NAME

awka − AWK language to ANSI C translator and library

SYNOPSIS

awka [−**c** *fn*] [−**X**] [−**x**] [−**t**] [−**o** *filename*] [−**a** *args*] [−**w** *args*] [−**I** *include-dir*] [−**i** *include-file*] [−**L** *lib-dir*] [−**l** *lib-file*] [−**f** *progname*] [−**d**] [program] [− −] [exe-args]
awka [−**version**] [−**help**]

DESCRIPTION

Awka is two products - a translator of AWK language programs to ANSI-C, and a library of essential functions against which the translated code must be linked.

The *AWK* language is useful for maniplation of datafiles, text retrieval and processing, and for prototyping and experimenting with algorithms. Usually AWK is implemented as an interpretive language - there are several good free interpreters available, notably **gawk**, **mawk** and 'The One True Awk' maintained by Brian Kernighan.

This manpage does not explain how AWK works - refer to the SEE ALSO section at the end of this page for references.

Awka is a *new awk* meaning it implements the AWK language as defined in Aho, Kernighan and Weinberger, *The AWK Programming Language*, Addison-Wesley Publishing 1988. **Awka** includes features from the Posix 1003.2 (draft 11.3) definition of the AWK language, but does not necessarily conform in entirety to Posix standards. **Awka** also provides a number of extensions not found in other implementations of AWK.

AWKA OPTIONS

−**c** *fn*
: Instead of producing a 'main' function, **awka** will instead generate '*fn*' as a controlling function. This is useful where the compiled C code is to be linked in with a larger application. The **-c** argument is not compatible with the **-X** and **-x** arguments. See the section **USING awka -c** below for more details on how to use this option.

−**X**
: **awka** will generate C code, which will then be compiled into an executable, using the C compiler and intallation paths defined when **Awka** was installed. The C code will be stored in 'awka_out.c' and the executable in 'awka.out' or 'awka_out.exe'.

−**x**
: The same as **-X**, except that the compiled program will also be executed using arguments following the '--' option on the command-line.

−**t**
: To be used in conjunction with **-x**. The C file and the executable will be removed following execution of the program.

−**o** filename
: To be used in conjunction with **-x** and **-X**. The generated executable will be called 'filename' rather than the default 'awka.out'.

−**a** args
: This embeds executable command-line arguments within the translated code itself. For example, *awka -X -a "-We" file.awk* will create an awka.out that will already have -We in its command-line when it is run. To see what arguments have been embedded in an executable, use -showarg at runtime.

−**w** args
: Prints various warnings to stderr, useful in debugging large, complex AWK programs. None of these are errors − all are acceptable uses of the AWK language. Depending on your programming style, however, they could be useful in narrowing down where problems may be occuring. *args* can contain the following characters:-

  a − prints a list of all global variables.

  b − warns about variables set to a value but not referenced.

  c − warns about variables referenced but not set to a value.

  d − reports use of global vars within a function.

  e − reports use of global vars within just one function.

f − requires declaration of global variables.

g − warns about assignments used as truth expressions.

NOTE: As at version 0.5.8 only a, b and c are implemented.

−**I** include-dir — Specifies a directory in which include files required by awka, or defined by the user, reside. You may use as many -I options as you like.

−**i** include-file — Specifies an include filename to be inserted in the translated code.

−**L** lib-dir — Specifies a directory containing libraries that may be required by awka, or defined for linking by the user. See the *awka-elm* manpage for more details.

−**l** lib-file — Specifies a library file to be linked to the translated code generated by awka at compile time (this only really makes sense if using awka -x). The lib-file is specified in the same way as C compilers, that is, the library libmystuff.a would be referred to as "-l mystuff".

Again, see the *awka-elm* manpage for details on awka extension libraries. Like the three previous options, you can use this as often as you like on a commandline.

−**f** progname — Specifies the name of an AWK language program to be translated to C. Multiple **-f** arguments may be specified.

program — An AWK language program on the command-line, usually surrounded by single quotes (’).

− − — All arguments following this will be passed to the compiled executable when it is executed. This argument only makes sense when -x has been specified.

exe-args — Arguments to be passed directly to the executable when it is run.

−**h** — Prints a short summary of command-line options.

−**v** — Prints version information then quits.

EXECUTABLE OPTIONS

An executable formed by compiling Awka-generated code against libawka.a will also understand several command-line arguments.

−**help** — Prints a short summary of executable command-line options, then exits.

−**We** — Following command-line arguments will be stored in the ARGV array, and not parsed as options.

−**Wi** — Sets unbuffered writes to stdout and line buffered reads from stdin.

−**v var=value** — Sets variable ’var’ to ’value’. ’var’ must be a defined scalar variable within the original AWK program else an error message will be generated.

−**F value** — Sets FS to value.

−**showarg** — Displays any embedded command-line arguments, then exits.

−**awkaversion** — Shows which version of awka generated the .c code for the executable.

ADDITIONAL FEATURES

awka contains a number of builtin functions may or may not presently be found in standard AWK implementations. The functions have been added to extend functionality, or to provide a faster method of performing tasks that AWK could otherwise undertake in an inefficient way.

The new functions are:-

totitle(*s*) — converts a string to Title or Proper case, with the first letter of each word uppercased, the remainder lowercased.

abort() — Exits the AWK program immediately without running the END section. Originally from TAWK, Gawk now supports abort() as well.

alength(*a*) — returns the number of elements stored in array variable *a*.

asort(*src* [,*dest*]) The function introduced in Gawk 3.1.0. From Gawk's manpage, this "returns the number of elements in the source array *src*. The contents of *src* are sorted using *awka'*s normal rules for comparing values, and the indexes of the sorted values of *src* are replaced with sequential integers starting with 1. If the optional destination array *dest* is specified, then *src* is first duplicated into *dest*, and then *dest* is sorted, leaving the indexes of the source array *src* unchanged."

ascii(*s*,*n*) Returns the ascii value of character *n* in string *s*. If *n* is omitted, the value of the first character will be returned. If *n* is longer than the string, the last character will be returned. A Null string will result in a return value of zero.

char(*n*) Returns the character associated with the ascii value of *n*. In effect, this is the complement of the ascii function above.

left(*s*,*n*) Returns the leftmost *n* characters of string *s*. This is more efficient than a call to substr.

right(*s*,*n*) Returns the rightmost *n* characters of string *s*.

ltrim(*s, c*) Returns a string with the preceding characters in *c* removed from the left of *s*. For instance, ltrim(" hello", "h ") will return "ello". If *c* is not specified, whitespace will be trimmed.

rtrim(*s, c*) Returns a string with the preceding characters in *c* removed from the right of *s*. For instance, ltrim(" hello", "ol") will return " he". If *c* is not specified, whitespace will be trimmed.

trim(*s, c*) Returns a string with the preceding characters in *c* removed from each end of *s*. For instance, trim(" hello", "oh ") will return "ell". If *c* is not specified, whitespace will be trimmed. The three trim functions are considerably more efficient than calls to sub or gsub.

min(*x1,x2,...,xn*) Returns the lowest number in the series *x1* to *xn*. A minimum of two and a maximum of 255 numbers may be passed as arguments to Min.

max(*x1,x2,...,xn*) Returns the highest number in the series *x1* to *xn*. A minimum of two and a maximum of 255 numbers may be passed as arguments to Max.

time(*year,mon,day,hour,sec*) time()
returns a number representing the date & time in seconds since the Epoch, 00:00:00GMT 1 Jan 1970. The arguments allow specification of a date/time, while no arguments will return the current time.

systime() returns a number representing the current date & time in seconds since the Epoch, 00:00:00 GMT 1 Jan 1970. This function was included to increase compatibility with Gawk.

strftime(*format, n*)
returns a string containing the time indicated by *n* formatted according to *format*. See strftime(3) for more details on format specification. This function was included to increase compatibility with Gawk.

gmtime(*n*) gmtime()
returns a string containing Greenwich Mean Time, in the form:-

Fri Jan 8 01:23:56 1999

n is a number specifying seconds since 1 Jan 1970, while a call with no arguments will return a string containing the current time.

localtime(*n*) localtime()
returns a string containing the date & time adjusted for the local timezone, including daylight savings. Output format & arguments are the same as gmtime.

mktime(*str*) The same as mktime() introduced in Gawk 3.1.0. See Gawk's manpage for a detailed description of what this function does.

and(*y,x*) Returns the output of '*y* & *x*'.

or(*y,x*) Returns the output of '*y* | *x*'.

xor(*y,x*) Returns the output of '*y* ^ *x*'.

compl(*y*) Returns the output of '~*y*'.

lshift(*y,x*) Returns the output of '*y* << *x*'.

rshift(*y,x*) Returns the output of '*y* >> *x*'.

argcount() When called from within a function, returns the number of arguments that were passed to that function.

argval(*n[, arg, arg...]*)
When called from within a function, returns the value of variable *n* in the argument list. The optional *arg* parameters are index elements used if variable *n* is an array. You may not specify values for *n* that are larger than **argcount()**.

getawkvar(*name[, arg, arg...]*)
Returns the value of global variable "*name*". The optional *arg* parameters work in the same as for **argval**. The variable specified by *name* must actually exist.

gensub(*r,s,f[,v]*) Implementation of Gawk's gensub function. It should perform exactly the same as it does in Gawk. See Gawk's documentation for details on how to use gensub.

The **SORTTYPE** variable controls if and how arrays are sorted when accessed using 'for (i in j)'. The value of this variable is a bitmask, which may be set to a combination of the following values:-

0 No Sorting
1 Alphabetical Sorting
2 Numeric Sorting
4 Reverse Order

A value for **SORTTYPE** of 5, therefore, indicates that the array is to be sorted Alphabetically, in Reverse order.

Awka also supports the **FIELDWIDTHS** variable, which works exactly as it does in Gawk.

If the **FIELDWIDTHS** variable is set to a space separated list of positive numbers, each field is expected to have fixed width, and awka will split up the record using the widths specified in **FIELDWIDTHS**. The value of **FS** is ignored. Assigning a value to **FS** overrides the use of **FIELDWIDTHS**, and restores the default behaviour.

Awka also introduces the **SAVEWIDTHS** variable. This applies when **FIELDWIDTHS** is in use, and **$0** is being rebuilt following a change to a **$1..$n** field variable.

If the **SAVEWIDTHS** variable is set to a space separated list of positive numbers, each output field will be given a fixed width to match these numbers. **$n** values shorter than their specified width will be padded with spaces; if they are longer than their specified width they will be truncated. Additional values to those specified in **SAVEWIDTHS** will be separated using **OFS**.

Awka 0.7.5 supports the inet/coprocessing features introduced in Gawk 3.1.0. See the documentation accompanying the Gawk source, or visit **http://home.vr-web.de/Juergen.Kahrs/gawk/gawkinet.html** for details on how these work.

EXAMPLES

The command-line arguments above provide a range of ways in which **awka** may be used, from output of C code to stdout, through to an automatic translation compile and execution of the AWK program.

(a) Producing C code:-

1. awka -f myprog.awk >myprog.c
2. awka -c main_one -f myprog.awk -f other.awk >myprog.c

(b) Producing C code and an executable:-

```
awka -X -f myprog.awk -f other.awk
```

(c) Producing the C and Executable, run the executable:-

```
awka -x -f myprog.awk -f other.awk -- input.txt
```

Afterwards, you could run the executable directly, as in:-

```
awka.out input.txt
```

Running the same program using an interpreter such as **mawk** would be done as follows:-

```
mawk -f myprog.awk -f other.awk input.txt
```

The following will run the program, passing it -v on the command-line without it being interpreted as an 'option':-

```
awka.out -We -v input.txt, OR
 awka -x -f myprog.awk -- -We -v input.txt
```

(d) Producing and running the executable, ensuring it
and the C program file are automatically removed:-

```
awka -x -t -f myprog.awk -f other.awk -- input.txt
```

(e) A simplistic example of how awka might be used in a Makefile:-

```
myprog:  myprog.o
      gcc myprog.o -lawka -lm -o myprog

 myprog.o:  myprog.c

 myprog.c:  myprog.awk
      awka -f myprog.awk >myprog.c
```

LINKING AWKA-GENERATED CODE

The C programs produced by **awka** call many functions in **libawka.a**. This library needs to be linked with your program for a workable executable to be produced.

Note that when using the **-x** and **-X** arguments this is automatically taken care of for you, so linking is only an issue when you use Awka to produce C code, which you then compile yourself. Many people many only wish to use Awka in this way, and never use **awka**-generated code as part of larger applications. If this is you, you needn't worry too much about this section.

As well as linking to **libawka.a**, your program will also need to be linked to your system's math library, typically **libm.a** or **libm.so**.

Typical compiler commands to link an **awka** executable might be as follows:-

```
gcc myprog.c -L/usr/local/lib -I/usr/local/include -lawka -lm -o myprog
```

OR

```
awka -c my_main -f myprog.awk >myprog.c
gcc -c myprog.c -I/usr/local/include -o myprog.o
gcc -c other.c -o other.o
gcc myprog.o other.o -L/usr/local/lib -lawka -lm -o myapp
```

If you are not sure of how your compiler works you should consult the manpage for the compiler. In release 0.7.5 Awka introduced Gawk-3.1.0's inet and coprocess features. On some platforms this may

require you to link to the socket and nsl libraries (-lsocket -lnsl). To check this, look at config.h after running the configure script. The #define awka_SOCKET_LIBS indicate what, if any, extra libraries are required on your system.

USING awka -c

The -c option, as described previously, replaces the main() function with a function name of your choosing. You may then link this code to other C or C++ code, and thus add AWK functionality to a larger application.

The command line "awka -c matrix 'BEGIN { print "what is the matrix?" }'" will produce in its output the function "int matrix(int argc, char *argv[])". Obviously, this replaces the main() function, and the argc and argv variables are used the same way - they handle what awka thinks are command-line arguments. Hence argv is an array of pointers to char *'s, and argc is the number of elements in this array. argv[0], from the command-line, holds the name of the running program. You can populate as many argv[] elements as you like to pass as input to your AWK program. Just remember this array is managed by your calling function, not by awka.

That's just about it. You should be able to call your awka function (eg matrix()) as many times as you like. It will grab a little bit of memory for itself, but you should see no growing memory use with each call, as I've taken quite some time to eliminate any potential memory leaks from awka code.

Oh, one more thing, *exit* and *abort* statements in your AWK program code will still exit your program altogether, so be careful of where & how you use them.

GOING FURTHER

Awka also allows you to create your own C functions and have them accessible in your AWK programs as if they were built-in to the AWK language. See the **awka-elm** and **awka-elmref** manpages for details on how this is done.

FILES

libawka.a, **libawka.so**, **awka**, **libawka.h**, **libdfa.a**, **dfa.h**

SEE ALSO

awk(1), **mawk**(1), **gawk**(1), **awka-elm**(5) **awka-elmref**(5), **cc**(1), **gcc**(1)

Aho, Kernighan and Weinberger, The AWK Programming Language, Addison-Wesley Publishing, 1988, (the AWK book), defines the language, opening with a tutorial and advancing to many interesting programs that delve into issues of software design and analysis relevant to programming in any language.

The GAWK Manual, The Free Software Foundation, 1991, is a tutorial and language reference that does not attempt the depth of the AWK book and assumes the reader may be a novice programmer. The section on AWK arrays is excellent. It also discusses Posix requirements for AWK.

Like you, I should probably buy & read these books some day.

MISSING FEATURES

awka does not implement **gawk**'s internal variable *IGNORECASE*. **Gawk**'s /dev/pid functions are also absent.

Nextfile and next may not be used within functions. This will never be supported, unlike the previous features, which may be added to **awka** over time. Well, so I thought. As of release 0.7.3 you _can_ use these from within functions.

AUTHOR

Andrew Sumner (andrewsumner@yahoo.com)

The **awka** homepage is at *http://awka.sourceforge.net*. The latest version of **awka**, along with development 'snapshot' releases, are available from this page. All major releases will be announced in comp.lang.awk. If you would like to be notified of new releases, please send me an email to that effect. Make sure you preface any email messages with the word "awka" in the title so I know its not spam.

NAME

awka-elm - Awka Extended Library Methods

DESCRIPTION

Awka is a translator of AWK programs to ANSI-C code, and a library (*libawka.a*) against which the code is linked to create executables. Awka is described in the *awka* manpage.

The Extended Library Methods (ELM) provide a way of adding new functions to the AWK language, so that they appear in your AWK code as if they were builtin functions such as substr() or index().

ELM code interfaces with the internal Awka variable structures and functions, and is suitable for anyone with some experience and proficiency in C programming.

This document is a step-by-step introduction to how the ELM works, so by the end of it you can write your own libraries to extend the AWK programming language using Awka. For example, you could write an interface to allow AWK programs to communicate with ODBC databases, or solve the travelling salesman problem given input of town locations - whatever you require AWK to do should now be possible.

AN OVERVIEW OF HOW IT WORKS

The C code produced by **awka** from AWK programs is heavily populated with calls to functions in the awka library (*libawka*). Hence after it is compiled, this code must be linked to the library to produce a working executable.

When parsing an AWK program, **awka** checks to see if each function call in the program is (a) a core builtin function, (b) a call to a user-defined AWK function in the program, or (c) a call to one of the extended builtin functions. The above order of priority is applied, so a user-defined function (b) overrides (c), and (a) overrides (b) to avoid conflicts.

If none of these prove to be true, the function call is written in the code in the format of a user-defined function, even though that function doesn't exist to its knowledge. **Awka** is assuming that by link time you will provide another object file or library that contains the missing function and resolve the call.

So if I pass **awka** the following code:

```
BEGIN { print mymath(3,4) }
```

The call it generates will look like this...

```
mymath_fn(awka_arg2(a_TEMP, _litd0_awka, _litd1_awka))
```

So all we need to do is write the mymath_fn() function, and link it with the awka-generated code, and bingo! AWK has been extended by you, to do what you want. And the only restrictions on what a function like mymath_fn() might do are those imposed by the C language!

So, you write the function, compile it into a library, use it in your AWK program, translate it, link it in, and you're away - its that simple (fingers crossed).

FUNCTIONS AND DATA STRUCTURES

Ok, the first thing to notice is that the function name in the AWK code, *mymath*, has been appended with *_fn* in the C code. This happens with all unresolved AWK function calls (also with user-defined function names, but that doesn't matter here). It's done to avoid unintentional conflicts with functions in other libraries.

The definition of any function is this:-

```
funcname_fn( a_VARARG * )
```

Ugh! What's this a_VARARG thingy? Yes, learned reader, the time has come to get acquainted with the dreaded Awka data structures. Well they're pretty simple actually. The two you need to know about are a_VAR and a_VARARG, and as the latter contains arrays of the former, I'll deal with a_VAR first.

The a_VAR Structure

```
typedef struct {
    double dval;         /* the variable's numeric value */
    char * ptr;          /* pointer to string, array or RE structure */
    unsigned int slen;   /* length of string ptr as per strlen */
    unsigned int allc;   /* space mallocated for string ptr */
    char type;           /* records current cast of variable */
    char type2;          /* special flag for dual-type variables */
    char temp;           /* TRUE if a temporary variable */
  } a_VAR;
```

These are used prolifically throughout the AWK library, and are at the heart of how it manipulates data. Remember, AWK variables are essentially typeless, as they can be cast to number, string or regular expression at your whim throughout a program. The only thing you can't cast to & from is arrays, as a variable is only either an array or a scalar (the other types).

Recall our mymath example earlier. In the AWK code, we had "mymath(3,4)", but the C code was "mymath_fn(awka_arg2(a_TEMP, _litd0_awka, _litd1_awka))".

The numeric value of 3 has been changed to _litd0_awka, and 4 to _litd1_awka. If you run awka with this example program & examine the output, you'll see that both _litd0_awka and _litd1_awka are pointers to a_VAR structures, and each has been set to the appropriate numeric values. Hence, all data passed to our functions will be embodied inside a_VAR's.

Confused? Yes? No? Take heart, it doesn't get much worse, and with a few more examples I hope things should be clearer. Looking at the call to mymath_fn above, you'll notice a call to awka_arg2(). Remember that mymath_fn only takes a pointer to an a_VARARG, so awka_arg2() obviously returns one of these.

What an a_VARARG contains is an array of a_VARs, and an integer showing how many there are in the array - thats all! Don't believe me? Then here's the structure in all its glory:

The a_VARARG Structure

```
typedef struct {
    a_VAR *var[256];
    int used;
  } a_VARARG;
```

The a_VARARG structure gives us an easy means of passing around flexible numbers of a_VARS to functions, much as you'd use vararg in a C program. If you don't know what vararg does and have some time, check the *stdarg* manpage.

So, to conclude, awka_arg2() takes two a_VARs and packages them nicely into an a_VARARG to make life easy for our function. Another thing to note - the a_VARARG function allows up to 256 arguments. No parameters, only arguments, and they always win them! Sorry, on with the serious stuff...

THE MYMATH FUNCTION IMPLEMENTED

So when we come to write mymath_fn, what type of thing should it contain? Ok, lets assume we want mymath to add the two numbers it receives as arguments, then add on the two numbers multiplied, and return the result, ie. (n1+n2)+n1*n2.

Well, here goes...

```
#include <libawka.h>

  a_VAR *
  mymath_fn( a_VARARG *va )
  {
   a_VAR *ret = NULL;
```

```
  if (va->used < 2)
    awka_error("function mymath expecting 2 arguments, only got %d.\n",va->used);

  ret = awka_getdoublevar(FALSE);
  ret->dval = (awka_getd(va->var[0]) + awka_getd(va->var[1])) +
              va->var[0]->dval * va->var[1]->dval;

  return ret;
}
```

Ok, there's not a lot to it, so lets start at the top. You need to include libawka.h, as it defines the data structures plus the whole Awka API that you'll be calling.

The definition of mymath_fn is as described earlier. It will need to return a numeric value, but as we're in AWK (conceptually), this will need to be enclosed in an a_VAR, hence the existence of *ret*.

The incoming a_VARARG can contain any number of a_VAR's - we only care about the first two, so we check to see whether these exist, and if not spit an error through the awka_error function (or you could use your own error handler). When writing your own functions, you'll need to remember that any number of arguments could be passed in, and they could be of any type, so you'll need to check them.

So far, *ret* is NULL, so we need to create a structure to point it to. Better than that, we call awka_getdoublevar(), which gets us a temporary variable, already initialised to contain a numeric value. You guessed it, there's an awka_getstringvar() that we could use if our function was to return a string. The value of FALSE passed to awka_getdoublevar() means that we don't want to be responsible for freeing this structure, but prefer to leave it to libawka's internal garbage collection. I can't see any reason why you'd choose TRUE, but its there just in case.

The next 2 lines do the core stuff. Ok, ret->dval is set, that makes sense. The expression refers to the contents of the a_VARARG->a_VAR array, again this is expected. At first, though, it calls awka_getd() for each of the arguments, but on the next line it references the dval value directly. Why the calls to awka_getd?

Because it can't be sure that the incoming variables are already cast to numbers, so these functions (actually macros) do the casting for us, and return the value of dval after the cast is done. Subsequently, we can look at dval directly as we know its been set to the current numerical value of the variable.

Lastly, we return ret.

COMPILING AND LINKING

Alright, let's get this working. Follow these steps:

1. Create mymath.c with mymath_fn(), exactly as its written above.
2. Create mymath.h containing: a_VAR * mymath_fn(a_VARARG *va);
3. gcc -c mymath.c (or use whatever C compiler you have).
4. awka -i mymath.h 'BEGIN { print mymath(3,4) }' >test.c
5. gcc -I. test.c mymath.o -lawka -lm -o mytest
6. mytest

The output from running mytest should be 19. Magic!

A more comprehensive example is the awkatk library available from the awka website. Hopefully you'll find it helpful, and who knows, you may even use it to write GUI interfaces from AWK!

HOW & WHEN WOULD YOU USE IT?

Obviously, this is intended to extend the limits of the AWK universe, as you could introduce any functionality written in C as a new builtin function within AWK.

There may be complex functions you've written in AWK and use all the time that are just plain inefficient, even using Awka. They're stable, you have the skill to implement them in C, so now you can, and your AWK programs become shorter in the process. It's no longer a choice of C **or** AWK, now

you can migrate sections to C as & when you like.

There are many functions in standard C libraries that AWK doesn't have. Things like strcasecmp(), fread(), cbrt(), and so on. Now you can implement them.

Lastly, I'd love to see Awka have functions to read & write proprietary formats like MS Excel, to communicate with ODBC databases, to perform complex mathematical or scientific operations, to implement true multi-dimensional arrays, to provide Fast Fourier Transform functions - I know its possible. If you do develop something neat like this, it'd be very cool if you were to make it available for everyone to share. Just send an email to andrewsumner@yahoo.com, and I'd be happy to host it on, or link it from the Awka website.

NOTE: KEEP YOUR API FLAT

So you've created quite a few Awka-ELM functions that you've put together into a library. Let's say they calculate the time needed to build the Sydney Harbour Bridge given a volume of manpower and the number of supervisors. Internally, there's quite a few algorithms that take into account strikes by unions, material shortages, and casualties as workers fall off the bridge.

Because of this complexity, within your library functions will need to call other functions. This is fine. What you need to do is not have an API function call another API function, but instead keep any functions they call hidden within the library, and also ensure these internal functions do not use the awka_getdoublevar(), awka_getstringvar() or awka_tmpvar() calls.

Apart from keeping your library structure nice and hierarchical and your API simple, it avoids overloading awka's internal pool of temporary variables. If this pool is overloaded, random chaos will ensue, so please avoid it.

NOTE: REFERENCING GLOBAL VARIABLES

All global variables in your AWK program are accessible by your library functions. Herein lies the potential for great danger, so be careful!

Global variables are, of course, pointers to a_VAR structures, and their name is the same as in the AWK script, with _awk appended. So the variable 'myvar' in the script would be myvar_awk in the translated C code. If you know what the variable name is, you can put an extern declaration of it in your library code then work with it directly, but this may be very restrictive, as it would mean that every script that uses your library would need that variable name reserved. There are other methods.

One of the easiest is with arrays. You can pass them in as arguments to your functions, as their address is passed over rather than a copy of their contents. Scalars are not as easy. Just say our function will work with a global variable, however it expects a string argument to contain the variable name in order to identify which variable to work with - this would make it pretty flexible.

You have available to you the *gvar_struct* variable **_gvar** (both described in **awka-elmref**(5)). This contains the name of every global variable in the script, and its a simple matter to search down the list to find a pointer to the a_VAR structure of the variable you want to use.

NOTE: CUSTOM DATA STRUCTURES

Looking again at the a_VAR structure, you may note that it contains a char * pointer that can reference strings, arrays and regular expressions. There is no reason why you couldn't introduce your own custom data structure and attach it to a global variable within one of your functions, as long as you adhere to the following rules:

1. Don't set the variable to anything in AWK after you set it to your customised value, as libawka will try (and fail) to free the value up, causing all sorts of flow-on problems.
2. Don't use the AWK language to copy or compare this variable to others, even with two variables of the same custom type (ie. custvar1 = custvar2), as libawka will have no idea how the copy should be done, and it will stuff it up. Instead, provide your own copy and comparison functions.
3. If your structures are memory intensive, you may consider providing a method of freeing the structures when they are no longer needed.
4. Document what your data structures and methods do, and how they should be used

in the AWK script. Please, please do this, as it could save you a lot of grief later. If your library becomes publicly available this is especially necessary.

This has been a very brief introduction indeed, but hopefully enough to get you started. I recommend you refer to the *awka-elmref*(5) manpage for a listing of key libawka API functions and data definitions that are available for you to use (but hopefully not abuse). If you have any questions at all, don't be afraid to contact me (andrewsumner@yahoo.com). Put the word "awka" at the front of your message title so I know its not spam.

SEE ALSO

awka(1), **awka-elmref**(5), **gcc**(1)

BUGS

Bound to be plenty. Let me know if you find a bug with the libawka interface, or get stuck with a problem. I am not, though, in any way responsible for bugs that are introduced by your code, nor am I liable for any damages or expenses incurred as a result. Nor am I liable for anything you do using Awka.

I'll help where I can, and I'll usually help debug someone's library if I have a personal interest in it. If you're not sure, try me anyway, the worst I can do is say no, and I might be able to help. I really like folk who send fixes along with bug reports, though. And I love the folk who send cash inducements (at last count, um, zero folk). Oh well, enough rambling, time to finish.

AUTHOR

Andrew Sumner, August 2000 (andrewsumner@yahoo.com).

NAME

awka-elmref - Awka API Reference for use with Awka-ELM libraries.

DESCRIPTION

Awka is a translator of AWK programs to ANSI-C code, and a library (*libawka.a*) against which the code is linked to create executables. Awka is described in the **awka** manpage.

The Extended Library Methods (ELM) provide a way of adding new functions to the AWK language, so that they appear in your AWK code as if they were builtin functions such as substr() or index(). The **awka-elm** manpage contains an introduction to Awka-ELM.

This page lists the available data structures, definitions, functions and macros provided by *libawka.h* that you may use in creating C libraries that link with awka-generated code.

I have broken the page into the following main sections: BASIC VARIABLE METHODS, ARRAY METHODS, BUILTIN FUNCTIONS, I/O METHODS, REGULAR EXPRESSION METHODS. So, without further ado...

BASIC VARIABLE METHODS

Data Structures

a_VAR

```
typedef struct {
  double dval;          /* the variable's numeric value */
  char * ptr;           /* pointer to string, array or RE structure */
  unsigned int slen;    /* length of string ptr as per strlen */
  unsigned int allc;    /* space mallocated for string ptr */
  char type;            /* records current cast of variable */
  char type2;           /* double-typed variable flag, explained later. */
  char temp;            /* TRUE if a temporary variable */
} a_VAR;
```

The **a_VAR** structure is used to store everything related to AWK variables. This includes those named & used in your program, and transient variables created to return values from functions and other operations like string concatenation. As such, this structure is ubiquitous throughout libawka and awka-generated code.

The *type* value is set to one of a number of #define values, described in the Defines paragraph below. Many functions and macros exist for working with the contents of a_VARs - see the Functions & Macros paragraph for details.

a_VARARG

```
typedef struct {
  a_VAR *var[256];
  int used;
} a_VARARG;
```

This structure is typically used to pass variable numbers of a_VARs to functions. Up to 256 a_VARs may be referenced by an a_VARARG, and the *used* value contains the number of a_VARs present.

struct gvar_struct

```
struct gvar_struct {
  char *name;
  a_VAR *var;
};
```

Provides a mapping of the global variable names in an AWK script to pointers to their a_VAR structures.

Internal Libawka Variables

a_VAR * a_bivar[a_BIVARS]
This array contains all the AWK internal variables, such as ARGV, ARGV, CONVFMT, ENVIRON and so on, along with $0 and the field variables $1..$n. *a_BIVARS* is a define, as are the identities of which element in the array belongs to which variable. Again, look for functions that manage these variables rather than working with them directly if possible.

extern struct gvar_struct *_gvar;
This is actually created & populated by the translated C code generated by *awka*, rather than by libawka.a. It is a NULL-terminated array of the gvar_struct structure defined earlier in this page, and contains the names of all global variables in an AWK script, mapped to their a_VAR structures.

Defines

a_VARNUL - the *type* value of an a_VAR if the variable is unused.

a_VARDBL - the *type* value for an a_VAR cast to a number.

a_VARSTR - *type* where the a_VAR has been cast to a string.

a_VARARR - *type* where the a_VAR contains an array.

a_VARREG - *type* where the a_VAR contains a regular expression.

a_VARUNK - *type* where the a_VAR is a string, but could also be a number. Variables populated by getline, the FILENAME variable, and elements of an array created by split(), are all of this special type.

a_DBLSET - for a string a_VAR that has been read in context as a number, the *type2* flag is set to this #define to prevent the string-to-number conversion being done again.

a_STRSET - the opposite of the above. The variable is a number, has been read as a string, hence the value of *ptr* is current, and the *type2* flag is set to this #define.

a_BIVARS provides the number of elements in the a_bivar[] array.

a_ARGC, **a_ARGIND**, **a_ARGV**, **a_CONVFMT**, **a_ENVIRON**, **a_FILENAME**, **a_FNR**, **a_FS**, **a_NF**, **a_NR**, **a_OFMT**, **a_OFS**, **a_ORS**, **a_RLENGTH**, **a_RS**, **a_RSTART**, **a_RT**, **a_SUBSEP**, **a_DOL0**, **a_DOLN**, **a_FIELDWIDTHS**, **a_SAVEWIDTHS**, **a_SORTTYPE** provide indexes to which elements in the a_bivar[] array are for which AWK internal variable.

Functions & Macros

awka_getd(a_VAR *)
This macro calls the awka_getdval() function, appending the calling file & line number

for debug purposes. It read-casts the variable to a number, and returns the double value of the variable. By read-cast, we mean that if the variable is a string it remains so, but *dval* is set, and *type2* is set to a_DBLSET. But if the a_VAR is a regular expression, the re structure is dropped and the variable converted to a number. If you're not sure whether an a_VAR you're about to read is a number, and you want to read it as one, simply call awka_getd(varname) - its the easiest way.

awka_getd1(a_VAR *)
Same as awka_getd, except this will be faster if the a_VAR * is a variable. Do not use this if the a_VAR * is a function call return value, as it'll call the function several times! In this case, use awka_getd() instead.

awka_gets(a_VAR *)
Similar to awka_getd(), this read-casts an a_VAR to a string, and returns the character array pointed to by *ptr*.

awka_gets1(a_VAR *)
Use this where the a_VAR * is a variable, not a function call that returns an a_VAR *, for faster performance.

awka_getre(a_VAR *)
Write-casts the a_VAR * to a regular expression, and returns the pointer to the awka_regexp structure. Write-cast means that the existing contents of the variable are dropped in favour of the new contents.

static char *awka_strcpy(a_VAR *var, char *str)
This function sets *var* to string type, and copies to it the contents of *str*. It returns a pointer to *var->ptr*.

a_VAR *awka_varcpy(a_VAR *va, a_VAR *vb)
This function copies the contents of scalar a_VAR **vb* to scalar a_VAR **va*, and returns a pointer to *va*.

double awka_varcmp(a_VAR *va, a_VAR *vb)
This function compares the contents of the two scalar variables, and returns 0 if the variables are equal, -1 if va is less than vb, or 1 if va is greater. Numerical comparison is used where possible, otherwise string.

a_VAR *awka_vardup(a_VAR *va)
This function creates a new a_VAR *, copies the contents of *va*, and returns a pointer to the new structure.

awka_varinit(a_VAR *)
A macro that takes a NULL a_VAR *, mallocs space for it, and initialises it to a_VARNUL.

void awka_killvar(a_VAR *)
Frees all memory used by the a_VAR, except the structure itself.

static a_VAR * awka_argv()
You can use a_bivar[a_ARGV] directly when reading the value of elements in the array, but when you want to write to the array, use the above function instead, as it will make sure the changes are recognised elsewhere in libawka.

static a_VAR * awka_argc()
You can use a_bivar[a_ARGC] directly when reading its value, but when you want to write to it, use the above function instead, as it will make sure the change is recognised elsewhere in libawka.

ARRAY METHODS

Data Structures & Variables

These are strictly internal to the array module within libawka. If you need functionality other than that provided by the array functions, I recommend creating your own custom array data structures and interface functions, otherwise you could cause serious problems. The structure definitions are too lengthy to list here, and the foolhardy may find them in lib/array.h within the awka distribution.

Defines

a_ARR_TYPE_NULL
The 'type' of an array that has not been initialised, or has been deleted.

a_ARR_TYPE_SPLIT
The 'type' of an array populated by the split() builtin function.

a_ARR_TYPE_HSH
The 'type' of arrays populated within the AWK script, eg. arr["pigs"] = cows.

a_ARR_CREATE
When searching arrays, specifies that an element is to be created if it doesn't already exist in the array.

a_ARR_QUERY
When searching arrays, this will not create a new element if it doesn't already exist.

a_ARR_DELETE
In an array search, this flag will cause the element to be deleted from the array.

Functions

void awka_arraycreate(a_VAR *var, char type);
Allocates an array structure of type *type*, makes *var->ptr* point to it, and sets *var->type* to a_VARARR. The *type* argument may be one of a_ARR_TYPE_NULL, a_ARR_TYPE_SPLIT or a_ARR_TYPE_HSH, according to how the array will be populated.

void awka_arrayclear(a_VAR *var);
Assumes *var* is an a_VARARR, this deletes the contents of the array structure pointed to by *var->ptr*.

a_VAR * awka_arraysearch1(a_VAR *v, a_VAR *element, char create, int set);
Searches array variable *v* for index *element*. If it does not exist, and *create* is a_ARR_CREATE, a new element in the array for this value will be added.

If the element is found (or created) and *create* is not a_ARR_DELETE, the function will return a pointer to the a_VAR for that element. For a_ARR_DELETE, the element will be deleted from the array. The *set* value should be FALSE.

a_VAR * awka_arraysearch(a_VAR *v, a_VARARG *va, char create);
Searches array variable *v* as per awka_arraysearch1(), except that this works with multiple index subscripts (eg, arr[x, y]).

double awka_arraysplitstr(char *str, a_VAR *v, a_VAR *fs, int max);
The AWK builtin split() function. It splits *str* into array variable *v*, based on *fs*, up to *max* number of fields. If *fs* is NULL, then a_bivar[a_FS] will be used. Otherwise *fs* may contain an empty string, a single-character string, or a regular expression. The number of fields created in **v** is returned.

int awka_arrayloop(a_ListHdr *ah, a_VAR *v);
This function implements the "for (i in j)" feature in AWK. You provide *ah*, making sure it is initialised to zeroes.

The best way to understand how to call this function is to type:

```
awka 'BEGIN { for (i in j) x = j[i]; }'
```

and see what is generated as a result. You don't have to understand the a_ListHdr structure or sub-structures to use this function.

int awka_arraynext(a_VAR *v, a_ListHdr *ah, int pos);
Given that *ah* has been populated by a call to awka_arrayloop(), this function copies the (string or integer) element at position *pos* in the list to *v*, then returns *pos*+1, or zero if there are no more elements in the array list.

void awka_alistfree(a_ListHdr *ah);
Frees the last list element in **ah**.

void awka_alistfreeall(a_ListHdr *ah);
Frees all memory held by *ah*, and sets its contents to zero/NULL.

a_VAR * awka_dol0(int set);
The best means of accessing the $0 a_VAR, as it updates its contents with any pending changes. Make *set* zero if you're reading the value of $0, but if you want to set $0, make it 1.

a_VAR * awka_doln(int fld, int set);
This function returns the a_VAR * of the $1..$n variable identified by *fld*, updating the field array with any refreshed $0 contents first if necessary. If you want to read the value of $*fld*, make *set* zero, otherwise it should be 1.

BUILTIN FUNCTIONS

These are documented in lib/builtin.h in the awka distribution. You can call any of the builtin functions as often as you like. Those that return a_VAR's also provide a *keep* flag that, if TRUE, will return a variable that you must free, otherwise they will use a temporary variable that you don't have to worry about freeing, but will be reused elsewhere sooner or later. The functions should be pretty much as you'd expect them, except that many require an a_VARARG as input, and we haven't discussed how to create one - we will now.

a_VARARG * awka_arg0(char);

a_VARARG * awka_arg1(char, a_VAR *);

a_VARARG * awka_arg2(char, a_VAR *, a_VAR *);

a_VARARG * awka_arg3(char, a_VAR *, a_VAR *, a_VAR *);

a_VARARG * awka_vararg(char, a_VAR *var, ...);

These functions populate & return a pointer to an a_VARARG structure. The char argument, if TRUE, will make you responsible for freeing the structure, otherwise it'll be a temporary one that libawka will manage. awka_arg0() will return an empty structure (ie. no args), awka_arg1() will have one a_VAR * in it, and so on. Where you want to put more than four a_VAR *'s inside an a_VARARG, you can call awka_vararg with as many as you like, or if there's seriously a lot, maybe write your own loop of code to populate an a_VARARG - its not rocket science.

I/O METHODS

Data Structures & Variables

_a_IOSTREAM

```
typedef struct {
  char *name;       /* name of output file or device */
  FILE *fp;         /* file pointer */
  char *buf;        /* input buffer */
  char *current;    /* where up to in buffer */
  char *end;        /* end of data in buffer */
  int alloc;        /* size of input buffer */
  char io;          /* input or output stream flag */
  char pipe;        /* true/false */
  char interactive; /* whether from a /dev/xxx stream or not */
} _a_IOSTREAM;
```

extern _a_IOSTREAM *_a_iostream;
extern int _a_ioallc, _a_ioused;

Controls input and output streams used by AWK's getline, print and printf builtin functions. The two int variables record the space allocated in the _a_iostream array, and the number of elements used, respectively. I list this information here in case you wish to create fread, fwrite and fseek functions for awka, as these will need low-level access to the streams.

Functions

a_VAR * awka_getline(char keep, a_VAR *target, char *input, int pipe, char main);

As previously described, *keep* controls whether you want to be responsible for freeing the a_VAR the function returns or not. Moving on, *target* is the a_VAR to hold the line of data to be read (you provide this one). *input* is the name of the input file or command. *pipe* is TRUE if *input* is a command rather than a file, eg. "sort stuff | getline x". *main* should always be false.

If *input* is NULL, getline will try to read from the file identified by a_bivar[a_FILENAME], or from the next element in the a_bivar[a_ARGV] array.

I won't go into detail about awka_fflush, awka_close, awka_printf & so on, as these should be easy enough to understand and use, and the chances are you should use the native C variety anyway where possible.

REGULAR EXPRESSIONS

Ah, now we're in murky water indeed, as awka inherited its RE library from Libc, and treats it like a magical black box that does its bidding. Want my advice? Treating the RE library & structure like a black box is a wise thing to do, as its ugly-looking stuff.

Ok, we know that when an a_VAR has been set to a_VARREG, its *ptr* value will point to an awka_regexp structure. Do we need to know what's in this structure? I don't think so. What we do need are the functions that help us compile and execute regular expressions. Oops, getting ahead of myself. RE's are like C programs, they need to be compiled before they can be used to search strings. This basically is a parsing of the RE pattern into a tree structure that is easier to navigate while searching, and is a one-off task.

awka_getre(a_VAR *)
This macro is the easiest method of creating & compiling a regexp. Providing you've set the a_VAR to the string value of the re pattern, this macro call works a treat.

a_VAR *awka_match(char keep, char fcall, a_VAR *va, a_VAR *rva);
This function is the implementation of AWK's match() function, and is the most simple way of evaluating an RE against a string. *keep* is as previously discussed, *fcall* should be set to TRUE if you want a_bivar[a_RSTART] and a_bivar[a_RLENGTH] to be set, otherwise FALSE, *va* contains the string, and *rva* contains the regular expression. The numerical a_VAR returned is 1 on success, zero on failure.

I was going to describe the lower-level methods of compiling and matching against RE's, but when I looked at it, there seemed to be a lot of complexity for no real gain in functionality. All you get is the ability to avoid using a_VAR structures to manage the regular expressions, and honestly I don't see what you'd gain from this given how much more complexity you'd have to deal with.

NOTES

I haven't described all of the functions available in libawka.h, not by any means. But I have tried to avoid functions that are really only meant for internal use, or that are only needed by translated code and should be done in other ways by library code. In the same way I've avoided describing structures that were intended to remain privy to a module within libawka, and you really shouldn't need to tamper with them.

Any questions at all, or suggestions for improving this page, let me know via andrewsumner@yahoo.com. Make sure you preface any message title with the word "awka" so I know its not spam.

SEE ALSO

awka(1), **awka-elm**(5).