the underlying DOS numeric file descriptor given the TAWK file handle using the built-in TAWK function fileno(). As in C, the following are always true:

```
fileno(stdin) == 0
fileno(stdout) == 1
fileno(stderr) == 2
```

The following example calls the C `_commit()` function, which causes buffered output for a file descriptor to be written to disk.

```
# Argument is a file descriptor.
extern int _commit(int)

BEGIN {
    pf = fopen("foo","w")
    print "hello world" > pf
    # fflush sends data buffered by TAWK to DOS.
    fflush(pf)
    # commit sends data buffered by DOS to disk.
    _commit(fileno(pf))
}
```

Just for interest, note that the DOS commit function can be called directly from TAWK using interrupt 0x21 function 0x68:

```
# Under DOS this is the same as _commit(fileno(pf))
interrupt(0x21,0x6800,fileno(pf));
```

Additional Information.

The file "errata.txt", supplied with the TAWK compiler, may include additional information on features added since the printing of this manual. In particular, you will find information on the "string" data type in this file.

Chapter 17: TAWK Built-In Functions

abort()

or

abort(*status*)

This function causes your TAWK program to execute the code associated with any TERM blocks, and then to exit immediately. All unwritten data to files will be saved, but any data written to output pipes that have not yet been closed will be abandoned. Use the close() function to close any output pipes that you want to be executed before using the abort() function. If a status is specified, it is the exit status of the program and must, for most operating systems, be in the range 0 to 255.

addressof(*string*)

This function is for EXPERTs. This function returns the address of a string as a 32-bit number. This is most often used to obtain string addresses for use by the interrupt() function or to pass addresses to C programs for special purposes.

[DOS and OS2 16-bit versions:]

The address is in the default format expected by the INTEL processor, which is a segment value in the high 16 bits and an offset value in the low 16 bits.

[All other versions]

The returned address is a linear 32 bit address.

Example (for DOS version):

```
# This function returns the segment portion
# of a 32-bit address.
function segmentof(x) {
    return and(0xffff,shiftr(x,16))
}

# This function returns the offset portion
# of a 32-bit address.
function offsetof(x) {
    return and(0xffff,x)
}

# This function changes the current directory
# (similar to TAWK's chdir function)
# It uses DOS interrupt 0x21 number 3b(hex).
# To execute the interrupt the registers must be:
# AX=0x3b00, DX = offset of string address,
# DS = segment of string address.
function msdoschdir(dir) {
    local addr = addressof(dir)
    interrupt(0x21,0x3b00,0,0,
        offsetof(addr),0,0,segmentof(addr))
}
```

WARNING! When you use the addressof() function, TAWK locks the string in memory as long as you are using the address, that is, as long as you retain a copy of the address in any TAWK variable or array element. When you let go of the address, the memory is unlocked. This automatic locking/unlocking of the memory used by the string is required because TAWK programs normally free strings automatically as soon as there is no longer any way to access the string from TAWK. Under DOS, even constant strings "like this" are normally virtualized so they can be swapped out of memory when they are not being used.

Normally this locking/unlocking is totally automatic and you do not need to worry about it, however, there are several places where TAWK can not keep track of the addresses that you obtain with the addressof() function:

1) when the address is saved in a structure by the `pack()` function; 2) when the address is used as an array index, because array indices are converted to strings; 3) when the address is passed to a C function which saves a copy of the address in its own static memory, and then returns.

In these cases TAWK does not know that you are still using the address, and may free or swap out the string it points to unless you also still have a copy of the address in a TAWK variable or array.

To illustrate how string address locking/unlocking works:

```
x = addressof("a string")
# The string "a string" is now locked in memory, and
# can be passed to interrupt() or to a C program.
  < code here to use x >
x = 0
# The string "a string" is now unlocked,
# and TAWK may free it to recover its memory space.
```

The following confusing program illustrates how a program can accidentally create an invalid address using the `pack()` function:

```
BEGIN {
  x[1] = addressof("a string");
  print peek(x[1])    # Prints 97 (ASCII val of "a")
  y = pack("@l",x);
  x[1] = 0;           # This unlocks "a string".
  unpack("@l",y,xx)
  # The address unpacked into xx[1] is invalid!
  print peek(xx[1])   # May print garbage
}
```

When `x[1]` was over-written in the above program, TAWK became free to discard the string: "a string" even though a copy of the address was saved in the packed structure. When the same address was later unpacked into `xx[1]` it was no longer guaranteed to point to the original string "a string". This type of error can be very difficult for you to find, because this program will work most of the time, but may fail only when TAWK is particularly low on memory.

Bitwise Logical Operators:

`and(x1,x2 ...)`
`or(x1,x2 ...)`
`xor(x1,x2 ...)`
`not(x1)`

These functions provide bitwise logical operations. They coerce their arguments to 32 bit integers and yield a 32 bit integer result. The **`and`**, **`or`** and **`xor`** functions perform the specified logical operation on each of the 32 corresponding bits in their integer arguments and return the result. The **`not`** function inverts each of the 32 bits in its argument and returns that result. The **`and`**, **`or`** and **`xor`** functions can take two or more arguments.

Examples:

```
BEGIN {
  printf("0x%x\n",and(0x20,0xff))  # prints 0x20
  printf("0x%x\n",or(0x20,0x0f))  # prints 0x2f
  printf("0x%x\n",xor(0x20,0xff))  # prints 0xdf
  printf("0x%x\n",or(1,2,4,8))    # prints 0xf
  printf("0x%X\n", not(1))        # prints 0xFFFFFFFF
}
```

See also: `shiftr()`, `shiffl()`.

argcount()

This function returns the number of arguments that were passed to the function in which it is called. The following example will print "2":

```
function foo(a,b,c) { print argcount() }  
BEGIN { foo(x,y) }
```

argval(N)

This function returns the *N*th argument of the current function. The following is a function to return the average of up to four arguments:

```
function average(a,b,c,d) {  
    local i, total  
    for (i = 1; i <= argcount(); i++) {  
        total = total + argval(i)  
    }  
    return total / argcount()  
}
```

atan2(y,x)

Arc-tangent of y/x in radians. The result is in the range -pi to pi. An easy formula to compute pi is:

```
pi = atan2(0,-1)
```

call(*funname*, *arguments ...*) and calla(*funname*, *argument_array*)

The call and calla functions perform an indirect function call. *Funname* is the name of the function to call. You can call almost any function this way, including functions written in TAWK, pre-defined TAWK functions (like print), or extern functions. The arguments to the function can be specified directly using call, or can be supplied in an array using calla. You can omit the *arguments* to call if no arguments are required.

The TAWK debugger may not be able to display stack frames below a function called with the call or calla functions. In other words, the "Calls" option in the debugger may not be able to display the functions that were in use prior to using the call or calla functions, until that function returns. This is a "feature", meaning it is a bug that we do not plan to fix.

Example:

```
BEGIN {  
    call("print","hello world")  
    # Can also put arguments in an array:  
    x[1] = "hello"  
    x[2] = "world"  
    calla("print",x)  
}
```

char(*value ...*)

The char function returns a string whose characters are specified by one or more ASCII *values*. Examples:

```
BEGIN {
    print char(65)      # prints A
    print char(65,66,67) # prints ABC
}
```

chdir(*directory*)

The `chdir` function changes the current disk drive and directory to those specified. `Chdir()` returns FALSE (0) if it fails, otherwise TRUE (non-zero).

[DOS, OS/2, Win32 Versions]

`Chdir()` operates differently from the DOS or OS/2 `cd` or `chdir` command in that it can change either the current disk drive or the directory path or both. For example: `chdir("/usr")` will change to the `/usr` directory on the current drive, `chdir("a:")` will change to the current directory on drive "a:", and `chdir("a:\\")` will change to the root directory on drive "a:". Either forward or backward slashes may be used in path names. Note that the backslash used as a directory separator must be doubled when appearing in a literal string in TAWK.

See also: `getcwd()`.

chmod(*filename,mode*)

The `chmod` function changes the mode of the specified *filename* to the specified *mode*. The *mode* can be specified as an integer or a string. If *mode* is an integer, it is a UNIX compatible integer file mode, as described further in the documentation for the `stat` function. If *mode* is a string, it must follow the following syntax:

[*who*] *op modes*

No spaces are allowed in the string.

The *who* part can be: "u" for user modes, "g" for group modes, "o" for other modes, or "a" for all three modes. If *who* is omitted, "a" is assumed. Any combination of "uog" is permitted, for example, "u", "ug", "uo", etc.

The *op* part can be: "=" to set the file mode to exactly the *modes* specified, "+" to add only those *modes* specified, or "-" to remove only those *modes* specified.

The *mode* part can be: "r" for read permission, "w" for write permission, "x" for executable permission, "h" for hidden file, "s" for system file, and "a" for the archive attribute.

Multiple modes can be set at the same time by separating them with a comma, for example: "o=r,u=rwx"

The `chmod` function returns TRUE (1) if it succeeds, or FALSE (0) if it fails.

[UNIX Version]

The "h", "s" and "a" *mode* letters are ignored.

[DOS, WIN32 and OS/2 Versions]

The *who* part is just ignored, since these operating systems have no such concept. The "r" and "x" modes are ignored, and the "w" permission controls the read-only attribute of the file.

Example:

```
BEGIN {
    # Add read and write permission for everyone:
    chmod("file", "+rw")
    # Set user permission to rwx, and
    # other and group permissions to read-only
    chmod("file", "u=rwx,og=r")
}
```

chsize(filename,size)

The *filename* can be the name of a file, or a file descriptor, such as that returned by the `fopen` function. `Chsize` sets the size of *filename* to the specified *size*. If *size* is smaller than the existing file size, the data at the end of the file is discarded. If *size* is greater than the existing file size, the size of the file is increased by adding zero bytes. If the *filename* is currently open, the size change may not take affect until you close the *filename*.

If the *filename* is currently open for reading, `chsize` will print a note message and fail.

If the *filename* is currently open for writing, note that `chsize` does not change the current location where output data is written; use `fseek` for that purpose. For example, if you are writing to a file, and want to both change the file size, and set the current output location to the new end of file location, you will have to use both `chsize` and `fseek` at the same time (in either order).

The `chsize` function returns TRUE (1) if it succeeds, or FALSE (0) if it fails.

[DOS Version]

Some DOS programs require a Control-Z character at the end of the file. If you need the file terminated by a Control-Z character, you must do it yourself, for example:

```
chsize("file",newsize)
printf("\x1a") >> "file"
```

close(x)

This function closes a file or pipe. The argument *x* can be any one of:

- 1) The name of an open file, for example:

```
getline < "myfile"
print > "outfile"
close("myfile")
close("outfile")
```

- 2) A file descriptor obtained by the `fopen` command, for example:

```
pf = fopen("myfile","r")
if (pf) {
    getline first_record < pf
    close(pf)
}
```

- 3) Input or output pipe commands. To close a pipe, specify the exact same string that was used to read or write from the pipe. Example of an input pipe:

```
"dir /w" | getline
close("dir /w")
```

Example of an output pipe example: (Note: The `sort -r` option is recognized under UNIX or Thompson Toolkit but not under DOS or OS/2.)

```
print "hello world" | "sort -r"
close("sort -r")
```

There is a limit on the number of files and pipes that may be open simultaneously, so it is important to close files or pipes that are no longer in use. Files that were opened automatically by TAWK (for example, command line arguments interpreted as files or filenames specified to the `"getline"` function) are automatically closed when the end of the file is reached. Similarly, input pipes (see example 3 above) are automatically closed when the information in the pipe is exhausted. So you only need to call `close()` in the following cases:

- Files that you open explicitly using the `fopen()` function remain open until you call `close()` to close them.

- If you wish to terminate processing a file before the end of the file is reached. HINT: To skip processing of the remainder of the current input file and force TAWK to go on to the next file on the command line, use:
`close(FILENAME).`
- Output pipes (see example 3 above) remain open until they are closed. Under DOS the commands specified by the output pipe are not actually executed until the pipe is closed or until the program ends.

It never hurts to close a file or pipe that was already closed. However, future versions of TAWK may print a "note" message if you call `close()` on the same pipe or file twice in a row.

convertnum(*str*)

or

convertnum(*str*,*base*)

Returns a number obtained by interpreting the string: *str* as an integer in the given number *base*, which must be in the range 2 to 16. If the *base* argument is not provided, `convertnum` determines the base by examining *str*. In this case, if the string begins with "0x" or "0X", or ends in "h" or "H", it assumed to be base 16, otherwise, if it begins with a "0", it is assumed to be base 8, otherwise it is base 10. If a *base* argument is provided, `convertnum` does not examine *str* to determine the base. Note that these rules are different and broader than those normally used by TAWK when converting strings to numbers.

```
BEGIN {
  print convertnum("1a",16) # prints 26
  print convertnum("1ah")   # prints 26
  print convertnum("101",2)  # prints 5
}
```

cos(*x*)

Cosine of *x*. *X* is in radians. To convert degrees to radians use:

`radians = pi * degrees / 180.`

See also: `sin()`, `atan2()` to compute pi.

ctime()

or

ctime(*timeval*)

`Ctime()` returns the time as a string in the following format:

`"Wed Jun 17 13:39:36 1994"`

If `ctime()` is invoked without arguments, it returns the current time. If a *timeval* argument is specified, then that time is returned instead. If *timeval* is specified it must be a time number such as that returned by the `time()` or `filetime()` functions.

debug_function()

debug_get_frame()

debug_get_stack_var()

debug_set_stack_var()

For Experts Only! These functions are used by the TAWK Debugger to control the program being debugged and have no use in normal TAWK programs. On-line documentation for these functions is provided in the TAWK installation directory in case you want to modify the TAWK Debugger or even write your own. Be forewarned that the TAWK Debugger is complicated and tricky.

delete(variable)

This function deletes any specified element from an array, or deletes an entire array if no particular element is specified. The parentheses are optional for this function. For example:

```
BEGIN {
    x[1] = "this"; x[2] = "that"; x[3] = "those"
    delete(x[2]) # Deletes only array element x[2]
    delete(x)    # Deletes all elements in array x.
}
```

Note that arrays are automatically deleted if they can no longer be accessed, and you do not need to specifically delete them explicitly in this case. In the following example array *x* is deleted automatically when the function returns:

```
function foo()
{
    local x
    x[1] = "x is an array"
}
```

[Compatibility Notes]

Old versions of *awk* permit the *delete* function to be called without parentheses, and this usage is supported. Prior versions of *awk* did not permit deletion of an entire array at once. Instead you had to say something like this:

```
for (i in x) delete x[i]
```

But in TAWK, it is much faster to delete the entire array at once like this:

```
delete x
```

dirlist(dirname,x)

Where: *dirname* is the directory name (for example: ".", "/", or "c:/usr/bin") and *x* is an optional array. The return value is 1 (TRUE) if the directory exists, or 0 (FALSE) if not. If *x* is specified, it is filled in with an array of the filenames found in the directory. The first filename is returned in *x*[1], the second in *x*[2], etc. Both regular filenames and directory names are included in the returned list.

For compatibility with TAWK version 4, the *getdirlist()* function is an alternate name for the *dirlist()* function.

See also: the *stat()* function to obtain more information about a file or directory.

exit()

or

exit(status)

The parentheses are optional for this function. This function causes your TAWK program to stop processing input files. The TAWK program will execute the END code blocks, if any, before it exits. If *exit* appears in an END block it terminates processing immediately. If a *status* is specified, it is the exit status of the program and must be in the range 0 to 255.

If no exit code is specified in the program, the default exit code is: 255 if an interrupt occurred; 2 if a fatal error occurred; 1 if a file specified on the command line to the Automatic Input Loop could not be opened; 0 otherwise.

NOTE:

The exit statement DOES NOT NECESSARILY exit your program! It will execute any END and /or TERM code blocks in your program first. See the *abort()* function to end a program immediately.

Example: Using Exit Codes in DOS.

The `errorlevel` function in DOS and OS/2 tests the exit code of the previously executed program. Note that it tests if the exit code is greater than or equal to the specified value. Therefore you must arrange your `errorlevel` tests in descending order, starting with the largest possible error code. For example, assuming that `myprog.awk` sets the exit code using the `exit()` or `abort()` statement:

```
awk -f myprog.awk
if errorlevel 3 goto case3
if errorlevel 2 goto case2
if errorlevel 1 goto case1
rem Code to process exit code 0 goes here
goto end
:case1
rem Code to process exit code 1 goes here
goto end
:case2
rem Code to process exit code 2 goes here
goto end
:case3
rem Code to process exit code 3 or higher goes here
goto end
:end
```

The following is an example that tests the exit code under UNIX or using the Thompson Toolkit:

```
awk -f myprog.awk
case $? in # variable $? contains the exit code
0) # Code to process exit code 0 goes here
;; # Doubled semi-colons terminate each case
1) # Code to process exit code 1 goes here
;;
2) # Code to process exit code 2 goes here
;;
*) # Code to process any other exit code
;;
esac
```

exp(x)

Computes the exponential function: e^x . In other words, e to the x th power, where e is the base of natural logarithms. To determine the value of e , simply use:

```
e = exp(1);
```

fdopen(*filedescriptor*,*mode*)

For Experts Only! This function creates a file handle that can be used within your TAWK program, given an open *filedescriptor* from the operating system. This function is similar to the `fopen` function, but it takes a *filedescriptor*, instead of a *filename*. File descriptors are normally obtained directly from the operating system, for example, by the operating system's `open()` function. Your TAWK program might receive such a file descriptor as a return value from a library called from TAWK as a DLL. In order to read from, or write to, an existing file descriptor, your TAWK program must call `fdopen` first, and use the return value from `fdopen` rather than the file descriptor itself, in calls to `getline`, `print`, etc. The *mode* argument is as described for the `fopen` function, modified as described below. You should make sure that the *mode* you specify is compatible with the mode with which the *filedescriptor* was opened originally.

[Win32 Version]

The *filehandle* can optionally be a Win32 native file `HANDLE` or a `SOCKET`. In this case, the *mode* should include "h". The following example uses the WIN32 socket function to create a new socket, and opens the socket so it can be read and written using normal TAWK functions like `fread` or `fwrite`. (This example is incomplete, because it does not include the required socket library initialization.)

```
extern int WINAPI socket(int,int,int);
function create_socket(af,type,protocol) {
    local fd
    fd = socket(af,type,protocol)
    if (fd == -1) abort(2)    # socket function failed
    return fdopen(fd,"r+h")
}
```

See also: `fileno`, `fopen`, `close`.

`fEOF(filename)`

This returns 1 if the specified *filename* contains no more data to read, 0 if there is more data, or -1 if the file is not open for reading. A file descriptor, such as returned by the `fopen()` function, can be used instead of a file name.

`ferror(filename)`

or

`ferror(filename,value)`

This returns 1 if an error has occurred on the specified *filename*, 0 if no error has occurred, or -1 if the file is not open. A file descriptor, such as returned by the `fopen()` function, can be used instead of a file name. If the optional *value* argument is specified, the error indicator for *filename* is set to that *value*, either 0 or 1. An error typically indicates an inability to read from or write to the file, for example, because the file mode does not permit the operation, the file is locked, the disk is full, or there are too few available file descriptors.

Example: Sometimes it is easier to check for errors after numerous read or write operations, rather than checking each individual read/write operation. Here is a copy command:

```
BEGIN {
    if (ARGC < 3) {
        print "Syntax: copy file1 file2"; abort()
    }
    while (x = fread(ARGV[1])) fwrite(x,ARGV[2])
    if (ferror(ARGV[1])) print "error reading",ARGV[1]
    if (ferror(ARGV[2])) print "error writing",ARGV[2]
}
```

`fflush(filename)`

Any buffered output is written immediately to the specified *filename*. The `fflush` function should only be used on files that are being written to; if called on a file open for reading, `fflush` causes a loss of all data currently buffered for input from that file. You rarely have to use this function because TAWK automatically flushes data before any `fseek` or `chsize` function, before a system or spawn call, before your program ends, and flushes after every write to files that are actually devices (for example, the console). A file descriptor returned by `fopen()` may be specified instead of the *filename*.

`filemode(filename)`

or

`filemode(filename,who)`

Returns a mode string representing the mode of the file or directory specified by the *filename*. If the *filename* does not exist, the empty string ("") is returned.

Under UNIX, the second *who* argument can be:

"user"	to return file permissions for the user (i.e., the owner of the file);
--------	------------------------------------------------------------------------

"group"	to return file permissions for the file's group;
"other"	to return file permissions for all others;
"all"	to return all permissions simultaneously.

Under DOS, OS2 and Win32 the second argument is ignored.

The letters in the returned mode string are assigned as follows:

"d"	the path is a directory;
"f"	the path is a file;
"c"	the path is a device (like the terminal);
"r"	the file can be read (this is always true under DOS);
"w"	the file can be written;
"x"	the file is executable;
"_"	used as a place holder when a permission is not granted.

For DOS, OS2 and Win32 the following may also be included:

"h"	the file is a hidden file;
"s"	the file is a system file;
"a"	the file archive bit is on.

For UNIX the following may also be included:

"b"	the path is a block special file;
"p"	the path is a named pipe or fifo;
"s"	this letter is included in the "user" or "group" file permissions if, respectively, the "set user id on execution" or "set group id on execution" mode is set for the file;
"l"	this letter is included in the "group" file permissions if the file has mandatory locking enabled;
"t"	this letter is included in the "other" file permissions if the "sticky" bit is set for the file. The "sticky" mode is not really associated with the "other" file permissions, that is just where the letter is returned. See your UNIX documentation for more information on these special modes.

Note: For any given file only one of "d", "f", "c", "b" or "p" is included in the returned string to indicate the base type of the file.

fileno(filename)

Returns the operating system's file descriptor that is associated with *filename*, or -1 if the file is not open. The operating system's file descriptor is usually a number in the range 0 to 20. This function is particularly useful when you need a file descriptor for a DOS interrupt.

When TAWK reads a file with the *getline* function, it automatically closes the underlying file descriptor when the end of the file is reached, unless the file was opened by *fopen()*. So *fileno()* may return -1 even though you just read a record from the file. To avoid this problem you can use *fopen()* on the file: after an *fopen()* call, the underlying file descriptor is never closed until your program closes the file with the *close()* function.

filesize(*filename*)

Returns the length in bytes of the specified *filename*, or -1 if the file does not exist.

filetime(*filename*)

or

filetime(*filename*,*whichtime*)

Returns the modification time of the specified filename as an integer.

[UNIX, OS/2, Win32 versions]

The second argument can be:

<u>Whichtime</u>	<u>Meaning</u>
"m" or "modification"	to return the last modification time of the file (this is the default);
"a" or "access"	to return the last access time of the file;
"c" or "creation"	to return the creation time of the file.

[DOS Version]

Under DOS, the second argument is ignored and the file modification time is always returned. (DOS does not keep track of the other times.)

findfirst(*pattern*)

or

findfirst(*pattern*,*flags*)

or

findnext()

These functions are obsolete: for new programs, use the `dirlist()` function, which is both easier to use and works uniformly on all operating systems.

These functions are used to find filenames matching a given filename pattern. The filename *pattern* uses the rules of the underlying operating system, for example, under DOS the pattern to match any filename is `"*.*"`, while under UNIX the pattern is just `"*"`. The pattern may include a directory prefix, for example `"\\usr\\bin*.*"`. `Findfirst()` returns the first filename that matches the pattern, if any, otherwise an empty string. `Findnext()` returns the next matching filename until no more filenames match the pattern, then returns an empty string. Only normal files are returned unless the optional *flags* argument is specified as one of the following values. The values (except number 8: volume id, which should be used alone) may be added together to return files matching any of the criteria. Normal files are always included in the returned files, even if *flags* is specified.

<u>Flags</u>	<u>Meaning</u>
2	return hidden files (Has no effect in UNIX version);
4	return system files (Has no effect in UNIX version);
8	return the volume id (DOS version only.) Note: the volume label is returned in 8.3 filename format, for example: "myfirstd.isk". You must remove the "." yourself.
16	return sub-directories;

Example: The following program prints the names of all normal files in the current directory. If you are using DOS, use "*" instead of "*.*":

```
BEGIN {
  x = findfirst("*")
  while (x) {
    print x
    x = findnext()
  }
}
```

flock(filename,offset,cnt)

or

flock(filename,offset,cnt,mode)

Locks the specified area of the *filename*. The locked area begins at the location indicated by *offset*, where 0 is the first byte of the file, and continues for *cnt* bytes. It is the programmer's responsibility to ensure that all locked regions are eventually unlocked by a call to *funlock()*. A file descriptor returned by *fopen()* may be specified instead of *filename*.

The optional *mode* argument may be:

<u>Mode</u>	<u>Meaning</u>
"x"	exclusive lock: other processes are denied access to the area.
"s"	shared lock: other processes may also lock the area. This option is ignored under DOS and OS/2: in these operating systems all locks are exclusive.
"n"	may be used with either "s" or "x" to indicate a non-blocking lock, that is, if the lock can not be obtained, the flock function returns immediately rather than waiting.
absent	same as "x".

[DOS Version]

Locking will not work under DOS unless the "share.exe" program or some other file-sharing controller like Microsoft Windows 3.1 is in use.

See also: *funlock()*

fopen(filename,mode)

Opens the specified *filename* for subsequent input/output operations of a type specified by the *mode* argument. TAWK will normally open files for you automatically the first time you use the filename with an input/output function, for example, *getline* or *print*. However, you can use *fopen* if you want to open the file yourself. For example, if you want your program to detect errors when opening files, it is most easily done by using *fopen* to open the files deliberately, and checking the *fopen* return value for 0. It is also especially advisable to use *fopen()* in the following cases: 1) If you want to specify an input/output mode other than the default ones, for example, if you want to both read from and write to a file simultaneously, or if you want to specify a binary file and it is inconvenient to use RAWMODE. 2) If you want to start reading a file at a specific location: use *fopen()* to open the file, then *fseek()* to move to the specified location: *fopen* is needed because you can not use *fseek()* until you have opened the file.

The *mode* argument must start with one of the following:

<u>Mode</u>	<u>Meaning</u>
"r"	File is opened for reading only.

"r+"	File is opened for reading and writing.
"w"	File is created for writing only. If the file previously existed its former contents are lost. The file position for writing is set to the beginning of the file.
"we"	Like "w", but fails if the file already exists. This can be used in multi-tasking environments to guarantee that a newly created file was not simultaneously opened by another process.
"w+"	Like "w" but reading is also allowed.
"w+e"	Like "we" but reading is also allowed.
"a"	File is opened for writing only. If the file did not exist it is created; if the file did exist the current file position is set at the end of the file so that written data will be appended to the end of the file.
"a+"	Like "a" but reading is also allowed.

Fopen Text/Binary Mode:

[DOS, OS/2 and Win32 Versions]

The end of line marker under UNIX is "\n" (a newline character.) Under DOS and OS/2 it is a "\r\n" (carriage-return followed by a newline.) Under DOS and OS/2 files can be opened in either "text" or "binary" mode. In "text" mode TAWK automatically translates the "\r\n" to "\n" on input, and "\n" to "\r\n" on output, and additionally allows a Control-Z character in the file to indicate end-of-file. In "binary" mode the file is not treated specially and no translation is performed.

The file mode argument may include a letter to indicate "text" or "binary" mode for the file under DOS or OS/2. Under UNIX these mode letters are ignored. If a file mode is specified then that mode is used for the file and the RAWMODE variable has no affect on this file. The following mode letters may be appended to the file mode:

<u>Mode</u>	<u>Meaning</u>
"b"	The file is opened in "binary" mode: no translation is performed and the file is not treated specially.
"t"	The file is opened in "text" mode: On input "\r\n" is translated to "\n", on output "\n" is translated to "\r\n", and a control-Z in the file is treated as an end-of-file marker.

Fopen Sharing Mode:

[DOS, OS/2 and Win32 Versions]

Under DOS and OS/2 the following letters can also be appended to the file mode to specify that the file is to be opened with a specified file-sharing mode. Sharing modes have no affect under DOS unless the "share.exe" program or some other file-sharing controller (like Microsoft Windows 3.1) is in use. Once you have opened a file with a file sharing mode, you can lock or unlock parts of the file using flock() or funlock(). The following are the supported file-sharing modes for both DOS and OS/2:

<u>Mode</u>	<u>Meaning</u>
"dr"	File is opened in DENY-READ sharing mode;
"dw"	File is opened in DENY-WRITE sharing mode;
"d+"	File is opened in DENY-READ-WRITE sharing mode;
"dn"	File is opened in DENY-NONE sharing mode; (The deny-none mode is confusing: it actually denies access to all other programs that do not specify a compatible sharing mode.)

Miscellaneous Fopen Modes:

The following additional mode letters are supported, if the facilities are available in the underlying operating system.

<u>Mode</u>	<u>Meaning</u>
"n"	No-inherit mode. The returned file descriptor will not be inherited by sub-processes. This may make more file descriptors available for programs executed with TAWK's system or spawn functions.
"e"	Exclusive open mode. If the file is being opened for writing, fopen will fail if the file already exists. This is sometimes used to create semaphore files.

Examples: To open file myfile.txt in the root directory of drive C: for reading:

```
fopen("C:\\myfile.txt", "r")
```

To open a database file for both reading and writing in binary mode with deny-none sharing mode:

```
fopen("\\dbase\\myfile.db", "r+bdn")
```

Fopen fails and returns 0 (FALSE) if the file could not be opened for any of the following reasons:

- file opened with mode "r" did not exist;
- the directory did not exist or was full;
- the file had a read only attribute and the file mode specified writing;
- a sharing violation occurred;
- insufficient privileges to access a file or directory;
- all file descriptors are already in use. (In this last case TAWK prints a warning message.)

If fopen succeeds, it returns a non-zero file descriptor that can be used interchangeably with the filename for all subsequent file operations on the file. However, you can also just ignore the file descriptor and continue to use the file name for file operations. For example, the following are all legal:

```
pf = fopen("myfile", "r")
getline < "myfile"
getline < pf
fread(10, "myfile")
fread(10, pf)
```

More Details About Fopen:

When you use the getline or print/printf functions to read from or write to a file, you do not need to use fopen() to open the file first. In this case files are opened automatically with the default modes as follows:

<u>function</u>	<u>default mode</u>	<u>meaning</u>
getline < file	"r"	file is open for reading only
print > file	"w"	file is open for writing only
print >> file	"a"	file is open for writing only, data is appended to any existing file.

When you intermingle print>"file" and print>>"file" statements, the first print (or printf) statement for the "file" encountered at execution time determines the file mode. Subsequent print (or printf) statements simply continue to write to the already opened file.

An incompatible file I/O operation is one that attempts to read from a file opened for writing-only, or to write to a file opened for reading-only. If the file was opened by fopen(), then incompatible operations are disallowed. If the file was opened automatically, then incompatible operations are allowed, but the file is automatically closed and re-opened with the new mode, and TAWK prints a warning message.

fread(*cnt*,*filename*)

Reads *cnt* bytes from *filename* and returns them as a string. If an error occurs an empty string is returned. If the file is exhausted, a string of less than *cnt* bytes in length will be returned. A file descriptor returned by `fopen()` may be specified instead of a *filename*.

fseek(*filename*,*location*), or fseek(*filename*,*location*,*flag*)

After calling this function, the next read or write operation on *filename* will occur at the specified *location* counted in bytes from the start of the file. The file must be open before using this function. The optional *flag* argument may be any of the following:

<u>Flag</u>	<u>Meaning</u>
0	The location is measured as an offset from the beginning of the file. This is also the default if no flag is specified.
1	the location is measured as an offset from the current location.
2	the location is measured as an offset from the end of the file.

A file descriptor returned by `fopen()` may be specified instead of *filename*.

ftell(*filename*)

Returns the current location in the *filename* in bytes from the beginning of the file. If the *filename* is not open -1 is returned. A file descriptor returned by `fopen()` may be specified instead of *filename*.

funlock(*filename*,*offset*,*cnt*)

Unlocks the specified area of the *filename*. The unlocked area begins at the location indicated by *offset*, where 0 is the first byte of the file, and continues for *cnt* bytes. It is the programmer's responsibility to ensure that all file regions locked by `flock()` are eventually unlocked by `funlock()`. A file descriptor returned by `fopen()` may be specified instead of *filename*.

fwrite(*str*,*filename*)

Writes the string argument: *str* to the *filename*. The number of bytes written from *str* is returned. If an error occurs 0 is returned. Note that the actual number of bytes written to the file will be greater than the return value because new-line ("`\n`") characters are translated to carriage- return new-line ("`\r\n`") sequences unless the RAWMODE variable specifies otherwise. A file descriptor returned by `fopen()` may be specified instead of *filename*.

getawkvar(*variable_name*)

Returns the value of the specified TAWK variable. Only variables that are declared global and that are not arrays can be specified. (An exception: if your program consists of a single source file then all variables are automatically considered global.) The following example prints "10":

```
global x
BEGIN {
    x = 10
    print getawkvar ("x")
}
```

getcwd()

The `getcwd()` function returns the current directory name. On PCs, it will include the disk drive specifier, for example: "D:/usr/patt". See also: `chdir()`.

getkey()
 or
getkey(1)

`Getkey()` returns a key from the keyboard. If none are ready it waits. If the key is a printing character, including tab, back-space, carriage-return, newline or formfeed, that character is returned. Otherwise `getkey()` returns the name of the key, for example "F1", "HOME" or "ESC". If the key is an Alt-, Ctrl- or Shift- combination the key name is prefaced with "A-", "C-" or "S-", respectively. For example the key combination shift- F1 returns "S-F1".

The behavior of the `getkey()` and `kbhit()` functions is undefined for the keys Control-C, Control-S, Control-P, Print-Screen and Control-Print-Screen.

[DOS and OS/2 Versions]

If `getkey` is given an argument of 1 it returns the raw 16 bit integer corresponding to the IBM key-code of the key pressed.

[DOS and DOS/32 Versions]

Under DOS this function uses the keyboard interrupt, so it will only work on IBM compatible computers. Non-IBM compatible computers are likely to crash if this function is used, but almost all computers are IBM-compatible in this regard.

[UNIX Versions]

Calling `getkey` or `kbhit` puts the terminal in "cbreak" and "noecho" mode. The terminal is automatically returned to the original mode when a `getline`, `fread`, `system` or `spawn` function is called, or when the program ends. `Getkey` attempts to recognize escape sequences, beginning with an ESC (27) character, as special keys, such as function keys. The escape sequences that are valid are retrieved using the "curses" database. This method does not always work! Additionally, some keyboards require programming, which TAWK does not attempt to do. The result is that sometimes function and special keypad keys are recognized, and sometimes not.

See also: `kbhit()`.

getline()
 or
getline(*variable_name*)

The `getline` function returns an input record either from the file being read by the Automatic Input Loop, or from a specified file or pipe. The parentheses are optional for this function.

To read from the Automatic Input Loop use:

```
getline
or
getline var
```

The record is returned in the variable: *var*, if specified; otherwise it is placed in the current record: \$0. As always, whenever \$0 is changed the corresponding fields: \$1, \$2, etc. and the NF (Number of Fields) variable are also updated. The NR (Total Number of Records) and FNR (Number of Records from current File) variables are also incremented.

To read from a specific file, use:

```
getline < "filename"
or
getline var < "filename"
```

The record is returned in the variable: *var*, if specified; otherwise it is placed in the current record: \$0. The "*filename*" can be specified as a string as shown, or can be a variable, or any TAWK expression. As a special case, if the filename is "-" the record

is read from the standard input. This form of `getline` does not increment the `NR` and `FNR` variables, unless the specified *filename* is the current input file being processed by the Automatic Input Loop. This form of `getline` also prevents the `BEGINFILE` and `ENDFILE` blocks from being executed automatically.

To read from a command using a pipe:

```
"command" | getline
or
"command" | getline var
```

This is called a "pipe" because it pipes the output from the specified *command* into your program. The record is returned in the variable: *var*, if specified; otherwise it is placed in the current record: `$0`. The *command* is a string containing the full command line of the command to run, and can be specified as a string as shown, or can be a variable, or any TAWK expression. The pipe works as follows: The *command* is executed once the first time it is used with `getline`. The records returned by `getline` are retrieved from whatever is output to the standard output by the specified command. Subsequent calls to `getline` using the identical *command* cause successive records to be returned from the command output until all records have been read.

In all cases the `getline` function uses the current values of the `RS` and `RECLLEN` variables to determine the format of the next record to read.

Return Value: `getline` returns 1 if a record is returned. It returns 0 if there is no more input available either from the Automatic Input Loop, the specified file, or the pipe. It returns -1 if it fails. Failure is most often caused by the specified file not being found or already being used in a shared environment, or if the pipe command fails.

After reading all the records from a file or pipe `getline` will continue to return 0 until you use the `close()` function to close that file or pipe. So if you want to start over reading at the beginning of a file or to re-execute a pipe command you must use the `close()` function.

There are limits on the number of files and/or pipes you can have open simultaneously. If you only read part of a file or pipe and are done with it you should close it using the `close()` function to free the operating system resources (like file descriptors) that were in use. It is not strictly necessary to `close()` files or pipes that you read to completion (that is, after `getline()` returns 0), because TAWK automatically releases the operating system resources in this case.

Example:

The following program reads in a file named: "filename", and simply prints it back out:

```
BEGIN {
    while (getline x < "filename" > 0) {
        print x
    }
}
```

WARNING:

The following example is incorrect:

```
BEGIN {
    while (getline x < "filename") {
        print x
    }
}
```

The above example does not test the return value of `getline` > 0. If the file: "filename" happens to not exist or be inaccessible for some other reason, `getline` returns -1, and this program will be stuck in an infinite loop.

gsub, gsubs

See the `sub` function.

index(*string1*,*string2*)

or

index(*string1*,*string2*,*start*)

This function returns the index where *string2* occurs in *string1*, or 0 if *string2* is not found in *string1*. If *start* is specified, it is the index where the the search begins in *string1*. For example:

```
index("abc", "a")      # returns 1
index("abc", "b")      # returns 2
index("abc", "d")      # returns 0
index("abcbc", "b", 3) # returns 4
```

In the special case where *string2* is empty ("") the index() function returns 1 if *string1* is non-empty, and zero otherwise, that is:

```
index("abc", "")      # returns 1  (a special case)
index("", "")         # returns 0
```

See also: rindex() function.

inp(*port*)**inpw(*port*)**

These functions perform a low level I/O function: input byte (inp) or input word (inpw) from the specified I/O *port*. The value retrieved from the input port is returned. Under OS/2 or UNIX these functions do nothing.

See also: outp(), outpw().

int(*x*)

Returns the integer part of *x*. The result may still be a floating point number (with a fractional part of 0) if it is too large to express as a 32 bit integer.

```
print 3/2      # Prints "1.5"
print int(3/2) # Prints "1"
print int(-3/2) # Prints "-1"
```

interrupt(*intno*,*AX*,*BX*,*CX*,*DX*,*SI*,*DI*,*DS*,*ES*)

[WARNING]

USE THIS FUNCTION WITH CARE! YOU CAN CRASH YOUR COMPUTER OR LOSE FILES BY IMPROPER USE!

[DOS and DOS/32 Versions Only]

If you are running under DOS this function performs interrupt number *intno* with the registers set to the specified register values. The *intno* argument is required. The other register arguments are optional, but if specified they must be given in the order shown. If you specify a variable for any register argument(s), the variable(s) will receive the values that were placed in those register(s) after the interrupt returns. (This is a technique called pass-by-reference, and is used in only a few places in TAWK.) The interrupt function returns the value of the carry flag after the interrupt.

Example: to get the current date you could use MS-DOS interrupt 0x21 function 0x2A, which returns the year in CX and the month and day in DX as follows. The interrupt requires AH (high half of AX) to be 0x2A. The BX register is not used by the interrupt, but a value of 0 is specified as a place holder because we need CX and DX.

```
interrupt(0x21, 0x2A00, 0, year, moday)
```

Documentation on interrupts is available from many sources, including "MS-DOS Functions" by Ray Duncan, Microsoft Press. See also: addressof(), OSMODE.

[DOS/32-bit Version:]

Only interrupt 0x21 is supported and string addresses must be handled specially. String addresses are normally passed in a register

pair, for example ES:DX. In the 32-bit version you must pass the entire 32-bit address returned by `addressof()` in the register that is meant to receive the offset (DX in the example above.) The interrupt function automatically sets the DS and ES registers to the correct values. Do NOT pass any values for DS or ES, or you will probably generate an error. If you are trying to write TAWK code that will work when compiled both for normal DOS and 32-bit extended mode DOS, you will have to check the OSMODE variable to determine which version is being used and call the interrupt function appropriately for each version.

[Other Operating Systems]

Under other operating systems this function does nothing.

kbhit()

`Kbhit()` returns non-zero only if a key is ready at the keyboard. If a key is ready it returns the key in the same format as that returned by the `getkey()` function. The behavior of the `kbhit()` function is undefined for the keys Control-C, Control-S, Control-P, PrintScreen and Control-PrintScreen. This function uses the keyboard interrupt so it will only work on IBM compatible computers. Non-IBM compatible computers are likely to crash if this function is used.

length(expr)

If *expr* is a string then the number of characters in the string is returned. If *expr* is an array then the number of elements in the array is returned. The length function can be specified without parentheses for backward compatibility with `awk`.

log(x)

Natural logarithm of *x*. Example: the following function computes base 10 logarithms:

```
function log10(x) { return log(x) / log(10) }
```

match(string,pattern)

or

match(string,pattern,start)

or

match(string,pattern,start,pstart,plength)

This function returns the index in the *string* that is matched by the *pattern*, or 0 if it does not match. In addition the `match()` function sets TAWK global variables as follows: `RSTART` is set to the index in the string of the first character matched by the pattern (this is the same as the return value of the match function); `RLENGTH` is set to the length of the string that was matched by the pattern.

If the optional *start* parameter is specified it is the index in the string where the search begins in the string. *Start* defaults to 1, which means search the entire string. If the optional *pstart* and *plength* variables are specified they must be variable names that will be filled in with arrays that contain information about the match as follows:

- `pstart[0]` The index in the string that was matched by the entire pattern (This is the same as `RSTART` above.)
- `pstart[N]` The index in the string that was matched by the Nth left parenthesis in the pattern. *N* is an integer from 1 to the number of pairs of parentheses in the pattern.
- `plength[0]` The length of the string that was matched by the entire pattern. (This is the same as `RLENGTH` above.)
- `plength[N]` The length of the string that was matched by the Nth set of parentheses in the pattern. *N* is an integer from 1 to the number of pairs of parentheses in the pattern.

The match function is very similar to the TAWK tilde (~) operator except that the `match()` function allows you to specify a starting index and it returns more information about the match. The following two statements are identical except that the `match()` function sets `RSTART` and `RLENGTH`:

```
if (x ~ /pattern/) ...
```

```
if (match(x,pattern)) ...
```

Here are some more examples:

```
BEGIN {
  match("abc",/b/)      # Sets RSTART=2, RLENGTH=1

  match("abbbc",/b*/)    # Sets RSTART=2, RLENGTH=3

  match("abbbc",/b*/,3)  # Sets RSTART=3, RLENGTH=2

  # The following sets: RSTART=2, RLENGTH=4,
  # pstart[0]=2, plength[0]=4,
  # pstart[1]=3, plength[1]=2
  match("abccde",/b(c*)d/,1,pstart,plength)
}
```

mkdir(*dirname*)

Function creates directory *dirname*. Returns 1 (TRUE) if it succeeds, or 0 (FALSE) if it fails. See also: `rmdir()`, `rmfile()`, `rename()`

not(*x1*)

See `and()`.

or(*x1,x2*)

See `and()`.

ord(*str*)

or

ord(*str,pos*)

Returns the ASCII sequence number corresponding to the first character of the string argument: *str*. If the optional *pos* argument is given, **ord** returns the ASCII sequence number of the character at that character position in the string, or -1 if *pos* is less than one or greater than the number of characters in the string. This function is the inverse of the **char** function.

Examples:

```
BEGIN {
  print ord("A")      # Prints 65
  print ord("ABC",2)  # Prints 66
  print ord("ABC",4)  # Prints -1
}
```

outp(*port,value*)

outpw(*port,value*)

[WARNING]

USE THESE FUNCTIONS WITH CARE! YOU CAN CRASH YOUR COMPUTER OR LOSE FILES BY IMPROPER USE!

These functions perform a low level I/O function: output byte (outp) or output word (outpw) to the specified I/O *port* with the specified integer *value*. Under OS/2 or UNIX these functions do nothing. The DOS version of TAWK includes an example program called "play.awk" that utilizes these functions to control the computer output speaker.

See also: inp(), inpw().

pack(template, val, ...)

or

pack(template, array)

The pack() function packs the specified values into a binary structure described by the *template* string and returns the string containing the resulting structure. The values can be specified immediately in the pack() function call (as in the first case above) or provided in an array (second case above.) The unpack() function can be used to unpack a structure back into an array of values using the same *template* string that was used to pack it. These two functions together can be used to read/write to binary data-bases or to pack/unpack structures to be passed to/from C programs.

The template string consists of one or more binary format specifications, separated by spaces. Each format specification looks like the following, and must appear all together without any intervening spaces. Square brackets: [] indicate optional items:

[*name*] @ [*flags*] [*count*] *type*

The *name* is the optional field name, the "@" sign is required, the *flags* and *count* are optional, and the *type* is a required character that indicates the binary type of the field. The type character is chosen from the following:

<u>Type</u>	<u>Meaning</u>
a	ASCII string, nul (zero) padded
A	ASCII string, space padded
b or B or c	byte character. If the value is a number the least significant byte is used. If the value is a string the first character is used.
s or S	short (16-bit) integer
l or L	long (32-bit) integer (l is the letter ell)
f	single precision floating point number
d	double precision floating point number
x	output a 0 byte
X	back up a byte
p	32-bit pointer to character string. If the value is a number, it is assumed to already be a pointer and is stored without modification as a long integer; if the value is a string, a 32-bit pointer to that string is stored. Note that strings used in TAWK are deallocated when no longer needed. It is the PROGRAMMER'S responsibility to make sure that the string pointed to by the "p" type remains valid over the life of the structure. You can do this by keeping a reference to the string in a TAWK variable or array. The internal details of how TAWK handles strings are covered in chapter 16.

Note: the "s" and "S" types are the same when used with pack, but affect how the items are unpacked. Similarly for "l" and "L", and for "b", "B" and "c". See unpack() for more information.

Pack Template Optional Numeric Count:

remember 8086 clump
pack least sig byte first

The optional *count* is a number that indicates the number of characters in the field for types "a" and "A", or a field count of the number of values to be consumed from the list of values for any other type. If no count is specified for types "a" and "A" the length of the string (plus one for a terminator character) is used.

Pack Template Optional *Flags*:

The "a" and "A" types may optionally be preceded by the following *flags*:

<u>Flag</u>	<u>Meaning</u>
+	Means right justify. (Default is left justify.) A field length must be specified with this flag.
&	Ignored by pack() function, used by unpack() function to prevent stripping spaces or nuls from field;

The "s", "S", "l", "L", "d" and "f" types may optionally be preceded by the following *flags* to control the byte ordering and packing. By default, the byte ordering is whatever the native ordering is on the host processor, and the items are packed together as closely as possible.

In addition, the <, >, and # flags may appear in the template string outside of a format specification, in which case they apply to all format specifications that follow them. For example: "@<S @<S" and "<@S @S" are equivalent.

<u>Flag</u>	<u>Meaning</u>
>	Pack bytes most significant byte first (default for most RISC processors.)
<	Pack bytes least significant byte first (default for Intel 8x86 type processors.)
#	Insert padding before the field so that it is aligned on an N byte boundary, where N is the length of the item in bytes. This type of structure packing is used by many compilers.

rexx D2x(1143) => 0477x
 in 4 bytes
 00 00 04 77
 in PAD
 77 04 00 00

If the values are provided to the pack() function in an array then the optional field *names* specified in the template string indicate the indices of the fields in the array. If no field *names* are specified in the template or if a field count is used then pack() assumes the array indices are integers: "1", "2", ...

Pack Template Examples:

"customer@a"	customer is a nul terminated alpha field.
"high@s"	high is a short (16-bit) integer field.
"name@+10A"	name is a 10 character right justified space-padded field.

The following example illustrates how to pack and unpack a structure called "complex" containing two floating point numbers named "real" and "imag".

```
# Note: In the C language this structure would be
# declared as: struct complex { float real, imag; }
local complex = "real@f imag@f"

# Create an array (cnum) holding a complex number:
cnum["real"] = 17
cnum["imag"] = 22
```



```

# Pack the complex number into a string.
# String x is created as an eight byte string
# containing the binary representation of the
# numbers 17 and 22 in binary floating point format.  x = pack(complex,cnum)

# X can be unpacked back into an array as follows:
# Array cnum2 is created as an exact replica of
# the original array cnum.
unpack(complex,x,cnum2)

# Note that you could also create x like this:
x = pack(complex,17,22)

# The binary field names are optional.
# This example creates a 10 byte string with
# the byte values 65 to 74:
y = pack("@10b",65,66,67,68,69,70,71,72,73,74)

# 65 is the ASCII value of the character "A"
# 66 of "B", etc, so this prints "ABCDEFGHJIJ":
print y

# If you unpacked the above you would get:
# tab[1] = 65, tab[2] = 66, etc.
unpack("@10b",y,tab)

# The following two pack examples produce
# the same result in variable y:
str = "a string"
y = pack("@p",str)
y = pack("@l",addressof(str))

```

2 5 B B
"68" 83 66 66

`paste(string, string, ... , separator)`

or

`paste(list, separator)`

The **paste** function returns a new string created by concatenating (pasting) together two or more strings, separated by strings specified by the *separator* argument. The strings to be pasted can be specified directly as the *string* arguments in the first syntax shown above, or can be provided in an array as the *list* argument in the second syntax shown above. The *separator* argument can also be a string or an array. If the *separator* argument is a string, it is placed between each of the pasted strings. If the *separator* argument is an array, the numbered elements of the array are used as the separator strings. If there are too few elements in the *separator* array, the last element is used as many times as necessary. To paste together the strings with no separator, use an empty string ("") for the separator. Examples:

```

BEGIN {
    print paste("a","b","c")    # prints acb
    x[1] = "a"; x[2] = "b"; x[3] = "c"
    y[1] = "e"; y[2] = "f"
    print paste(x," ")          # prints: a b c
    print paste(x,y)            # prints: aebfc
}

```

See also: **split**, **splitp**, which are the opposite of **paste**.

`peek(address)`

The **peek()** function reads and returns the byte at the specified memory *address* as an integer. The address is specified in the native format of the operating system. Under DOS (except DOS/32) and OS/2 it is in 32 bit segment:offset form, that is, the upper 16 bits of address are the segment and the lower 16 bits are the offset.

Hint: To convert the integer return value to a one character string use: `sprintf("%c",peek(address))`.

See also: `poke()`.

poke(address,value)

[WARNING]

USE THIS FUNCTION WITH CARE! YOU CAN CRASH YOUR COMPUTER OR LOSE FILES BY IMPROPER USE!

The `poke()` function sets the byte at the specified memory *address* to the *value*, which must be an integer. The address is specified in the native format of the operating system. Under DOS (except DOS/32) and OS/2 it is in 32 bit segment:offset form, that is, the upper 16 bits of address are the segment and the lower 16 bits are the offset.

Hint: To poke the first character of a string use: `poke(address,ord(string))`.

See also: `peek()`.

print

or

print(argument, argument ...)

This function prints its *arguments* separated by the value of the OFS (Output Field Separator) variable and terminated by the value of ORS (Output Record Separator) variable. If no *arguments* are specified the current line (\$0) is printed.

By default, the OFS variable is a space and the ORS variable is a newline ("`\n`") so this function prints its arguments separated by spaces and automatically terminates the line.

Parentheses are optional for this function. For example: `print` and `print()` are equivalent.

The **print** and **printf** functions can be followed by a redirection indicating that the output is to go to a file or pipe. The allowed forms of redirection are:

`print(arguments)`

If there is no redirection then output goes to stdout (standard output.) If the standard output was not redirected when the command was invoked it appears on the user's console.

`print(arguments) > "filename"`

Output goes to the specified *filename*. If this is the first print or printf to this particular file then the new data over-writes the existing data in the file, if any. The file will be created if it did not exist. The filename can be specified with a literal string as shown or in a variable or any valid TAWK expression that evaluates to a string containing a valid filename.

`print(arguments) >> "filename"`

Output is appended to the specified *filename*. The file will be created if it did not exist.

When `>` or `>>` redirections are used it is the first print or printf statement encountered when the program executes that determines whether a file will be over-written or appended to. After the file is opened the `>` and `>>` symbols just continue to append new data to the already opened file.

`print(arguments) | "command"`

This is an "output pipe". The data is sent to the input of the specified *command*. The *command* can be specified with a literal string as shown or in a variable or any valid TAWK expression that evaluates to a string containing a command.

Under some operating systems the command is executed asynchronously, that is, it is started up and each print or printf sends additional data to the command. Under DOS the data written to the pipe is saved in a temporary file and the command is not executed until the the pipe is closed or the program ends.

Notes:

- A common mistake is to forget to put quotes around the output filename. The following code prints to the file specified in the TAWK variable: filename rather than to the file "filename".

```
print > filename
```

The following are correct methods to print to a file named "filename":

```
print > "filename"
```

```
x = "filename"
print > x
```

- Another common mistake is to try to print a blank line with this:

```
print
```

But a "print" statement without arguments prints the current record (\$0). To print a blank line use this:

```
print ""
```

- If one of the arguments to **print** contains a greater-than (>) symbol then the argument list must be parenthesized to prevent confusion over whether the > symbol means greater-than or output redirection. For example, the following statement is ambiguous:

```
print a > b
```

Change it to one of the following:

```
print(a > b)
print(a) > b
```

printf(*format*)

or

printf(*format, argument, ...*)

This function performs formatted printing giving the user complete control of the resulting output. This function prints the optional list of *arguments* after performing conversions specified by the *format* string.

The parentheses are optional. For example, `printf("hello")` and `printf "hello"` are equivalent.

The format string may contain regular characters that are just printed and format specifications that always begin with the % character. The format specifications in the format string are interpreted from left to right and each specification tells printf how to format the next argument to printf. For example, both of the following print "hello world":

```
BEGIN {
    printf "hello world\n"
    printf "%s %s\n", "hello", "world"
}
```

The above examples include "\n" in the string because printf, unlike print, does not automatically append a new-line character to terminate the output line unless it is included in the string.

Each format specification looks like the following. No spaces may appear in the format specification except as specifically allowed below. Square brackets: [] indicate optional items:

```
% [flags] [WWW] [.NNN] x
```

The meanings of the parts of the format specification are:

% The % character is required.

flags The flags are optional characters that modify the formatting as described further below.

- WWW** This is an optional numeric field width. If the converted value has fewer bytes than this number it is padded to fit this field width.
- .NNN** This indicates an optional numeric precision argument. If it is specified it must be preceded by a period. The meaning of this argument depends on the format: For integer number formats this argument indicates the minimum number of digits to be printed. For %e and %f it indicates the number of digits to appear after the decimal point. For %g it indicates the maximum number of significant digits to be printed. For %s it indicates the maximum number of character bytes to be printed.
- x** This represents a format character, which must be one of those described below.

The valid format characters are as follows:

<u>Format</u>	<u>Meaning</u>
%c	ASCII character. If the argument is a string the first character of the string is printed. If the argument is a number the corresponding ASCII character is printed.
%d	signed integer in decimal number format.
%i	same as %d
%u	unsigned 32-bit integer in decimal number format.
%b	unsigned 32-bit integer in binary number format.
%o	unsigned 32-bit integer in octal number format.
%x	unsigned 32-bit integer in hexadecimal number format.
%e	floating point number in format: [-]d.ddddde[+-]ddd
%f	floating point number in format: [-]ddd.ddddd
%g	like %e or %f, whichever is shorter, with nonsignificant trailing zeros suppressed.
%s	string of characters.
%m	money format: negative numbers appear in parentheses, and the default precision is two decimal places. Positive numbers are preceded and followed by a space so that positive and negative numbers printed in a column can easily be lined up on the decimal point simply by specifying a field width, for example %10m.
%X	like %x but uses capital letters "A" through "F".
%E	like %e but print an uppercase "E" instead of "e".
%G	like %g but print an uppercase "E" instead of "e".
%%	just prints a regular % character and does not consume any of the printf arguments.

The following flag characters may appear in a format specification following the percent character:

<u>Flags</u>	<u>Meaning</u>
-	The result is left-justified within the field width. Normally if a field width is specified, then the result is right justified within the field width.
+	Signed numbers will be preceded by a plus sign if the number is positive.
<space>	(a space character) Print a space preceding the number if it is positive. This is used to help line up numbers in columns.

- ^ Use engineering notation for %e or %g specifications.
- # Always print a decimal point for %e, %f, or %g specifications. Normally the decimal point is suppressed if there are no zeros following it.
- 0 If a field width is specified with a numeric format that is right justified (that is, the - flag is not given), then the number is expanded to fit the field width with extra leading 0 characters rather than with space characters.
- z Suppresses trailing insignificant zeros in %e or %f format.
- l An l character is ignored. In the C language it means a long number but all numbers in TAWK are long by default.

The field width (WWW above) and/or precision (.NNN above) can be specified as an asterisk (*) instead of a number. In this case the number to be used for the field width or precision is obtained from the next printf argument, which is consumed. Thus each printf format specification can consume either 1, 2 (if either field width or precision is a *) or 3 arguments (if BOTH field width and precision are a *) from the printf argument list.

Redirecting Printf Output:

The **print** and **printf** functions can be followed by a redirection indicating that the output is to go to a file or pipe. The allowed forms of redirection are:

`printf(format, arguments)`

If there is no redirection then output goes to stdout (standard output.) If the standard output was not redirected when the command was invoked it appears on the user's console.

`printf(format, arguments) > "filename"`

Output goes to the specified *filename*. If this is the first print or printf to this particular file then the new data over-writes the existing data in the file, if any. The file will be created if it did not exist. The filename can be specified with a literal string as shown or in a variable or any valid TAWK expression that evaluates to a string containing a valid filename.

`printf(format, arguments) >> "filename"`

Output is appended to the specified *filename*. The file will be created if it did not exist.

When > or >> redirections are used it is the first print or printf statement encountered when the program executes that determines whether a file will be over-written or appended to. After the file is opened the > and >> symbols just continue to append new data to the already opened file.

`printf(format, arguments) | "command"`

This is an "output pipe". The data is sent to the input of the specified *command*. The *command* can be specified with a literal string as shown or in a variable or any valid TAWK expression that evaluates to a string containing a command.

Under some operating systems the command is executed asynchronously, that is, it is started up and each print or printf sends additional data to the command. Under DOS the data written to the pipe is saved in a temporary file and the command is not executed until the the pipe is closed or the program ends.

PRINTF EXAMPLES

We placed the output of the following printf statements between bars |like this| so you can see precisely how the resulting output is aligned and how many spaces are added.

String Format Examples

<u>Printf Example</u>	<u>Resulting Output</u>
<code>printf "%s", "Hello"</code>	Hello

printf "%10s", "Hello"	Hello
printf "%-10s", "Hello"	Hello
printf "%3s", "Hello"	Hello
printf "%.3s", "Hello"	Hel
printf "%10.3s", "Hello"	Hello
printf "%-10.3s", "Hello"	Hel
printf "%c", "Hello"	H
printf "%c", 65	A

Integer Number Format Examples

<u>Printf Example</u>	<u>Resulting Output</u>
printf "%d", 41	41
printf "%7d", 41	41
printf "%07d", 41	0000041
printf "%-7d", 41	41
printf "%.3d", 41	041
printf "%7.3d", 41	041
printf "% d", 41	41
printf "%+d", 41	+41
printf "% d", -41	-41

Alternate Number Base Format Examples

Note that these formats interpret the argument as a 32 bit unsigned quantity.

<u>Printf Example</u>	<u>Resulting Output</u>
printf "hex: %x", 26	hex: 1a
printf "binary: %b", 41	binary: 101001
printf "octal: %o", 41	octal: 51
printf "unsigned: %u", 41	unsigned: 41
printf "unsigned: %u", -41	unsigned: 4294967255

Floating Point Number Format Examples

<u>Printf Example</u>	<u>Resulting Output</u>
printf "%e", 41.5	4.150000e+01
printf "%f", 41.5	41.500000
printf "%g", 41.5	41.5
printf "%ze", 41.5	4.15e+01
printf "%.5e", 41.5	4.15000e+01
printf "%.5f", 41.5	41.50000
printf "%10.5f", 41.5	41.50000
printf "%g", 123456	123456
printf "%g", 1234567	1.23457e+06
printf "%.3g", 1234567	1.23e+06
printf "%g", .000123	0.000123
printf "%g", .0000123	1.23e-05
printf "%m", 41.5	41.50
printf "%m", -41.5	(41.50)

rand()

The rand() function returns a pseudo-random number in the range $0 \leq r < 1$. The srand() function can be used to set the seed for the random number generator. If srand() is not called the pseudo-random number sequence returned by rand() is the same for each invocation of the program.

regex(string)

or

regex(string,flags)

The *string* is pre-compiled into a regular expression pattern. The *flags* argument may be "i" for ignore-case, "s" for shortest match, or "is" for both. The resulting return value can be used anywhere a regular expression is expected: as a function argument (for example with the split, sub, or match functions) or with the ~ or !~ operators.

Why would you use this function? In TAWK, regular expression matching occurs in two phases: first the regular expression is compiled into an internal state machine, then the string to be matched is examined using that state machine. Compiling the state machines is slow; examining strings with them is lightning fast. The regex() function compiles the regular expression so that multiple compilations can be avoided. Note that if you know what the regular expression is at compile time you can specify the regular expression /like this/ which also tells TAWK to compile the regular expression just once.

Examples:

```
BEGIN {
  # The "i" means case-insensitive matching:
  x = regex("this is a pattern", "i")
  if (match(y,x)) print "it matched"
  if (y ~ x) print "matched again"
}
```

To use the return value of the regex() function in the pattern part of a pattern-action block you should use the ~ operator as in the following example:

```
BEGIN { re = regex("this is a pattern") }

$0 ~ re { print "current record matches!" }
```

registercallback(funname)

[Win32 version only]

The *funname* parameter is the name of a TAWK function, as a string. This function returns a function pointer that can be passed to a Win32 function as a call-back function pointer, or 0 if it fails. When Win32 calls the callback function, your TAWK function will be called. The TAWK function named by *funname* should have exactly four parameters. There are only a limited number of callback functions available, so when the callback function is no longer needed, you should release it using the unregistercallback function.

See also: the discussion of Dynamic Linking in the chapter on Calling External Functions, and the unregistercallback function.

rename(file1,file2)

This function renames *file1* to *file2*. This function can rename either files or directories. The filenames may not contain wildcard matching characters. The restrictions of the underlying operating system apply: files may be renamed or moved within a disk drive and partition but can not be moved to a different disk drive or partition. Under DOS and OS/2 directories may be renamed but not moved to a new location in the directory tree. Returns 1 (TRUE) if it succeeds, or 0 (FALSE) if it fails.

rindex(*string1*,*string2*)

or

rindex(*string1*,*string2*,*end*)

This function returns the largest index where *string2* occurs in *string1*, or 0 if *string2* is not found in *string1*. If *end* is specified it is the index of the last character of *string1* that will be considered in the match. For example:

```
rindex("abcbc","bc")      # returns 4  rindex("abcbc","bc",4)  # returns 2
```

See also: index function.

rmdir(*dirname*)

This function removes directory *dirname*. The directory must be empty of files. Returns 1 (TRUE) if it succeeds, or 0 (FALSE) if it fails. Example:

```
rmdir("C:\\usr\\patt")
```

or:

```
rmdir("/usr/patt")  # Works under both DOS and UNIX
```

rmfile(*filename*)

This function removes file: *filename*. The *filename* may not contain wildcard matching characters. Returns 1 (TRUE) if it succeeds, or 0 (FALSE) if it fails.

```
rmfile("C:\\usr\\patt\\profile.sh")
```

[UNIX Version]

Under UNIX, a single physical file may have more than one filename. Each filename is called a link to the actual physical file. If there is more than one link to a file, the actual physical file will not be removed until every link to it is removed.

Screen I/O Functions:

The following functions (with names beginning with scr_) directly manipulate the display screen. The row and column positions are numbered starting at 0. For example, scr_scp(0,0) moves the cursor to the upper left corner of the screen.

Under DOS, high speed is obtained by using the direct video interrupt (number 16), thus bypassing the ANSI.SYS driver, if any. When running under DOS, these functions require an IBM compatible computer and might crash non-IBM compatible computers. (But almost all PC computers are IBM compatible these days.)

Under UNIX, the first call to any of these functions activates the "curses" function library. This has many side-effects, including clearing the screen, and placing the keyboard in "cbreak" and "noecho" mode. Curses mode is automatically terminated before a call to getline, fread, system, spawn or scr_end, or when the program ends.

scr_end()

[UNIX Version]

Under UNIX, this function terminates the "curses" function library and returns the terminal to normal mode. TAWK automatically terminates curses before getline, fread, system or spawn functions, or when the program ends, so you should normally not need to call this function.

[Other Versions]

This function has no effect.

scr_frame(*title*,*x1*,*y1*,*x2*,*y2*)

or

scr_frame(*title*,*x1*,*y1*,*x2*,*y2*,*color*)

Draws a box on the screen. This is typically used to draw window frames on character mode displays. The edges and corners of the box are drawn using line drawing characters, if available on the terminal, otherwise + - and | characters. *Title* is the window title, which is centered in the upper edge of the box. The upper left coordinates are given by *x1*,*y1*, and the lower right coordinates by *x2*,*y2*. All coordinates start at (0,0) at the upper left corner of the screen. The optional *color* argument is the color of the box; see the *scr_put* function for a list of colors.

[UNIX Version]

TAWK attempts to obtain appropriate line drawing characters from the "curses" database. This does not always work, even if the terminal supports line drawing characters.

scr_gcm(*scan1*,*scan2*)

[DOS, OS/2 Versions]

The *scan1* and *scan2* arguments should be variables. The current cursor starting and ending scan lines are returned in the variables *scan1* and *scan2*. The starting and ending scan lines indicate the height of the cursor in scan lines. The number of scan lines per screen character varies for different video display systems.

[Win32 Version]

The *scan1* argument, which should be a variable, receives the current cursor height as a percentage between 0 and 100.

[UNIX Version]

The *scan1* argument, which should be a variable, receives the current cursor visibility, as inferred by the "curses" library. The value returned is 0, 1 or 2.

scr_gcp(*row*,*column*)

The *row* and *column* arguments should be variables. The current screen cursor position is returned in the variables *row* and *column*.

scr_get(*len*,*row*,*column*)

Returns the text string of length: *len* from the screen at position *row* and *column*. Only the text at that position is returned; the attribute bytes are ignored.

scr_getcells(*len*,*row*,*col*)

Returns the cell string of length: *len* from the screen at position *row* and *column*. The cell *string* consists of pairs of bytes where the first byte is the character and the second is the attribute.

scr_put(*string*,*row*,*column*)

or

scr_put(*string*,*row*,*column*, *attr*)

Places the text *string* on the screen at the position specified by *row* and *column*. If *attr* is specified, it is the attribute byte of the characters written to the screen, otherwise the current screen attributes are left alone. The attribute byte has different meanings for different displays, but for most color displays the bottom four bits are the foreground color, the next three bits are the background color (which restricts the background color to colors 0 to 7 below), and the upper bit, if set, usually causes the character to stand-out in some way, such as blinking. The colors are defined as follows:

0	black	8	gray
---	-------	---	------

1	blue	9	light blue
2	green	10	light green
3	cyan	11	light cyan
4	red	12	light red
5	magenta	13	light magenta
6	brown	14	light yellow
7	white	15	high-intensity white

[UNIX Version]

TAWK attempts to set the colors using the "curses" database. This is notoriously unreliable. On monochrome terminals the colors are ignored except: an attribute of 0x70 (black-on-white) should produce reverse video; if the high bit (0x80) is set, the terminal's stand-out mode is selected, if possible.

scr_putcells(*string*,*row*,*column*)

Puts the cell *string* to the screen at position *row* and *column*. The cell *string* consists of pairs of bytes where the first byte is the character and the second is the attribute.

scr_refresh()

or

scr_refresh(*flag*)

[UNIX Version]

This function controls automatic screen refresh mode. By default, automatic refresh mode is ON. If the optional *flag* argument is 0, automatic refresh is turned OFF. If the optional *flag* argument is 1, the screen is immediately refreshed, and automatic refresh mode is turned ON. If automatic refresh mode is on, the screen is refreshed after every *scr_* function. If automatic refresh mode is off, your program must refresh the screen, when desired, by calling *scr_refresh*. If no arguments are supplied to *scr_refresh*, the screen is refreshed immediately, but the state of the automatic refresh mode is left unchanged. This function is provided because screen refresh using the "curses" library (used by TAWK) is extremely slow on some terminals, and it may be quicker to buffer up many changes to the screen and send them all at once.

[Other Versions]

This function has no effect.

scr_scm(*scan1*,*scan2*)

[DOS, OS/2 Versions]

Sets the cursor starting and ending scan lines. This function can be used to set the height of the cursor.

[Win32 Version]

Scan1 is the desired height of the cursor as a percentage from 0 to 99. For example, *scr_scm*(0) removes the cursor, and *scr_scm*(99) makes it the full height of the character cell.

[UNIX Version]

TAWK attempts to set the cursor visibility using the curses library *curs_set*() function. The *scan1* argument should be 0, 1 or 2 to select among different cursor types.

See also: *scr_gcm*().

scr_scp(*row*,*column*)

Sets the screen cursor position the specified *row* and *column*.

setawkvar(variable_name,value)

or

setawkvar(variable_name,value,flag)

Sets the value of the specified TAWK variable to the specified *value*. If the optional third *flag* argument is 1 and the specified value is a string then it is treated the same way TAWK treats input fields, that is, the string is examined and treated as a number if the string contains only a valid number.

Only variables that are declared global and that are not arrays can be set by setawkvar. An exception: if your program consists of a single source file then all variables are automatically considered global.

The following example prints "17".

```
global x
BEGIN {
    setawkvar("x",17)
    print x
}
```

shiffl(x1,x2)**shiftr(x1,x2)**

Shiffl does a left shift and shiftr does an arithmetic right shift. *X1* is the number to shift, and *x2* is the number of bit positions to shift. *X1* is coerced to a 32 bit integer. The arithmetic right shift performs sign extension, that is, the top-most bit (which is the sign bit) is duplicated in each bit position that is shifted in. See also: and(). Examples:

```
BEGIN {
    printf("0x%x\n", shiffl(4,1))      # prints 0x8
    printf("0x%x\n", shiftr(4,1))     # prints 0x2

    # The shiftr function performs sign extension:
    # The following prints 0xffff0000
    printf("0x%x\n", shiftr(0x80000000,15))
}
```

sin(x)

Sine of *x*. *X* is in radians. See also: cos().

sleep(seconds)

This function delays execution for a specified number of *seconds*, which may be expressed as an integer or as a floating point number. Under DOS, this function is implemented as a busy loop with accuracy only to within the accuracy of the IBM-PC clock, which is about 1/18 second. Under other operating systems, this function yields control to the operating system so that other programs can run. The delay may be longer than that specified if the operating system does not return control to the TAWK program promptly after the sleep time has expired. Example:

```
sleep(1.5)  # Waits 1 1/2 seconds
```

spawn(*command*)
 or
spawn(*command,environment*)
 or
spawn(*command,environment,flags*)

This function runs the *command* and returns the exit status of the *command*, or -1 if the *command* could not be run. The *command* typically includes a program name and the program arguments separated by spaces. The program may not run if it can not be found, if there is not enough memory, or if too many files are open. If the program is specified without a path, then the program is searched for using the PATH, as specified in the ENVIRON array, or in the *environment* array. If the optional second *environment* argument is specified, it is an array containing the environment strings to be passed to the program, and defaults to the ENVIRON array.

If the optional third *flags* argument has the value 1, the *command* is executed asynchronously, if this is possible in the operating system, and spawn returns the process id of the child process rather than the *command* exit status.

Under DOS, Win32 or OS/2, the spawn() function differs from the system() function in that it executes the program directly rather than starting a copy of the command interpreter to do it. It is therefore faster, uses less memory and returns the true exit status of the program under DOS. However, note that batch files and built-in functions (like the "dir" command in DOS) are executed directly by the command interpreter and so must be run using the system() function rather than spawn().

[DOS Version]

The *flags* argument is ignored, as there is no multi-tasking under DOS. In the 16-bit DOS version, there is a maximum limit on the size of the environment passed to the called program. See the SYSSWAP variable for more information.

[OS2 Version]

OS2 presentation manager programs (these are programs that do graphics under OS2) can not be run with spawn: use system() instead.

[UNIX Version]

The shell specified in the "SHELL" environment variable is invoked to execute the *command*.

[Other Version Differences]

The DOS, OS/2 and Win32 versions look in the current directory for the program before searching the PATH. UNIX searches using the PATH variable only, and does not check the current directory unless the PATH contains a "." entry.

See also: system(), ENVIRON, SYSSWAP.

split(*string,array_name*)
 or
split(*string,array_name,fs*)

This function splits the *string* into fields. The *array_name* argument is the name of a variable that receives an array containing the fields found in *string*. The first field is returned in array_name[1], the second in array_name[2], etc.

The *fs* argument indicates the field separator to be used to break up the fields. If the *fs* argument is not specified the current value of the FS variable is used. The way the fields are determined depends on the value of the *fs* argument (or FS variable) as follows:

FS is a space: " "

This is a special case that indicates that the fields are delimited by any run of white space characters. Additionally, any leading or trailing white space characters in *string* are ignored. If you want to specify that the field separator is really just a single space use: // or "[]"

FS is any other single character

That single character is the field separator. If the field separator character appears at the beginning or end of *string* or if two of the field separator characters are adjacent in *string* it indicates an empty field at that position.

FS is an empty string: ""

This is a special case that causes the *string* to be broken up into individual characters. Thus the returned array contains one of the characters of *string* in each array element.

FS is a multi-character string:

The string is interpreted as a regular expression pattern. The fields in *string* are separated by non-zero non-overlapping runs of characters that match the specified regular expression.

FS is a regular expression pattern:

The regular expression pattern specifies the field separator. The fields in *string* are separated by non-zero non-overlapping runs of characters that match the specified regular expression.

The return value of the split function is the number of fields found, which is also the number of elements in the returned array.

Examples:

```
BEGIN {
  split(" a b c",tab," ")
  print tab[1]    # Prints: a
  print tab[2]    # Prints: b
  print tab[3]    # Prints: c

  split("a||b",tab,"|")
  print tab[1]    # Prints: a
  print tab[2]    # Prints nothing:
  print tab[3]    # Prints: b

  # To set the separator to a real space, use this:
  # (Note there are two spaces between
  # the a and the b)
  split("a  b",tab,/ /)
  print tab[1]    # Prints: a
  print tab[2]    # Prints nothing:
  print tab[3]    # Prints: b
}
```

See also: splitp(), and paste(), which is the opposite of split().

splitp(*string*,*array_name*)

or

splitp(*string*,*array_name*,*fp*)

This function splits the *string* into fields. The *array_name* argument is the name of a variable that receives an array containing the fields found in *string*. The first field is returned in *array_name*[1], the second in *array_name*[2], etc.

The *fp* argument indicates the pattern specifying what is allowed in a field. If the *fp* argument is not specified the current value of the FPAT variable is used. The *fp* argument can be specified as a regular expression or as a string that will be interpreted as a regular expression pattern.

The return value of the splitp function is the number of fields found, which is also the number of elements in the returned array.

Note that this function is identical to the split() function, except that it differentiates the fields in the *string* based on the pattern that matches the fields themselves rather than the pattern that matches the separators between the fields.

Example:

The splitp function is particularly useful for tokenizing program files. In these cases the presence or absence of space between tokens is irrelevant and all that matters is the pattern that specifies the tokens to be recognized. For example, the following program tokenizes a simple language consisting of alpha-numeric identifiers, +, -, *, / and = :

```

BEGIN {
    str = "a = b+c"    # String to tokenize
    fpat = /[a-zA-Z0-9]+|[-+*/=]+/
    splitp(str,x,fpat)
    for (i in x) print "." x[i] "."
}

```

The above program tokenizes the string "a = b+c" and prints:

```

.a.
.=.
.b.
.+.
.c.

```

Note that anything that is not matched by `fp` is thrown away. For example, the above program when given the string: `str = "a = b%c"` will throw away the `"%"` character because it is not included in the `fp` pattern and is therefore considered part of the separator between the fields. When using this function make sure you include every pattern you need to match in `fp`.

See also: `split()`, `FPAT`.

sprintf(*format*)

or

sprintf(*format,value,...*)

This function formats the specified list of zero or more *values* using the *format* string and returns the resulting string. The *format* string is the same as for the `printf()` function. See `printf()` for more information.

sqrt(*x*)

Returns the square-root of *x*.

srand()

or

srand(*seed*)

This function sets the *seed* for the random number generator and is used to generate a repeatable sequence of pseudo-random numbers. If no *seed* is specified, `srand()` generates a new semi-random seed using the time of day. Use the `rand()` function to obtain pseudo-random numbers. If `srand()` is not called the random number sequence returned by `rand()` is the same for each invocation of the program.

stat(*filename*)

or

stat(*filename,info*)

This function determines if a file exists, and optionally returns information about *filename* in the *info* array. The *filename* may be a file or directory name, or an open file descriptor, such as returned by the `open` function. The `stat()` function returns TRUE (1) if the file or directory exists, or FALSE (0) if it does not.

Note: If the *filename* is the UNC filename for the root directory of a remote system, for example: *//machinename/resource/* you must make sure to include the final slash, or `stat` may return FALSE even though the directory exists.

The *info* argument, if specified, should be a variable, which is returned as an array containing the following elements:

"SIZE"	File size, in bytes.
"MODE"	File mode, see below.

"MTIME"	Time file was last modified.
"ATIME"	Time file was last accessed, if known. If not, this field may contain garbage.
"CTIME"	Time file was created, if known. If not, this field may contain garbage.
"NLINK"	Number of links to file. [UNIX only]
"DEV"	Internal information about the file. [UNIX only]
"RDEV"	More internal information about the file. [UNIX only]
"UID"	File owner Identifier. [UNIX only]
"GID"	Group Identifier. [UNIX only]
"INODE"	Inode number. [UNIX only]

The file mode, as provided by info["MODE"], is a bit field. The low order 16 bits of the mode contain the UNIX compatible mode bits for the file. The meaning of the individual bits are as follows:

UNIX Compatible File Modes

0x100	read permission for file owner
0x80	write permission for file owner
0x40	execute permission for file owner, or search permission for directory owner
0x20	read permission for file's group
0x10	write permission for file's group
0x8	execute permission for file's group, or search permission for directory's group
0x4	read permission for others
0x2	write permission for others
0x1	execute permission for others, or search permission for directory by others

[UNIX Version]

The other mode bits are used internally by UNIX. If you wish to decipher their meaning, you will need documentation for your particular UNIX implementation.

[DOS, Win32 and OS/2 Versions Only]

All files always have read permission for owner, group and other. If the file has write permission (i.e., the file is read-only), all three write permissions for owner, group and other will be set. In addition, the PC-specific file attributes are returned in the high order 16-bits of the mode. These bits may be set as follows:

0x10000	read-only attribute
0x20000	hidden attribute

0x40000	system attribute
0x100000	directory attribute
0x200000	archive attribute

strdup(*string*) or **strdup(*string*,*ncopies*)**

If there is only one argument this function allocates and returns a new copy of *string*. This is normally unnecessary but is sometimes useful when interfacing to C routines to make sure that you have a new copy of string. If the *ncopies* argument is specified strdup returns a new string containing that many adjacent copies of the *string*. Examples:

```
strdup("a",1)      # Returns "a"
strdup("ab",3)     # Returns "ababab"
```

Substitution Functions: **sub**, **subs**, **gsub**, **gsubs**

sub(*pattern*,*replacement*) or **sub(*pattern*,*replacement*,*variable_name*)** or **sub(*pattern*,*replacement*,*variable_name*,*flag*)**

This is a family of four functions that perform substitutions in a string. The mnemonic for the functions names are:

<u>Function</u>	<u>What it does</u>
sub	substitute for regular expression
subs	substitute for string
gsub	global substitute for regular expression
gsubs	global substitute for string

The syntax of all four functions is identical to the **sub** function shown. All four functions substitute occurrences of the specified *pattern* with the specified replacement *string*. The **sub** and **gsub** functions interpret the *pattern* as a regular expression. The **subs** and **gsubs** functions interpret the *pattern* as a plain string to look for.

The **sub** and **subs** functions substitute only the first occurrence of the *pattern* found. The **gsub** and **gsubs** functions substitute all non-overlapping occurrences of the *pattern* found.

If *variable_name* is specified, it should be a variable that contains a string; the substitutions are made on the string contained in the variable. The substitutions occur IN PLACE, that is, the contents of the variable are changed. This is a technique called pass-by-reference and is used in only a few places in TAWK. If a *variable_name* is not specified the substitutions are made in the current record (\$0) instead.

The optional *flag* argument controls interpretation of the *replacement* string. If the *flag* is 0, the *replacement* string is used as is, without any interpretation. If the *flag* is 1 (the default) the *replacement* string undergoes interpretation as follows: each occurrence of an ampersand (&) character in the *replacement* string is replaced by whatever part of the string was matched by the pattern. For the **sub** and **gsub** functions only: each occurrence in the *replacement* string of \$n, where n is a digit, is replaced by whatever part of the string was matched by the nth set of parentheses in the pattern. \$0 means the same as &. Both & and \$ can be preceded by a backslash to insert a normal & or \$ in the replacement string.

Return value: These functions return the number of substitutions made.

Examples:

```
BEGIN {
  x = "masasapa"
  sub(/a/, "i", x)
  print x           # Prints "misasapa"

  gsub(/a/, "i", x)
  print x           # Prints "misisipi"

  gsub(/i(.)/, "i$1$1", x)
  print x           # Prints "mississippi"

  gsub(/i(.)/, "i$1", x, 0)
  print x           # Prints "mi$1si$1si$1pi"

  gsubs("$1", "s", x)
  print x           # Prints "mississispi"
}
```

For the sub and gsub functions, the regular expression can be given as a literal regular expression /like this/ or in a string "like this". If the regular expression is specified as a literal string, remember that the string will undergo two levels of backslash interpretation: one when processing the string and another when processing the regular expression. For example:

```
BEGIN {
  path = "C:\\usr\\patt"
  gsub(/\\/, "/" . x)
  print x           # Prints "C:/usr/patt"

  path = "C:\\usr\\patt"
  gsub("\\\\", "/" . x)
  print x           # Prints "C:/usr/patt"
}
```

substr(*string*,*startpos*)

or

substr(*string*,*startpos*,*length*)

When used as a function substr() returns the sub-string of the specified *string* starting at position: *startpos*. If *length* is specified it specifies the maximum length of the returned string. Examples:

```
BEGIN {
  print substr("abcdef", 3)      # Prints: cdef
  print substr("abcdef", 3, 2)   # Prints: cd
}
```

substr(*variable_name*,*startpos*) = replacement

or

substr(*variable_name*,*startpos*,*length*) = replacement

In TAWK you can use the substr function on the left hand side of an assignment. The *variable_name* argument is a variable that contains a string that you want to modify. This use of substr replaces the substring of the string contained in the variable starting at *startpos* and continuing *length* characters with the *replacement* on the right hand side of the assignment. The length of replacement string does not have to be the same as the substring that is being replaced. In fact, the length specified to substr can be 0 to indicate that the string is to be inserted at the specified position. If *length* is not specified it defaults to replacing the entire string from *startpos* on. For example:

```

BEGIN {
  x = "abcd"
  substr(x,2,2) = "xyz"    # Replace "bc" with "xyz"
  print x                 # Prints: axyzd

  substr(x,2,3) = ""      # Removes "xyz"
  print x                 # Prints: ad

  substr(x,1,0) = "hello" # Inserts "hello"
  print x                 # prints: helloworld
}

```

This form of `substr` is commonly used in conjunction with the `match` statement, which sets the `RSTART` and `RLENGTH` variables to the start and length of the substring that was matched. For example:

```

BEGIN {
  x = "abcd"
  if (match(x,"bc")) {
    substr(x,RSTART,RLENGTH) = "this replaces bc"  }
}

```

Beware the following potential problem:

```

BEGIN {
  while (match(x,"bc")) {
    substr(x,RSTART,RLENGTH) = "this replaces bc"
  }
}

```

This code will run forever because the replacement string also contains "bc", which will also be matched and replaced, and so on forever. Use the `gsub` function in this case, or specify a starting index for the `match` function and make sure that you always advance the starting index beyond the end of the replacement string.

system(*command*)

The `system()` function runs the specified *command*, which typically consists of a program and program arguments separated by spaces. The program is run using the operating system's command interpreter. This allows the *command* to be any type of executable program including internal commands like "dir" (DOS) or "set" (UNIX) or a batch file or shell script. This is different from the `spawn()` command, which invokes a program directly without using the command interpreter. If no error occurs, the `system()` function returns the exit status of the command interpreter after executing the program. (Except under DOS: see below.)

The path of the the command interpreter used by `system()` is specified by an environment variable in TAWK's `ENVIRON` array. Under DOS, OS/2 and Win32 it uses the "COMSPEC" environment variable, and under UNIX it uses "SHELL". This environment variable is usually pre-set to the path of the command interpreter by the operating system, so you don't have to worry about it. You can cause the `system()` function to use an alternate command interpreter simply by changing this environment variable in TAWK's `ENVIRON` array. For example, under DOS or OS/2 you can specify the Thompson Toolkit Shell as the command interpreter by setting:

```

BEGIN {  ENVIRON["COMSPEC"] = "C:/usr/bin/sh.exe" }

```

If the command interpreter program specified in the `ENVIRON` array can not be found or fails for some reason, there may be no warning message printed, but the `system()` function will return an error code of -1.

[DOS Version]

The exit status of `command.com` is always 0! So the return value of `system()` does not reflect the exit code of the executed command. There is also no way to detect if the program invoked by the `system()` function was abnormally terminated by a ^C interrupt. If you need reliable exit codes, either use the `spawn()` function or use a better command interpreter like the Thompson Toolkit Shell.

Under DOS there is a maximum limit on the size of the environment passed to the called program. See the `SYSSWAP` variable for more information.

Warning: Microsoft has indicated that the `system()` function may not execute properly under MS-DOS version 2. We tested under MS-DOS version 2 and had no problems, but be warned.

For more information see: `ENVIRON` array, `SYSSWAP` variable.

`table(index1,value1, ...)`

The `table` function creates a TAWK table (or array) from a list of pairs of indices and values. The first argument is the first index, the second argument is the first value, etc. The number of arguments to the `table` function must be even. For example: the following code:

```
BEGIN { x["a"] = "this"; x["b"] = "those" }
```

is identical to:

```
BEGIN { x = table("a","this","b","those"); }
```

To create an array, every other argument to the `table` function would be the integer array number, for example:

```
BEGIN { x = table(1,"this",2,"those"); }
```

`time()`

or

`time(year,month,day,hour,minute,second)`

`Time()` without arguments returns the current time as an integer with resolution in seconds. Up to six optional arguments may be given to convert a specified date and time to an integer. Not all arguments need be given; unspecified arguments default to the lowest value. A warning message is printed if the year or month is out of range; no checks are made on the other arguments. The ranges of the optional arguments are:

Element	Range
year	1980 - 2100
month	1 - 12
day	1 - 31
hour	0 - 23 (0 means between midnight and 1 am)
minute	0 - 59
second	0 - 59

Warning: If you are comparing the return values of the `time()` and/or `filetime()` functions to determine which of two times is more recent do not go beyond the year 2038. The integer values returned for dates beyond year 2038 are actually negative numbers so they do not compare properly with dates before year 2038.

`timetab(table)`

or

`timetab(table,timeval)`

`Timetab()` returns the current date and time or, if the *timeval* argument is specified, the date and time indicated by *timeval*. If *timeval* is specified it is a time number that was returned by the `time()` or `filetime()` functions. The date and time are returned in the *table* variable, which is converted to an array whose elements are as follows:

Element	Range
"YEAR"	1980 - 2100
"MONTH"	1 - 12
"DAY"	1 - 31
"HOUR"	0 - 23 (0 means between midnight and 1 am)
"MIN"	0 - 59
"SEC"	0 - 59
"WEEKDAY"	1-7 (1 is Sunday, 2 is Monday, ...)
"YEARDAY"	1 - 366 (Ordinal day of the year)

The following example prints when the file `"/config.sys"` was last modified:

```
BEGIN {
    when = filetime("/config.sys")
    timetab(x,when)
    printf("config.sys was modified on: %d:%d:%d\n",
        x["MONTH"],x["DAY"],x["YEAR"])
}
```

See also: `ctime()`.

toupper(*string*) and **tolower(*string*)**

These functions return a string that is the same as the argument *string* but with characters converted to upper case or lower case. For example:

```
BEGIN {
    x = tolower("ABC")
    print x          # Prints: abc
}
```

translate(*string*,*searchlist*,*replacementlist*) or **translate(*string*,*searchlist*,*replacementlist*,*flags*)**

This function returns a string that is the same as the *string* argument but that is translated by replacing all occurrences of the characters specified in the *searchlist* with the corresponding characters in the *replacementlist*. The first character in the *searchlist* is translated to the first character in the *replacementlist*, the second to the second, etc. If the *replacementlist* is an empty string the characters in *searchlist* are just deleted. If the *replacementlist* is shorter than the *searchlist* the last character is replicated as necessary, unless "d" is included in the optional *flags* argument. Both *searchlist* and *replacementlist* can include ranges of the form: "a-z", which means all characters from "a" to "z", inclusive. Character ranges can be reversed, for example, "a-d" means "abcd", and "d-a" means "dcba". To include a dash character ("-") in the list it must be the first or last character in the list, or the "n" option must be specified. The optional *flags* argument is a string of characters that are options with meaning as follows:

- "d" Delete: If *searchlist* is longer than *replacementlist* then characters in *searchlist* that do not have a translation specified by *replacementlist* are deleted. Normally these characters would be translated to the last character of the replacement list.
- "s" Squeeze: Translations that would result in adjacent identical characters from *replacementlist* in the output string are replaced by (squeezed into) a single instance of that character. Note that only characters that are translated are squeezed. Characters that do not appear in *searchlist* are left alone.

- "c" Complement: The searchlist used by translate is the complement of the specified searchlist. That is, the searchlist used by translate is composed of all characters that do not appear in the searchlist.
- "n" This option suppresses interpretation of character ranges in the searchlist and replacementlist, so for example, "a-c" would mean "a-c" instead of "abc".

Examples:

```
BEGIN {
  # Convert str to upper case: This is like
  # the toupper() function but it can be
  # modified to handle non-English languages.
  result = translate(str,"a-z","A-Z")

  # Reset the high bit of each character in str:
  # (This is useful for WordPerfect files)
  result = translate(str,"\x80-\xff","\x0-\x7f")

  # Change all white space characters to spaces.
  result = translate(str," \t\r\n"," ")

  # Use the squeeze option to change all sequences
  # of one or more white space characters to a
  # single space:
  result = translate(str," \t\r\n"," ","s")

  # Use the complement option to change all
  # non-alphanumeric characters into spaces:
  result = translate(str,"a-zA-Z0-9"," ","c")

  # Delete all non-alphanumeric characters:
  # (The "d" option could be included but is not
  # necessary in this particular case because
  # replacementlist is empty.)
  x = translate(str,"a-zA-Z0-9","","c")
}
```

typeof(x)

Returns a string corresponding to the type of x. The possible return values are:

"int"	integer;
"float"	floating point number;
"string"	string;
"array"	any type of array or table;
"regex"	a regular expression pattern;
"fileid"	a file descriptor from the fopen() function;
"uninitialized"	an uninitialized variable; (This variable will act like a 0, or "", depending on use.)
"address"	the result of the addressof() function;
"unknown"	any other type.

unpack(template,string,array)

The unpack() function assumes that the *string* has a binary structure described by the *template* string and expands the individual elements of the structure into an *array* of values. The *array* argument must be the name of a variable in which the resulting array is returned. The unpack() function does the opposite of pack(). The same *template* string can be used to both the pack() and unpack() functions. These two functions together can be used to read/write to binary data-bases or to pack/unpack structures to be passed to/from C programs.

The template string consists of one or more binary format specifications, separated by spaces. Each format specification looks like the following, and must appear all together without any intervening spaces. Square brackets: [] indicate optional items:

[*name*] @ [*flags*] [*count*] *type*

The *name* is the optional field name, the "@" sign is required, the *flags* and *count* are optional, and the *type* is a required character that indicates the binary type of the field.

If the optional *name* is specified, the unpack function uses it as the index of the array element created to hold that field. Fields without names are returned in array elements with ascending numeric indicies, i.e., the first is returned in array[1], the second in array[2], etc.

The format *type* character is chosen from the following list:

Pack/Unpack Format Characters

a	ASCII string, nul (zero) padded
A	ASCII string, space padded
b	signed byte
B	unsigned byte
c	one character string
s	signed short (16-bit) integer
S	unsigned short (16-bit) integer
l	signed long (32-bit) integer
L	unsigned long (32-bit) integer
f	single precision floating point number
d	double precision floating point number
x	move forward one byte, i.e., skip the next byte
X	move backward one byte
p	32-bit pointer to string, which must be 0 (NULL) or must point to a valid string. If the pointer is 0, a NULL value (also called an uninitialized value) is stored in the corresponding array element. If the pointer is non-0, the nul-terminated string pointed to by this 32-bit pointer is copied into TAWK's memory as a string, which is stored in the corresponding array element. Note that your TAWK program may crash if the 32-bit pointer is invalid.

NOTE: When you unpack a floating point number, you better make sure the binary string really contains a valid floating point number or you may get an error message, either when it is unpacked or later when you try to use the number.

The "*type*" character may optionally be preceded by a number that indicates the number of characters in the field for types "a" and "A", or a field count of the number of values to be consumed from the list of values for any other type. If no count is specified for types "a" or "A", unpack consumes characters from the string until the terminator character (either nul for "a" or space for "A") is found.

The "a" and "A" types may also optionally be preceded by the following flags:

+	Value is right justified. (Default is left justified.) A field length must be specified with this flag.
&	Prevents stripping spaces or nuls from the field;

The "s", "S", "l", "L", "d" and "f" types may optionally be preceded by the following *flags* to control the byte ordering and packing. By default, the byte ordering is whatever the native ordering is on the host processor, and the items are packed together as closely as possible.

<u>Flag</u>	<u>Meaning</u>
>	Pack bytes most significant byte first (default for most RISC processors.)
<	Pack bytes least significant byte first (default for 8x86 type processors.)
#	Insert padding before the field so that it is aligned on an N byte boundary, where N is the length of the item in bytes. This type of structure packing is used by many compilers.

In addition, the <, >, and # flags may appear in the template string outside of a format specification, in which case they apply to all format specifications that follow them. For example: "@<S @<S" and "<@S @S" are equivalent.

See the pack() function for examples using unpack().

unregistercallback(*funname*)

[Win32 version only]

The *funname* parameter is the name of a TAWK function. You can only use a limited number of TAWK functions (currently, three) as callback functions simultaneously. This function tells TAWK that the specified TAWK function is no longer being used as a call-back function.

See the discussion of Dynamic Linking in the chapter on Calling External Functions, and the registercallback function.

xor(*x1,x2*)

See and().

Chapter 18: TAWK Built-In Variables

ARGC

This is the number of program arguments plus one. The "plus one" is for the program name that is also contained in the ARGV array. For example, if there are no program arguments, `ARGC = 1`; if there is one program argument, `ARGC = 2`; etc.

When TAWK is looking for a program argument to be processed by the Automatic Input Loop, it looks only as far as `ARGV[ARGC-1]`. If you add new elements to the ARGV array that you want the Automatic Input Loop to process as program arguments, you must also increase the value of ARGC appropriately.

See also: ARGV, and the chapter on Program Arguments.

ARGI

This is the index in the ARGV array of the next program argument that will be processed by TAWK's Automatic Input Loop. ARGI is initialized to 1, so the first argument that TAWK processes is `ARGV[1]`. You can change ARGI if you wish and TAWK will use the new value as the index of the next program argument to process.

See also: ARGC, ARGV, and the chapter on Program Arguments.

ARGV

This variable is an array that contains the program name and arguments. The program name is in `ARGV[0]` and the program arguments, if any, are in `ARGV[1]`, `ARGV[2]`, etc.

Normally the program arguments are separated by spaces. To include a space in a program argument, surround the argument with quotes on the command line "like this".

Under DOS, the `ARGV[0]` element contains the full pathname of the executable program. Under other operating systems or in a bound program (that runs under both DOS or OS2) `ARGV[0]` usually contains the actual path, if any, used to invoke the program. If the program was found in the current directory, the directory part of the path may be missing. If the program was entered interactively, `ARGV[0]` contains just "awk".

See also: ARGC, ARGI, and the chapter on Program Arguments.

BUFSIZE

For experts: This variable specifies the size of input/output buffers used internally by TAWK. Sometimes, you can optimize the speed of input/output operations by increasing this size. BUFSIZE is used only when a file is first opened, so you must set it before opening the file. Setting BUFSIZE to a size larger than 65535 has no effect. Under DOS, and particularly if no memory manager (like smartdrv) is in use, the best performance is usually obtained by setting BUFSIZE to the "cluster" size of your disk. Under most other operating systems, input/output is buffered more effectively, and changing BUFSIZE will probably have little effect.

CONVFMT

FOR EXPERTS ONLY! This variable contains the format specification that is used when it is necessary to automatically convert a floating point number into a string anywhere in the program except for an output statement, for example, the `print` or `printf` statements, where OFMT is used instead. The format specification is in the same format as used by the `printf` statement. See the `printf` statement for a list of valid formats. The default format is `"%.6g"`. Note that CONVFMT affects only floating point numbers, not integers.

Changing CONVFMT can change the format of the strings used for array indices if the indices happen to be floating point numbers. This can create surprising results. The following example illustrates this, where the exact same statement can create two different array elements depending on the value of the CONVFMT variable:

```
BEGIN {
  i = 1.5 + 1.5
  x[i] = "this"
  # This creates array element x["3"]
  CONVFMT = "%.2f"
  # This now creates array element x["3.00"]
  x[i] = "this"
}
```

DLLS

This variable is a string containing a space-separated list of filenames that are the Dynamic Link Libraries (DLLs) that TAWK should automatically search to find **extern** functions that were declared without any specific DLL name. This variable is initialized to a list of DLL filenames that contain the most commonly used DLL functions for each operating system supported.

EMSLIMIT - see **XMSLIMIT** below.

ENV

This is the old name for the ENVIRON array. It is set to the same array as the ENVIRON array when the program starts. For new programs use ENVIRON instead of ENV.

ENVIRON

This variable is an array. The environment strings are available in TAWK in the built-in array variable: ENVIRON in the format:

```
ENVIRON["NAME"] = "value".
```

For example, the minimum environment under MS-DOS would be:

```
ENVIRON["COMSPEC"] = "C:\\command.com"
ENVIRON["PATH"] = "C:\\dos;C:\\usr\\bin"
```

Note that the names of the environment variables are case-sensitive; you must specify ENVIRON["PATH"], not ENVIRON["path"]. Remember also that if you need to use a backslash in a string in any TAWK program you must use two backslashes because backslash is the string escape character.

Changes to the environment made by modifying the ENVIRON array are passed to commands invoked by the TAWK system() and spawn() functions.

ERRNO

This variable is of little value but is included for use by experts. ERRNO contains the ANSI compatible error code after an operating system function returns an error. This error code is compatible with the **errno** variable in ANSI-compatible C compilers. This variable is often set whether an error occurs or not, so its value is only valid after an error occurs, for example, when a file can not be opened. Note: prior to TAWK version 5, ERRNO contained the operating system specific error code; this value is now returned in ERRNO_OS.

ERRNO_OS

This variable is of little value but is included for use by experts. ERRNO_OS contains operating system specific error code after an operating system function returns an error. This variable is often set whether an error occurs or not, so its value is only valid after an error occurs, for example, when a file can not be opened.

ERRNO_CRIT

[DOS Version]

After a critical error, this integer will hold the device code in the high 16 bits and the error code in the low 16 bits. Normally, critical errors cause TAWK to abort, so this variable is only useful if you are ignoring critical errors by setting SIGNAL["SIGCRIT"] to "ignore" or "fail".

[Other Versions]

This variable is ignored.

FILEMODE

This variable contains the fopen-compatible mode that TAWK uses to open files that are opened automatically for reading. See the fopen() function for a description of valid modes. The default is FILEMODE = "r", which means the files are opened for reading. To read binary files you could set FILEMODE = "rb". To read shared files you could set FILEMODE = "rdn". The FILEMODE variable has no effect on the standard input. A fatal error will ensue if FILEMODE is not a valid fopen-compatible mode or is set to a write-only mode like "w".

FILENAME

Contains the name of the current input file during the Automatic Input Loop. If TAWK is reading from the standard input FILENAME contains "-". Changing this variable currently has no effect but may have an effect in future versions of TAWK, so it should be avoided.

FLOATMASK

For experts only. This variable can be used to enable/disable some math error messages, including those generated by the floating point math coprocessor. The FLOATMASK variable is an integer bit mask. If a bit is set (1), the corresponding error message is enabled; if it is reset (0), the error message will not appear. The value for FLOATMASK is obtained by adding together the bit values of all the error messages you want to receive.

The default value of FLOATMASK allows all error messages except "denormalized operand", "underflow", and "precision loss". The rationale for this is that the "denormalized operand" and "underflow" errors are normally recoverable, and the "precision loss" error occurs with almost every floating point operation, so these messages are usually of no interest.

FLOATMASK bit meanings.

<u>Error Message</u>	<u>Comment</u>	<u>Bit</u>
invalid operation	Generated by the math processor for operations like infinity times 0.	0x1
denormalized operand	Generated by the math processor when an input argument was "denormal". A denormal floating point number is one with an inexact mantissa, and can arise when a floating point result is too small to be accurately represented.	0x2

divide by zero <i>or</i> modulus by zero	Self explanatory.	0x4
overflow <i>or</i> overflow in conversion of floating point number to integer	May be generated by the math processor when a result is too large to be represented, in which case it is stored as infinity, <i>or</i> when a string being converted to a floating point number has an exponent too large.	0x8
underflow	Generated by the math processor when a result is too small to be represented, ie, the exponent is too small. The result may be "denormal".	0x10
precision loss (default: off)	Generated by the math processor for a rounding error. Most floating point operations entail small rounding errors, so this error message is disabled by default. If enabled, this message will occur often!	0x20
invalid argument	atan2(0,0) or log or sqrt of a negative number.	0x40
argument singularity	log(0)	0x80
loss of significance (default: off)	sin, cos, or tan functions inexact result	0x100
total loss of significance	sin, cos, or tan functions invalid result	0x200
too many significant digits	A string being converted to a number has too many significant digits.	0x400
hex number larger than 8 digits	A string being converted to a number contains a hexadecimal number with too many digits. Only 8 hex digits are allowed for a 32 bit integer.	0x800

Examples: To assign a new value to FLOATMASK, for example, to enable all error messages except "precision loss", you could use the following. The number 0xffdf is the hex representation of a number with all bits set except bit 0x20.

```
BEGIN { FLOATMASK = 0xffdf }
```

To change a single bit in FLOATMASK, use the or() and and() functions. For example, to turn off the "hex number larger than 8 digits" message, you could use:

```
BEGIN { FLOATMASK = and(FLOATMASK, not(0x800)) }
```

To turn this message back on, you could use:

```
BEGIN { FLOATMASK = or(FLOATMASK, 0x800) }
```

[UNIX / PC Versions Note]:

Not all of the math processor errors are generated on all systems. Also, an operation that generates an error on one system may not generate an error on another system. However, floating point operations have been standardized by the IEEE, so the differences between math processors on the various systems can be ignored by casual users.

This is the number of records that have been read from the current file by the Automatic Input Loop. See also NR, which is the total number of records read from all files.

FPAT and FS specify how TAWK will break the current record (\$) into fields (\$1, \$2, etc.) The FPAT variable is used to specify the pattern that the fields must match. The FS variable is used to specify the pattern that the field separator, which is the space between the fields, must match. Either FS or FPAT may be specified, but not both. The one that is not in use must be disabled by being set to 0.

- 1) If FPAT is set to 0 (the default value) it is disabled. In this case FS will be used to find the fields instead of FPAT.
- 2) If FPAT is set to any other value it is the pattern that the fields must match.

- 1) If FS is set to 0 it is disabled. In this case FPAT will be used to find the fields instead of FS.
- 2) If FS is a space (" ") it means that fields are separated by any number of white-space characters (space, tab, form-feed, etc.) and additionally any leading or trailing white-space characters in the record are ignored.
- 3) If FS has any other single character value (for example: "|") then that character is the field separator. Note that each occurrence of this character separates fields. For example, if the record begins (or ends) with this character it means the first (or last) field will be an empty string.
- 4) If FS is a regular expression pattern or a multi-character string, it is the pattern that separates fields in the record.

In comma-quote files, the fields are separated by commas, but fields may also contain a comma if the entire field is quoted. For example:

This can not be handled properly by setting FS = "," because the comma in field 2 would be treated as a field separator. But this case is easily handled by setting FPAT as below. The FPAT pattern below says that a field either starts with a quote, contains anything but a quote, and ends with a quote, or that a field consists of anything but a comma or quote.

In some comma-quote files, the quoted fields are themselves allowed to contain quotes. Use the following if quotes are included by doubling.

Use the following if quotes are included by preceding with a backslash:

To make TAWK go back to using FS instead of FPAT, set $FPAT = 0$.

FPAT can also be used to tokenize program source files. The following TAWK program tokenizes C program files. This complete example also handles C comments, which are complicated by the fact that they can continue from one line to the next.

```
# These are patterns describing C language tokens:
local eq_tok = /=|=|!=|<=|>=|<|>/
local arith_tok = \
    /\+|\+|--|-->|>>|<<|&&|\||\||\|[-+*/%^~!&|(){}.,?:/]/
local assign_tok = />>=|<<=|[-+*/%^&|=]/
local id_tok = /[a-zA-Z_][a-zA-Z0-9_]/
local num_tok = /[0-9a-fA-F]+|0[0-7]*|[0-9]+/
local string_tok = /"([^\"]|\\")*/
local char_tok = /'([^\']|\\')'/
local other_tok = /[^\t\f]/ # Any other character.
local OR = "|"

BEGIN {
    FS = 0 # Disable FS
    # Set FPAT to recognize C tokens:
    FPAT = eq_tok OR arith_tok OR assign_tok OR \
        id_tok OR string_tok OR char_tok OR \
        num_tok OR other_tok
}

# Skip pre-processor statements (they begin with #) /^#/ { next }

# If in a multi-line comment look for: "*/"
# The incomment variable is TRUE if we are
# in a multi-line comment.
incomment && /\*\\ / {
    sub(/^.*\\ /s,"")
    incomment = 0;
}

# Skip multi-line comments:
incomment { next }

# Delete in-line comments /* like this */
# Note s flag on the pattern for shortest match:
{ gsub(/\/*.*\\ /s,"") }

# Look for start of multi-line comment:
/\*\\ / {
    sub(/\/*.*\\ /s,"");
    incomment = 1;
}

# Print out the C tokens, one per line:
{
    for (i = 1; i <= NF; i++) { print $i }
}
```

See also: the split() and splitp() functions.

FUNINFO

For Experts Only! This array is used only by the TAWK debugger. It contains information about functions defined in your TAWK program. The format of the FUNINFO array is described in online documentation included with the TAWK Compiler, just in case you want to write your own debugger.

MALLOCS

For Experts Only! This variable contains the number of memory allocations used internally by TAWK. It is used to debug TAWK itself.

MEMAVAIL

[DOS Version]

This variable shows the amount of regular DOS memory still available. This variable is of little use because when regular memory fills up a compiled program simply starts using extended or expanded memory or disk space. There is currently no easy way for a TAWK program to determine how much extended and/or expanded memory and/or disk space is left for the program's use. When a TAWK program runs another external program using the `system()` or `spawn()` function it swaps out most of its data space, so the MEMAVAIL variable has no bearing on how much memory will be available for the external program (unless SYSSWAP = 0). EXCEPT: in a combined TAWK/C program created with the aid of a C compiler the MEMAVAIL variable contains the amount of memory that is available to run an external program using the `system()` or `spawn()` functions.

[DOS/32 Version]

MEMAVAIL returns the total amount of memory available, including virtual memory. This can be a very big number.

[OS/2 Version]

MEMAVAIL returns the size of the largest block of real memory available. The total memory available may be much larger than this. OS/2 may spend some time determining how much memory is available.

[UNIX Versions]

MEMAVAIL returns the total amount of memory available. Calling MEMAVAIL may take time because it forces a complete flush of all cached memory while determining the amount of memory available. You should probably avoid using MEMAVAIL under UNIX and just assume you have infinite memory available.

See the release information supplied with your version for possible changes or additional information.

NF

This is the number of fields in the current record (\$0). Assignment to NF is permitted: if the new value is less than the old the extra fields are deleted; if the new value is greater than the old then extra empty fields are added.

NR

This contains the total number of records that have been processed from all files processed by the Automatic Input Loop. This number accumulates as files are processed. NR is initially 0. See also FNR.

OFMT

This variable contains the format specification that is used when it is necessary to automatically convert a floating point number into a string for an output statement such as the `print` or `printf`. The format specification is in the same format used by the `printf` statement. See the `printf` statement for a list of valid formats. Note that OFMT affects only floating point numbers, not integers. The default format is "%.6g". See also: CONVFMT.

This example changes the default floating point output format:

```
BEGIN {
  print 1.1      # Prints "1.1"
  OFMT = "%.6f"
  print 1.1      # Prints "1.10000"
}
```

OFS

This variable is used for two purposes:

- 1) OFS separates arguments that are output by the print statement.
- 2) When \$0 is reconstructed as a result of assignment to a field or to NF the new value for \$0 is created by concatenating the old fields separated by the value of OFS.

The default value of OFS is a space: " ".

The following two statements are identical but the first executes faster:

```
BEGIN {  
    print a,b,c  
    print a OFS b OFS c  
}
```

ORS

The ORS variable contains the output record separator that is output after every print statement. The ORS variable is not used by the printf statement which must include the record separator explicitly.. The default value of ORS is "\n".

OSMODE

This variable indicates the operating system being used. The value of this variable is initialized to one of the values below. Note that OSMODE indicates the operating system that the program was compiled for, not necessarily the one being used. For example, a DOS program will always have OSMODE == 0, even if it is run under Windows NT or OS/2.

- 0 normal DOS, or Windows 3.1;
- 1 OS/2;
- 2 32-bit extended mode DOS;
- 3 Win32: either Windows NT or Windows 95;
- 4 UNIX.

PROGFN, PROGLN

For experts: the PROGFN and PROGLN read-only variables contain, respectively, the source file name and line number of the currently executing program statement. These may be useful for debugging purposes. For example, you can print this variable at various places in your program to indicate what line number is being executed. Implementation note: these are not really run-time variables at all. The actual values for PROGFN and PROGLN are known and substituted at compile time.

PROGTIME, PROGCTIME

For experts: the PROGTIME and PROGCTIME read-only variables contain date the currently executing program statement was compiled.. PROGTIME is the number of seconds since 1970, and PROGCTIME is a string representing the current date and time. Simply using PROGCTIME in your program will embed the program compilation time into the executable file. The "strings" function, supplied with the Thompson Toolkit, can later be used on the executable file to determine when the program was compiled. Implementation note: these are not really run-time variables at all. The actual values for PROGTIME and PROGCTIME are known and substituted at compile time.

PROMPT

This variable contains the prompt that TAWK prints before it reads a record from the terminal. The PROMPT is only printed when TAWK reads from its standard input, and the standard input is a terminal, and either standard output or error output is also a terminal. The default PROMPT is "tawk input? ". To suppress the prompt set PROMPT = "" in your BEGIN block.

RAWMODE

[DOS, Win32 and OS/2 Versions]

This variable should not be used unless you are an EXPERT. This discussion applies only to normal TAWK programs that are not combined TAWK/C programs (i.e., programs created by linking with C code.) See the chapter on Combined TAWK/C programs for a discussion of how RAWMODE works in a combined TAWK/C program.

The RAWMODE variable is an integer whose bottom three bits are binary flags that separately control TAWK end-of-line and Control-Z processing. Normally RAWMODE = 0 causing TAWK to:

- A) stop processing the file whenever a Control-Z is encountered;
- B) translate "\r\n" to "\n" on input;
- C) translate "\n" to "\r\n" on output.

RAWMODE may be set to other values as follows:

RAWMODE value	Meaning
1	TAWK will not treat Control-Z specially. This bit is ignored in a combined TAWK/C program, which uses RAWMODE = 2 to control both input translation and Control-Z processing.
2	TAWK will not translate "\r\n" to "\n" on input.
4	TAWK will not translate "\n" to "\r\n" on output.

The values of RAWMODE can be combined by adding them together. For example if RAWMODE = 7, then all three flags are enabled and no special processing takes place on either input or output.

RAWMODE has no effect on files that are opened with fopen() using the "t" (text-mode) or "b" (binary-mode) flags.

You should be aware that if RAWMODE is enabled then funny results will ensue when you print to the terminal. For example, if a line printed to the terminal ends only with "\n" instead of "\r\n", then the cursor is not moved to the left hand side of the screen. Also be aware that when printing to the terminal MS-DOS truncates lines containing a Control-Z character.

Most DOS editors can not edit any part of a file after any Control-Z character in the file. If you allow a file to be created that contains Control-Z characters and try to edit it you may find that you can edit only the first part of the file.

The implementation of the RAWMODE variable may change in future releases of TAWK.

[UNIX Version]

RAWMODE defaults to 7, that is, no characters are treated specially. However, RAWMODE is implemented, and can be used, for example, to translate files from UNIX text format to PC text format.

RECLEN

This variable indicates the maximum input record length. The default value is 8000 (DOS) or 32000 (DOS/32 or other versions). Input records that exceed this maximum length will be split apart. Any left over characters are added to the next record. TAWK normally translates the "\r\n" sequence used by DOS to indicate end of line into the single character "\n". If the input record is split exactly between the "\r" and the "\n" this automatic translation may not occur. Also note that RECLEN is the length of the input record AFTER translating "\r\n" to "\n", so if a record contains "\r\n" sequences its length as a string in TAWK will be less

than the length of the record as it appeared in the file. This can be corrected by preventing such translation with the RAWMODE variable.

To read in fixed length records you can set the RECLLEN variable to the fixed record length and disabling the RS (record separator) variable by setting it to 0.

RELENGTH

This variable is set by the match() function. It is the length of the matching string found by the last match() function call. See the match() function for more information.

RS

This is the record separator that TAWK looks for in input files when it reads in records. There are four possible cases:

- 1) If RS = 0 then no record separator is recognized. In this case RECLLEN specifies the length of the input records.
- 2) If RS is any single character, that character is the record separator. The default value for RS is RS = "\n", which causes records to be individual lines. Note that under DOS and OS/2 lines are terminated by "\r\n" but TAWK normally translates "\r\n" to "\n" transparently as the line is read in. (See the RAWMODE variable for more information.)
- 3) If RS = "" (an empty string), it is identical to RS = "\n\n". This special case was used in older versions of awk to read in multi-line records. Now RS can be a pattern so this special case is obsolete, but still supported.
- 4) If RS is a regular expression or a multiple character string it is the regular expression pattern that describes the record separator. There are some restrictions on patterns used as a record separator:
 - 1) The RS pattern may not contain the ^ or \$ pattern characters.
 - 2) The RS pattern must not require TAWK to look ahead more than one character in the input stream in order to determine if the pattern matches or not.

Warning messages are printed if these restrictions are not observed. The first restriction is checked when an assignment is made to RS. The second is not checked until an input record is encountered that violates the one character look-ahead restriction. The following examples illustrate the last restriction:

The following example is a legal value for RS:

```
RS = /a|ab/
```

The following example is incorrect because, for example, if the input record is "abc" then TAWK must read the "c" before it can determine that the record separator was the "a", and then it has to push the "bc" back to the input stream, which is more than one character of push-back and is illegal.

```
RS = /a|abb/      # This is Not OK!
```

RSTART

This variable is set by the match() function. It is the starting index of the match found by last match() function call. See the match() function for more information.

RSM

This variable holds the last input record separator found. It is set whenever a record is read in by the Automatic Input Loop or by the getline function. RSM is useful when the RS variable contains a pattern, so you can determine what was the actual record separator that was matched. RSM is also useful if you are processing records so large that their length might exceed the RECLLEN variable, and be split up: if RSM is an empty string (""), it indicates that no record separator was found, either because the end of file was reached or because the record length reached RECLLEN characters.

SIGNAL

This variable is an array that specifies how TAWK will respond to signals. Signals are asynchronous events, such as interrupts. For example, when the user presses the Control-C key, this generates a signal to TAWK. The indices of the SIGNAL array are the names of signals that TAWK recognizes. The value of each element in the SIGNAL array is a string that indicates how TAWK should respond to that specific signal.

Permitted SIGNAL Values

<u>SIGNAL Value</u>	<u>Action</u>
"DEFAULT"	TAWK will take the default action for this signal. The default action usually causes the TAWK program to abort.
"ABORT"	TAWK will abort the program. Note that TERM blocks are always executed, if possible, even if a program is aborted.
"IGNORE"	TAWK will ignore the signal, if possible.
"FAIL"	A special value allowed only for the "SIGCRIT" signal. See "SIGCRIT", below.
Any other string	This value is the name of a TAWK function that TAWK will call when the signal occurs. When the function returns, your program will continue from where it left off, unless otherwise documented below.

SIGNALS Recognized by TAWK

<u>SIGNAL Index</u>	<u>Discussion</u>
"SIGINT"	User interrupt, typically generated if user presses Control-C. Occurs in all operating systems. Under DOS, this interrupt is also generated if the user presses Control-Break.
"BREAK"	User pressed Control-Break. Occurs in OS/2, Win32.
"SIGCRIT"	Critical error. Occurs in DOS, OS/2, Win32. The most common critical errors are file input or output to a non-existent or locked file. The valid values for this signal are "DEFAULT", "ABORT", or "FAIL". The "DEFAULT" value usually causes the operating system to display a pop-up box when a critical error occurs, allowing the user to select "abort, retry, ignore, fail." The "ABORT" value causes the TAWK program to abort without displaying the pop-up box. The "FAIL" value causes the operation to fail without displaying the pop-up box. Any other value is invalid, so you can not instruct TAWK to call a function when a critical error occurs. However, if you set SIGNAL["SIGCRIT"] = "FAIL", the failure indication will be passed along to your TAWK program. The ferror() function can be used to see if an error has occurred on a filename. The ERRNO_CRIT variable will contain information about the error.
"SIGCLOSE"	Window was closed. Occurs in Win32. This signal can not be ignored. If you provide a handler, TAWK executes the handler, then TERM blocks, then exits.
"SIGTERM"	Software Termination signal. Occurs in UNIX, Win32 and OS/2 32 bit versions. This signal is sent when the computer is being shut down. Your program has only a few seconds to live. This signal can not be ignored. If you provide a handler, TAWK executes the handler, then TERM blocks, then exits.
"SIGHUP"	Logoff or hangup. Occurs in UNIX, Win32. In UNIX, this signal is also sent when the window is closed. In Win32, this signal can not be ignored. If you provide a handler, TAWK executes the handler, then TERM blocks, then exits.
"SIGQUIT"	User interrupt that generates a core dump if not caught or ignored. Occurs in UNIX.

The UNIX version may recognize additional signals. To get the complete list, run this program using your UNIX version of TAWK:

```
BEGIN { for (i in SIGNAL) print i }
```

SIGNAL array elements that are not recognized are ignored. Therefore, if you have to run on multiple operating systems, you can set all the signals you may need, and TAWK will implement only those that make sense on the current operating system.

The following example will print a message whenever the user presses Control-C. To demonstrate this program, add it to a larger program that actually does something, so you will have a chance to press Control-C while it is running.

```
function myhandler() {
    print "ha ha, you cant exit"
}
BEGIN { SIGNAL["SIGINT"] = "myhandler"; }
```

The following example demonstrates one way to test if a disk is in a floppy drive, without intervention of the operating system.

```
BEGIN {
    SIGNAL["SIGCRIT"] = "FAIL"
    if (! chdir("A:")) print "no floppy in drive A:"
    SIGNAL["SIGCRIT"] = "DEFAULT"
}
```

See also: `ERRNO_CRIT` variable for additional information on `SIGCRIT`; `FLOATMASK` variable to control floating point exceptions; the discussion of `TERM` blocks in chapter 3. If all you want to do is perform some action no matter how your program ends, it is safer and easier to use a `TERM` block than to install signal handlers.

SORTTYPE

`SORTTYPE` controls the automatic sorting of arrays that are accessed with the "for (*variable* in *array*)" construct. The possible values you may assign to `SORTTYPE` are:

<u>SORTTYPE Value</u>	<u>Meaning</u>
<code>SORTTYPE=0</code>	No automatic sorting; the array elements will be accessed in a semi-random order.
<code>SORTTYPE=1</code> (default value)	Alphanumeric sorting; numeric indicies are sorted numerically and alphabetic indicies are sorted alphabetically. Only integer and fixed point notation (eg: 1.9 or 0.007) are sorted numerically. Exponential notation (eg: 1e10) is not recognized by the sorting routine.
<code>SORTTYPE=2</code>	Alphabetic sorting using ASCII collating sequence.

You can affect the sort order by adding any of the following values to `SORTTYPE`:

<u>SORTTYPE Value</u>	<u>Meaning</u>
Add 4	Upper and lower case letters will be sorted together. If entries differ only in case the upper-case entry will come first. The standard English character set is used to determine the case of letters.
Add 8	The sorting order is reversed. If you are not sorting, this has no effect, i.e., <code>SORTTYPE = 8</code> is the same as <code>SORTTYPE = 0</code> .
Add 16	Ignore fractions. This option allows you to sort things like: "fig 1.9" and "fig 1.10" in the intuitive manner. Otherwise the 1.9 and 1.10 are treated as fractions, so "1.10" comes before "1.9".
Add 32	Leading zeros are treated as a significant part of the number. For example, if <code>SORTTYPE = 33</code> , the numbers would be sorted: 1, 2, 01, 02, rather than: 1, 01, 2, 02.

For example, `SORTTYPE = 2` selects alphabetic sorting, `SORTTYPE = 6` selects case-insensitive alphabetic sorting, and `SORTTYPE = 14` selects reversed alphabetic case-insensitive sorting.

The value of `SORTTYPE` is used once each time a **for** loop is encountered. If you want to use different sort orders in different places in the same program, you should set `SORTTYPE` immediately before each **for** loop. The following example shows how to use nested **for** loops that use different sort types:

```

END {
    SORTTYPE = 2
    # array1 will be sorted alphabetically
    for (i in array1) {
        SORTTYPE = 10
        # array2 will be sorted in reverse
        for (j in array2) {
            ...
        }
    }
}

```

Sorting arrays is time consuming. If the size of the array exceeds available real memory the sorting operation will slow down drastically. SORTTYPE = 0 can be used to speed up a **for** loop if sorting is not required.

See the chapter on arrays for sorting examples.

stdin, stdout, stderr

These pre-defined variables represent the standard-input, the standard-output, and the standard-error-output when your TAWK program is executed. By default they are connected to the user's console but they can also be redirected when the program is invoked using the < and > symbols in the command line. (Standard-error can not normally be redirected under DOS unless you use an enhanced shell like the Thompson Toolkit.)

For example, to print a message to the standard-error-output, use:

```
BEGIN { print "a message" > stderr }
```

To read 512 bytes from the standard-input you could use:

```
BEGIN { buf = fread(512,stdin) }
```

Do not confuse stdin with the keyboard. Standard-input normally comes from the keyboard but if the standard-input for the program is redirected it comes from the specified file. If you want to read from the keyboard you can either use the `getkey()` function, or you can read directly from the device driver using the special filename "CON" under DOS or OS2 or "/dev/tty" under UNIX. This special filename always stands for Console and can not be redirected. For example, to read a line of input from the keyboard regardless of file input redirections, you can use:

```
getline msg < "CON"          # DOS, OS/2 versions  getline msg < "/dev/tty"      # UNIX
version
```

Similarly for stdout and stderr: these normally go to the user's console but they can be redirected. To write to the terminal regardless of file output redirection you can use:

```
print "a message" > "CON"      # DOS, OS/2 versions
print "a message" > "/dev/tty" # UNIX version
```

STACKAVAIL

STACKAVAIL is the number of bytes of stack left. It is provided for your interest only. The initial stack size is set to a very high value (12000 bytes even under DOS). If you get a "stack overflow" message, it usually means that you have a runaway function recursion problem.

SUBSEP

This variable is used as part of an old method to simulate multi-dimensional arrays using one dimensional arrays. TAWK provides true multi-dimensional arrays so this is obsolete, but it is still supported for backward compatibility. See the chapter on Arrays for a full description.

SYMINFO

For Experts Only! This array is used only by the TAWK debugger. It contains information about variables defined in your TAWK program. The format of the SYMINFO array is described in online documentation included with the TAWK Compiler, just in case you want to write your own debugger.

SYSSWAP

[DOS Version Only]

Under DOS, this variable controls whether TAWK programs will swap out their data space before executing another program using the system() or spawn() functions. This variable is used only in the normal DOS version of TAWK, not in the DOS/32 or other versions.

Under DOS, TAWK swaps out its data space to make more room in memory for the executed program. Without swapping, most large TAWK programs would not be able to run other programs at all. By default, the SYSSWAP variable is non-zero, causing swapping to occur. Setting SYSSWAP=0 suppresses swapping. There are several possible reasons to suppress swapping:

- 1) To conserve EMS/XMS memory for use by the program you want to run;
- 2) To run programs that must remain resident. If a resident program is loaded on top of a TAWK program that is swapped out, the TAWK program immediately terminates with an indignant error message;
- 3) To run programs requiring too much environment space. Before the TAWK program swaps itself out, it copies the environment for the program to be called to its own stack space in order to save it in a safe place. However, there may be insufficient space on the stack to hold the environment, in which case the environment will be truncated. If TAWK does not swap out, the limit on the environment size is much larger.
- 4) If the user of the compiled TAWK program does not have enough EMS or XMS memory or disk space to perform the swapping operation, the system() or spawn() call will fail.

In combined TAWK/C programs, swapping is normally disabled unless specifically enabled by setting the awk_xmalloc variable in your C code. Memory allocation and swapping in combined TAWK/C programs is an involved subject discussed in the chapter on combined TAWK/C programs.

TMPDIR

This variable specifies the directory where the TAWK program will create temporary files. It is initialized to the value of the TMP environment variable, if any. TAWK does not create temporary files until it needs them. If you change TMPDIR before TAWK creates a temporary file TAWK will use the new directory that you specify. Make sure you set TMPDIR to a valid directory path.

[DOS Versions]

Under DOS TAWK performs its own virtual memory management using a temporary paging file created in this directory. Therefore you may be able to use more memory in your TAWK program if you set this variable to a directory on a disk that has at least 16 Mb of free disk space. You may also be able to make very large TAWK programs run faster under DOS if you set this variable to a RAM-DISK. However, you are always better off letting TAWK use the memory directly as XMS or EMS memory than indirectly via a RAM-DISK.

VENDOR

This variable contains the value "Thompson Automation".

VERSION

This contains the current TAWK version number as a string, for example, "5.0".

WARNINGS

This variable controls warnings printed by your TAWK program.

WARNINGS = 0 disables all warning and note messages.

WARNINGS = 1 enables general warning messages;

WARNINGS = 2 enables math warning messages;

WARNINGS = 4 enables note messages. "Note" messages are like "Warnings" but are not as serious.

You can add the various values above to individually enable different types of warnings. The default value of the WARNINGS variable is 7 (all warning and note messages enabled) unless the -w option was specified to your TAWK program, in which case WARNINGS=0.

See also: FLOATMASK variable.

XMSLIMIT, EMSLIMIT and XMSRESIZE

[DOS Version Only]

Under DOS, TAWK programs use memory in the following order:

- 1) Regular memory (typically below 640K) first;
- 2) Extended memory (up to XMSLIMIT);
- 3) Expanded memory (up to EMSLIMIT);
- 4) Disk temporary file.

Extended memory is memory above 1 Megabyte when running under DOS. In order for a TAWK program to use extended memory, an extended memory manager such as Microsoft's HIMEM.SYS must be installed first. Expanded memory is a different type of memory and may be provided by an expanded memory manager in a 386-based computer or by an add-in memory card in any computer.

Normally you do not need to worry about XMSLIMIT and EMSLIMIT: TAWK programs automatically check for the presence of extended or expanded memory and use it if available. The default values of XMSLIMIT and EMSLIMIT allow compiled TAWK programs to use all available extended and/or expanded memory.

However, if your TAWK program needs to call another program that needs extended or expanded memory, you may set XMSLIMIT to specify the maximum number of Kbytes of extended memory that the compiled TAWK program may use, and/or EMSLIMIT to specify the maximum number of Kbytes of expanded memory that the TAWK program may use. For example, if XMSLIMIT=0 then no extended memory will be used, and the TAWK program will not even check for the existence of extended memory. If XMSLIMIT=320 then a maximum of 320 Kbytes of extended memory will be used.

By default, when TAWK needs XMS memory, it allocates it all. If you set XMSRESIZE = 1, TAWK will only allocate as much as it needs, and will reallocate more as it needs it, up to the limit specified by XMSLIMIT. This variable might be useful if you are calling, from TAWK, a program that needs to use XMS memory. Unfortunately, some popular XMS memory managers contain bugs that can crash your computer if this feature is used. Thompson Automation can not provide you with a list of memory managers that do, or do not, work. Therefore, we recommend that you use this variable only if you absolutely need it, and only with caution, and only on your own computer. We strongly recommend that you do NOT use this variable at all in any software that you ship to your customers.

[DOS/32, Win32, OS/2 or UNIX Versions]

TAWK programs compiled for these systems use the virtual memory capabilities provided by the underlying operating system or DOS extender, and the XMSLIMIT and EMSLIMIT variables are ignored.

Appendix 1: Details of Number Representation

TAWK stores numbers internally either as 32 bit integers or as double precision floating point numbers, whichever is appropriate. Integers are used whenever possible because integer calculations are much faster than floating point calculations. However, a number that is entered with a decimal point or using scientific notation is stored as a floating point number, even if it could be stored as an integer. A feature of TAWK is that when an integer calculation over-flows during an arithmetic calculation, TAWK automatically recalculates the result using floating point calculations and stores the correct result as a floating point number. On IBM compatible computers, numbers in TAWK have 15 significant decimal digits and may fall in the range 1e-307 to 1e308. TAWK automatically uses a floating point co-processor if one is present in the computer; otherwise floating point operations are performed by software subroutines.

The 32 bit signed integers used by TAWK have a range from -2147483648 to 2147483647. Numbers entered in hexadecimal notation are stored directly as 32 bit signed integers. Therefore, when used in a signed arithmetic context (such as addition) hexadecimal numbers in the range 0x0 to 0x7fffffff represent positive numbers in the range 0 to 2147483647, and hexadecimal numbers in the range 0x80000000 to 0xffffffff represent negative numbers in the range -2147483648 to -1. The reason for this is that TAWK considers the integers used in arithmetic contexts to be signed quantities. In a 32-bit signed quantity, the top bit is the sign bit. Hexadecimal numbers larger than 0x7fffffff (which is $2^{31} - 1$) have the upper most bit set, so they act like negative numbers when used in an arithmetic context. Normally, you would never notice this, because hex numbers are typically not used in arithmetic contexts; they are usually used in logical contexts such as the bitwise manipulation functions: and, or, etc. Consider the following two statements that have the same effect:

```
x = xor(y, -1)
x = xor(y, 0xffffffff)
```

The sprintf, pack and unpack functions allow you to specify whether a value is to be treated as a signed or unsigned quantity. An interesting problem arises when a 32 bit unsigned integer larger than 2147483647 is converted to a number. To accurately preserve the sign and value of the number, TAWK stores these numbers as floating point numbers.

For mathematical completeness, we will here present how to convert a 32-bit unsigned integer to a signed floating point number. Since TAWK considers all integers to be signed integers, some deviousness is required. The following program determines what value a negative 32 bit signed integer would have if it were treated as an unsigned quantity. The converted value will be a positive number stored as floating point if necessary.

```
# This prints: signed = -1 unsigned = 4.29497e+9
BEGIN {
    val = 0xffffffff
    uval = 0 + sprintf("%u", val)
    print "signed =", val, "unsigned =", uval
}
```

This works too:

```
BEGIN {
    val = 0xffffffff
    # Use pack and unpack to convert
    # from signed to unsigned.
    unpack("@L", pack("@L", val), x)
    print "signed =", val, "unsigned =", x[1]
}
```

Round-Off Error Suppression

TAWK goes to some effort to suppress round off errors. Consider the following program:

```
BEGIN {
    x1 = 1.23;
    x2 = 1.0 + .23;
    printf("%g\n", x2 - x1);
    if (x1 == x2) printf("equal\n");
    else printf("not equal\n");
}
```


Results In:

<u>TAWK</u>	<u>Other Language</u>
0	2.77556e-017
equal	not equal

The reason this fails in most other languages (including C, C++ and all UNIX versions of AWK) is that floating point calculations almost always result in some round off error, so when two numbers calculated two different ways are compared they are usually not quite the same.

In TAWK, the number of significant digits specified when a number is entered is saved with the number. This information is preserved across addition, subtraction, multiplication, and division by an exact multiple of 2 or 10. Then when numbers are compared, only the significant digits participate. If a result is zero within the significant digits supplied with the number, then it is considered a true zero instead of a teeny tiny number.

Note that the results of non-trivial division or modulus and the results of all transcendental functions are treated as though all result digits are significant, so they do not benefit from TAWK's round off error suppression. For example, the following example usually prints a teeny tiny number rather than zero:

```
print((1.0/.1 + .23/.1) - 1.23/.1)
```

Appendix 2: Compatibility with AWK

There are a few minor conflicts between the "The AWK Programming Language" book (which is the primary reference for the AWK language), the POSIX extensions to AWK, and existing implementations of AWK on UNIX systems. These are described here.

Common AWK Incompatibilities:

- 1) Backslashes in Strings: Unfortunately, there are a few UNIX versions of awk that interpret backslashes in strings incorrectly. TAWK can interpret backslashes either way. See the TAWK -eb option in the TAWK Compiler chapter for a complete discussion.
- 2) Reserved names: TAWK has more built in functions and variables than other versions of awk. The names of built in functions and variables are reserved. If an AWK program ported from UNIX uses any reserved names, you will have to change the names in the AWK program before it can be run using TAWK.
- 3) Function Evaluation: TAWK evaluates function arguments from right to left. While the order of evaluation is unspecified in the AWK book, most other versions of AWK evaluate function arguments from left to right. TAWK uses right to left evaluation to make it easier to link to C libraries and call C functions directly from TAWK, because C also uses right to left evaluation.
- 4) Regular expressions: The relative precedence of the anchoring operators (^ and \$) and the alternation operator (|) is different in some versions of AWK. Thus the regular expression:

```
^a|b$
```

In TAWK and most versions of awk means:

```
(^a) | (b$)
```

But in non-conforming versions of awk might mean:

```
^(a|b)$
```

- 5) Assignment: TAWK evaluates the right-hand side of the assignment first, which is C compatible. Some other AWKs evaluate the left-hand side of an assignment first. For example:

```
x = 1
line[x] = line[x++]
```

In TAWK results in:

```
line[2] = line[1]
```

But in some other AWKs the left hand side (line[x]) is evaluated first resulting in:

```
line[1] = line[1]
```

- 6) sub and gsub functions.

The original POSIX standard seemed to imply that the replacement string specified to the sub and gsub functions must undergo an additional level of backslash interpretation. A subsequent clarification by the POSIX committee members corrected this. As a result, different AWK implementations of sub and gsub work different ways, and all can claim to be correct. TAWK is compatible with the latest POSIX interpretation, which is also compatible with all older versions of AWK. But TAWK also recognizes the \$n construct in the replacement string of the sub and gsub functions, while other AWKs do not. The result is that calls to sub and gsub are not 100% portable between different versions of AWK.

Index to Functions and Variables

- abort function, 12, 14, 15, 24, 86, 93
- addressof function, 86, 87, 104, 128
- and function, 32, 87
- ARGC variable, 15, 16, 17, 23, 131
- argcount function, 34, 88
- ARGI variable, 15, 17, 23, 55, 131
- ARGV variable, 15, 16, 17, 18, 23, 55, 64, 66, 67, 82, 131
- argval function, 88
- atan2 function, 35, 36, 88

- BUFSIZE variable, 131

- call function, 88
- calla function, 88
- char function, 88, 105
- chdir function, 86, 89
- chmod function, 89
- chsize function, 90
- close function, 14, 18, 21, 24, 86, 90, 95, 102
- convertnum function, 91
- CONVFMT variable, 27, 131
- cos function, 35, 91
- ctime function, 91

- debug_function function, 91
- debug_get_frame function, 91
- debug_get_stack_var function, 91
- debug_set_stack_var function, 91
- dirlist function, 92
- DLLS variable, 132

- EMSLIMIT variable, 146
- ENV variable, 132
- ENVIRON variable, 119, 125, 132
- ERRNO variable, 132
- ERRNO_CRIT variable, 133
- ERRNO_OS variable, 133
- exit function, 11, 12, 14, 15, 92, 93
- exp function, 93

- fdopen function, 93
- feof function, 94
- ferror function, 94
- fflush function, 85, 94
- filemode function, 94
- FILEMODE variable, 133
- FILENAME variable, 11, 14, 18, 133
- fileno function, 84, 85, 95
- filesize function, 96
- filetime function, 91, 96, 126
- findfirst function, 96, 97
- findnext function, 96
- FLOATMASK variable, 133
- flock function, 97, 98, 100
- FNR variable, 10, 11, 15, 22, 101, 102, 135
- fopen function, 31, 82, 84, 85, 90, 94, 95, 97, 99, 100, 128
- FPAT variable, 8, 48, 49, 120, 135
- fread function, 23, 99, 100
- FS variable, 8, 48, 49, 119, 135
- fseek function, 11, 94, 97, 100
- ftell function, 100
- FUNINFO variable, 136
- funlock function, 97, 98, 100
- fwrite function, 100

- getawkvar function, 100
- getcwd function, 101
- getkey function, 24, 101, 104, 144
- getline function, 19, 21, 22, 23, 24, 90, 95, 99, 101, 140, 144
- gsub function, 5, 53, 54, 123, 125, 136, 149
- gsubs function, 123

- index function, 87, 103
- inp function, 103
- inpw function, 103
- int function, 27, 103
- interrupt function, 85, 86, 87, 103

- kbhit function, 24, 101, 104

- length function, 17, 57, 104
- log function, 104

- MALLOCS variable, 137
- match function, 33, 49, 53, 54, 55, 104, 125, 140
- MEMAVAIL variable, 137
- mkdir function, 105

- NF variable, 11, 22, 46, 48, 50, 101, 136, 137, 138
- not function, 32, 87
- NR variable, 12, 22, 43, 44, 50, 101, 102, 137

- OFMT variable, 27, 131, 137
- OFS variable, 19, 48, 49, 109, 138
- or function, 32, 87
- ord function, 105, 109
- ORS variable, 19, 20, 109, 138
- OSMODE variable, 104, 138
- outp function, 105
- outpw function, 105

- pack function, 49, 77, 87, 106, 128, 147
- paste function, 108
- peek function, 87, 108
- poke function, 109

- ul style="list-style-type: none; padding-left: 0;">
- print function, 3, 10, 11, 12, 13, 19, 20, 21, 22, 23, 24, 36, 49, 87, 90, 99, 109, 131, 137, 138
- printf function, 19, 20, 21, 24, 87, 99, 110, 121, 131, 137, 147
- PROGCTIME variable, 138
- PROGFN variable, 138
- PROGLN variable, 138
- PROGTIME variable, 138
- PROMPT variable, 139
- rand function, 114, 121
- RAWMODE variable, 82, 97, 98, 100, 139, 140
- RECLLEN variable, 23, 102, 139, 140
- regex function, 52, 55, 57, 114
- registercallback function, 114
- rename function, 114
- rindex function, 115
- RLENGTH variable, 33, 53, 104, 125, 140
- rmdir function, 115
- rmfile function, 115
- RS variable, 23, 102, 140
- RSM variable, 23, 140
- RSTART variable, 33, 104, 125, 140
- scr_end function, 115
- scr_gcm function, 116
- scr_gcp function, 116
- scr_get function, 116
- scr_getcells function, 116
- scr_put function, 116
- scr_putcells function, 117
- scr_scm function, 117
- scr_scp function, 115, 117
- setawkvar function, 118
- shifl function, 118
- shiftr function, 86, 118
- SIGNAL, 12
- SIGNAL variable, 141
- sin function, 35, 118
- sleep function, 118
- SORTTYPE variable, 42, 43, 143
- spawn, 99
- spawn function, 83, 94, 119, 125, 132, 137, 145
- split function, 31, 49, 114, 119, 120
- splitp function, 31, 49, 120
- sprintf function, 109, 121, 147
- sqrt function, 55, 121
- srand function, 114, 121
- STACKAVAIL variable, 144
- stat function, 121
- stderr variable, 11, 84, 85, 144
- stdin variable, 84, 85, 144
- stdout variable, 84, 85, 144
- strdup function, 123
- sub function, 53, 54, 114, 123, 136, 149
- subs function, 123
- SUBSEP variable, 47, 144
- substr function, 43, 124
- SYMINFO variable, 145
- SYSSWAP variable, 137, 145
- system, 99
- system function, 83, 94, 119, 125, 132, 137, 145
- time function, 126
- timetab function, 126
- TMPDIR variable, 70, 145
- tolower function, 29, 127
- toupper function, 29, 127
- translate function, 43, 44, 127
- typeof function, 128
- unpack function, 49, 87, 106, 128, 147
- unregistercallback function, 130
- VENDOR variable, 145
- VERSION variable, 145
- WARNINGS variable, 68, 146
- XMSLIMIT variable, 146
- xor function, 32, 87, 147