NAME

awk_api — API for accessing AWK interpreter internals

SYNOPSIS

```
#include <stdlib.h>
#include "awk_api.h"

typedef ... awknum_t;               /* numeric values */
typedef ... awk_val_t;              /* awk values, number or string */
typedef ... awk_array_cookie_t;     /* hook for array access */
typedef ... awk_elem_t;             /* single array element */
typedef ... awk_array_t;            /* a flattened-out awk array */
typedef ... awk_param_t;            /* a function parameter */

typedef void (*awk_extenfunc_t)(awk_val_t *retval,
               size_t nparams, awk_param_t *params);

const awk_val_t *awk_get_elem(awk_array_cookie_t array,
                              const char *indexval, size_t indexlen);
size_t awk_get_elem_count(awk_array_cookie_t array);
awk_array_t *awk_get_array(awk_array_cookie_t array);

const char *awk_add_C_func(const char *name, awk_extenfunc_t funcp);
void awk_atexit( void (*funcp)(void *arg0), void *arg );
```

DESCRIPTION

This document describes the *awk* internals API. It is for use by extension functions that wish to be dynamically linked into any *awk* interpreter that supports this API.

Design Goals

After discussion with other *awk* maintainers, it was decided that simplicity and smallness should be primary goals for this API. In particular, the main point of this API is for *awk* to call out to C, *without* providing any access to facilities within the interpreter. Thus, this API purposely does not provide facilities to access Awk variables, built-in functions, user-defined functions, record management, control flow, or regular expression matching.

String Values

In this API, all string values may be assumed to have a terminating 0 byte. *However*, C functions should not assume that embedded 0 bytes are absent, and should always perform operations based on the provided string length, instead of naively using *strlen*(3) on any character pointer.

Any string values created by an extension function should include the terminating byte, although length values should only include the actual number of characters in the string.

Memory Ownership

Furthermore, all pointers to variable names, variable string values, and array index string values should be treated as *read-only*; it is assumed that they point into internal data structures, and extension functions should not attempt to modify the data.

VALUES

Awk values are described by the following structure:

```
/* A numeric value. At least a C double */
typedef double awknum_t;        /* typedef'ed against future changes */

/* Bits for use in flags in a value.  */
typedef enum {
    NUM     = 1,    /* type is numeric: mutually exclusive to STR */
    STR     = 2,    /* type is string: mutually exclusive to NUM */
```

```
    NUMCUR  = 4,    /* optimization: numeric value is current */
    STRCUR  = 8,    /* optimization: string value is current */
} awk_val_flags;

/* Return printable representation of value flags. */
extern const char *awk_val_flags2str(int flags);

/*
 * An awk value.  This has a type of either numeric or string.
 * The "other" value may also be current, as an optimization.
 */
typedef struct {
    short flags;            /* bitwise OR of awk_val_flags values */
    awknum_t numval;        /* numeric value */
    const char *strval;     /* string value */
    const size_t strlen;    /* string length */
} awk_val_t;
```

Values are used directly in function parameters, and in array elements.

ARRAYS

Awk arrays can potentially be quite large. For this reason, extension functions receive a ‘‘cookie’’ representing just a handle for an array. Extension functions can lookup a single index in an array, or request that the array be ‘‘flattened’’ into a C array representing every element.

```
/*
 * For efficiency, awk arrays are not flattened unless explicitly
 * requested.  Thus, C functions initially receive just an
 * array cookie.
 */
typedef void *awk_array_cookie_t;

/*
 * This returns a constant pointer to the given element value
 * if it's in the array, NULL otherwise.
 */
const awk_val_t *awk_get_elem(awk_array_cookie_t array,
                              const char *indexval, size_t indexlen);

/*
 * This returns the number of elements in the array.
 */
size_t awk_get_elem_count(awk_array_cookie_t array);
```

When an array is flattened, each element looks like this:

```
typedef enum {
    /* These flags are mutually exclusive! */
    DELETED     = 1,    /* element removed by called C function */
    REPLACED    = 2,    /* element *value* changed by called C function */
} awk_elem_flags;

/* Return printable representation of elem flags. */
extern const char *awk_elem_flags2str(int flags);

/* Single awk array elements */
typedef struct {
```

```
    short flags;            /* one of the values in awk_elem_flags */
    const char *indexstr;   /* index string, array indices are *always* strings */
    size_t indexlen;        /* length thereof */
    awk_val_t value;        /* original element value */
    awk_val_t newvalue;     /* new value if REPLACED is in flags */
} awk_elem_t;
```

Initially, the `flags` member is set to zero. If an extension function wishes to delete an element, it should set `flags` to `DELETED`. To replace an element's *value*, the extension function should set `flags` to `REPLACED` and fill in the `newvalue` element of the structure. (The original `value` element should not be touched; the interpreter will release its resources.) If a string value is supplied, the text of the string *must* come from *malloc*(3), and the interpreter will accept further responsibility for the memory.

A full array is represented by this structure:

```
/* Awk arrays */
typedef struct {
    size_t elem_count;      /* number of elements in the array */
    awk_elem_t *elems;      /* dynamically allocated */
    size_t new_count;       /* if set to non-zero, brand new elements were added */
    awk_elem_t *newelems;   /* these are the new elements, each one has flag == 0 */
} awk_array_t;

/*
 * This "flattens" the array represented by the cookie into
 * an awk_array_t for access by the C function.
 */
awk_array_t *awk_get_array(awk_array_cookie_t array);
```

The `awk_get_array()` function ''flattens'' the array referred to by the `array` cookie into a linear array of C structures. If an extension function wishes to add elements to the array, it should set `newcount` to the appropriate value, and set `newelems` to array of `awk_elem_t` structures, each of which is filled in appropriately. *All* storage for the new elements, including the array of structures, the index string values, and any element string values, must also come from *malloc*(3), and the interpreter accepts further responsibility for the memory.

FUNCTION PARAMETERS

At the Awk level, a parameter may receive its value from one of three sources:

1. A scalar variable. The value of the scalar is passed by value. C extension functions receive such a parameter but should not modify the contents of the value.
2. An array. In Awk, the array is passed by reference. C extension functions receive an array cookie which may be used to access or modify the array, as described in the previous section.
3. An uninitialized variable. If an Awk function uses a parameter derived from such a variable as an array, the original variable becomes an array and cannot then be used as a scalar.

   On the other hand, if an Awk function uses a parameter derived from an uninitialized variable as a scalar, the original variable cannot then be used as an array.

   This API allows C functions to provide the same semantics.

The following structure defines parameter values:

```
/* Bits for use in flags in a function parameter */
typedef enum {
    UNTYPED         = 1,    /* value is not typed in awk program */
    SCALAR          = 2,    /* value was or is changed to scalar */
    ARRAY           = 4,    /* value was or is changed to array */
```

```
    ARRAY_CHANGED   = 8,    /* array was modified */
} awk_param_flags;

/* Return printable representation of elem flags. */
extern const char *awk_elem_flags2str(int flags);

typedef struct {
    short flags;                    /* bitwise OR of awk_param_flags */
    union {
        awk_val_t p_val;            /* by value value if SCALAR */
        awk_array_cookie_t p_arr;   /* by reference array if ARRAY */
    } u;
} awk_param_t;
#define param_val   u.p_val
#define param_arr   u.p_arr
```

A C extension function that receives an `UNTYPED` value *must* OR in the type flag if it changes it. This allows the interpreter to realize that the type was changed.

ADDING C EXTENSION FUNCTIONS

There are two steps involved in adding an extension function. The first is to write the function itself. The second is to install it into the interpreter.

Writing C Extension Functions

An extension function should have the following signature:

```
void myextension(awk_val_t *retval, size_t nparams, awk_param_t *params);
```

I.e., the first parameter is pointer to a return value structure, which the extension function will fill in appropriately. If a string value is returned, the memory must come from *malloc*(3), and the interpreter will assume responsibility for the memory.

The second value is the number of actual parameters passed at the time of the function call. This may be more or less than the number of arguments that the function expects; it is up to the function to issue any diagnostics or return an error value of some kind if this is a problem.

The third value is pointer to an array of parameter structures, as described in the previous section. For convenience, the `awk_api.h` header file provides this `typedef`:

```
typedef void (*awk_extenfunc_t)(awk_val_t *retval,
                size_t nparams, awk_param_t *params);
```

Installing An Extension Function

The following function adds an extension function into the interpreter:

```
const char *awk_add_C_func(const char *name,       /* function name */
                           awk_extenfunc_t funcp,  /* pointer to C function */
                           size_t nargs);          /* expected number of arguments */
```

The first parameter is a regular C string with the name of the function. The name must follow Awk naming conventions in order for the function to be called. It is safe to use a C string constant for this parameter; if the memory came from *malloc*(3), the interpreter does *not* accept responsibility for it!

The second parameter is a pointer to the C extension function.

The third parameter is the *expected* number of arguments.

The return value is `NULL` if the function was correctly installed. Otherwise it is a pointer to a string than can be printed directly. The error message will have already been translated if the interpreter supports message translation.

The `awk_add_C_func` function should be called from a separate function named `dl_onload` which will be called by the interpreter when loading the extension module. Any other setup functions may be

called from this function as well.

Hook For Cleanup Actions

The API provides the following ‘‘hook’’ for cleanup actions:

```
void awk_atexit( void (*funcp)(void *arg0), void *arg );
```

The first parameter is a pointer to a callback function returning `void` and accepting a `void *` parameter. The second function is a pointer to arbitrary data that will be passed to the callback function when it's called. Provide `NULL` if you do not wish to pass any data.

The callback functions are called in Last In First Out (LIFO) order, *after* the Awk program's `END` block has finished executing, but before the interpreter itself calls *exit*(3).

MEMORY MANAGEMENT

It is the intent of this design that memory management responsibilities are clear: *Don't touch my memory; I accept responsibility for memory you give me*. All pointers into Awk values and arrays *must* be treated as read-only. Changed array element values must be dynamically allocated, as must all memory for added array elements.

Furthermore, for use with *gawk*, C string functions should not be used, since there may be arbitrary binary data, including ASCII NUL characters, embedded in array indices and variable string values.

SEE ALSO

Effective AWK Programming

REQUEST FOR COMMENTS

Should the `awk_` prefix be removed?

Should all the constants (`enum` values) have some sort of prefix too?

Should the API use `wchar_t` instead of `char` for text values?

Should ‘‘constructor’’ functions be provided for building the various structures and doing the dynamic memory management?

Is returning a `char *` message indicating the error better or worse than trying to define a set of `errno`-style values? (The idea is that each interpreter implementing the API is then free to have its own set of error returns.)

Should it be possible to force the interpreter to clean up and exit; for example if an extension function could not be installed? For the module to call *exit*(3) itself seems impolite.

What is missing from this interface?

What is extraneous in this interface and should be removed?

What other comments do you have?

Initial discussion should take place in the `comp.lang.awk` USENET newsgroup. You may also email comments to `arnold@skeeve.com`. Thanks.