AFF lattice data storage format
Version 2

Andrew V. Pochinsky, Sergey N. Syritsyn
MIT CTP, Cambridge, MA

December 4, 2007

This document describes the AFF data storage format. The AFF is hierarchical data format, efficient both in space and access time for storage of multiple small amounts of data.

Contents

1 Purpose of AFF

Lattice calculations produce a lot of data. At the analysis stage the data usually consists of enormous number of small pieces, for example, correlator values for each operator, each momentum and each in/out state. Present approach is to store each piece[1] as text in a separate file which has a fully descriptive name. Although this is both convenient for analysis and accessible by a text editor, text format leads to a significant space overhead. In addition, some file systems, e.g., NFS and PVFS, are slow when accessing many small files. Storing all data in an XML file leads to even greater space overhead; extraction of a single data item requires parsing and validating the whole XML file.

AFF data storage format is aimed to replace this data storage scheme. AFF organization is aimed at optimization of data random read access. We suggest to store all data related to a configuration or to an ensemble of configurations in the same file. To navigate an AFF data file, we introduce the system of hierarchical keys. Data is stored in platform-independent, binary form. To assure the data validity, both data and meta information is checked against stored MD5 checksum.

2 Suggested AFF usage

We suggest replacing the output of ADAT strippers with an AFF file format. Complicated file names are replaced with hierarchical key names. The actual set of key names must be convenient for both interactive use and scripts for automatic data processing.

Scripts access the data in an AFF file through a command line utility which gets the data from a file and outputs the required data in an appropriate (e.g., text) form. C/C++ analysis codes access the data through the AFF library, which returns data in the form appropriate for a given platform.

Interactive browsing and modification of an AFF file is done through a command line utility. The utility allows searching and printing the keys, printing data, merging files, insertion of data, and deleting entries. Conversion from XML to ADAT is possible, but only for XML files with unique keys.

3 Command line utility

The command line utility, `lhpc-aff` allows one to manipulate AFF files from the shell. It contains several tools and has a built-in help for each of them.

4 Platform-independent data

Both data and meta information is stored in platform-independent format. Information on bit size of numbers is written in the header of an AFF file. All

[1] As it is done by ADAT stripping utilities

integer numbers are stored in a big-endian form. Double precision numbers are stored in a portable binary format; the parameters of the floating point representation are stored in the file providing enough information to restore `double` numbers on a machine of any reasonable architecture. A complex number is stored as a sequence of two double precision numbers, first the real part, and second the imaginary part.

Table 1: Numeric data types

Type	Size, bytes	Encoding	Comment
Void	0	1	Empty node
Char	1	2	String(array of chars)
Int	4	3	32-bit integer
Double	8	4	double precision real number
Complex	16	5	double precision complex number

5 Data file organization

An AFF file represents data organized as a tree structure. It starts at a *root key*, which may have multiple subkeys. Each subkey of a given key must have a unique name.

Each subkey may have data associated with it, which is an arbitrary length array of any predefined elementary types. Single number is represented as an array of length 1. Possible data types are listed in table 1.

The data in AFF is named using hierarchical names, called *keys*. The namespace organization and semantics of the keys is very close to UNIX file names. A data key is a sequence of subkeys, which we write here as a UNIX file name: `/key1/key2/.../keyN`. The top node in an AFF file is its *root*, called `/`. Part of a key between consecutive slashes is called a *subkey*. To simplify transitions between AFF and XML, we restrict the character set allowed in subkeys to the following grammar (this is a subset of XML names):

$$\begin{aligned} \langle subkey \rangle &::= (\langle Letter \rangle | \texttt{\_} | \texttt{:}) | \langle nameChar \rangle^* \\ \langle nameChar \rangle &::= \langle Letter \rangle | \langle Digit \rangle | \texttt{.} | \texttt{-} | \texttt{\_} | \texttt{:} \end{aligned}$$

The subkeys are case-sensitive as the are in XML.

6 Data file layout

An AFF file has:

- a header, describing the numeric storage format, tables and data position information and checksums; it is placed in the beginning of the file;
- a symbol table, storing all key names;
- a tree table, storing all nodes of key tree;
- a data section.

Each section may be located anywhere in the file. The section positions are stored in a header. The AFF file starts with a header, then there is usually a data section, and symbol and tree tables are in the end of the file. It should be noted that this order of sections in the AFF file is not mandated, a file with arbitrary placed sections is valid (even if they overlap.)

6.1 Header

There are two versions of the file format now. The current version of the library reads both formats but writers only version 2 AFF files.

Table 2: Header layout

	Size, bytes (V1)	Size, bytes (V2)
Signature	32	32
Data header	32	40
Symbol table header	32	40
Tree header	32	40
Header MD5 sum	16	16
Total	144	168

Signature strings for a historical version of the file format and the current version are given in table 3.

6.2 Symbol table, tree, and data headers

All three section headers have the same format described in table 5. The section offset and size is stored in big-endian order regardless of the machine endianess. Version 2 adds a number of records into each section header to make reading of the AFF data faster. The current version of the library reads both V1 and V2 files.

Table 3: Signature layout

	Size, bytes
File version string, null-terminated	21
Bits in `double`	1
Radix of `double`	1
Bits in `double` mantissa	1
Max exponent in `double`	2
Negative min exponent in `double`	2
Header size in bytes	4
Total	32

Table 4: Version strings

Version	Signature string
V1	`"LHPC AFF version 1.0"`
V2	`"LHPC AFF version 2.0"`

6.3 Symbol table

The symbol table is a list of strings separated by a null char. The string stored in the symbol table are implicitly numbered starting with zero. This ordering is used in the tree table below to refer to subkeys of the nodes.

6.4 Tree table

The AFF file tree is represented by a table of entries. Each entry describes one node in a tree. Nodes without data have type `affNodeVoid` and are stored according to table 6. All other nodes are stored according to table 7. Types of the nodes are encoded according to table 1. The root node is not stored in the tree table, as it always has itself as a parent and an empty name, and there is no data stored in it. Other nodes are implicitly numbered starting with `1`. These numbers are used in the parent node fields to refer to node's parent. A proper tree table describes a tree, e.g., every node has a parent and there is no cycles.

Table 5: Symbol table, tree, and data header layout

	Size, bytes (V1)	Size, bytes (V2)
Offset	8	8
Size in bytes	8	8
Number of records	0	8
Section MD5 sum	16	16
Total	32	40

Table 6: Void tree entry layout

	Size, bytes
Type	1
Parent node Id	8
Node name Id (ref. to symbol table)	4
Total	13

7 AFF library interface

AFF library is written in C and can be used by including a library header file `lhpc-aff.h`. There is also `lhpc-aff-config` utility that allows one to obtain proper flags and libraries needed by AFF. The library uses global names starting with `aff` in all case combinations. Not all such names may be described in the present specification. It is illegal to rely on behavior of undescribed functions, data and types.

The data types used by the AFF library interface are listed in table 8. All structures are opaque so that the interface serves as an abstraction barrier between the implementation and the application codes. The only exception is `struct AffMD5_s`.

The library does not contain any global variables and does not call any thread-unsafe functions. If a mutlithreaded program does not try to access the same AFF object from different threads without proper locking, it is safe to use the library with POSIX threads.

The interface consists of three parts.

- Library information routines provide an interface to common features.
- Writer routines help to write data into AFF files.
- Reader routines are used to read data from AFF files and to navigate through the key.

Table 7: Non-vod tree entry layout

	Size, bytes
Type	1
Parent node Id	8
Node name Id (ref. to symbol table)	4
Size of stored array	4
Offset of stored data	8
Total	25

Table 8: AFF interface opaque types.

`struct AffWriter_s`	A handler of an AFF file opened for writing
`struct AffReader_s`	A handler of an AFF file opened for reading
`struct AffTree_s`	A handler of an AFF tree
`struct AffNode_s`	A handler of an AFF tree node
`struct AffSTable_s`	A handler of an AFF symbol table
`struct AffSymbol_s`	A symbol created and stored by the symbol table
`struct AffMD5_s`	MD5 sum state
`enum AffNodeType_e`	Type of the data stored in a node

7.1 Library information

7.1.1 `aff_version()`

Synopsis

`const char *aff_version (void);`

Description

Returns a string identifying the library version.

Return Value

A non-`NULL` string.

7.1.2 `aff_name_check()`

Synopsis

`int aff_name_check (const char *name);`

Description

Check that `name` satisfies the constraints of section 5 and returns a non-zero value if it does not.

Return Value

Zero if `name` is a permissible name, a non-zero value otherwise.

7.2 Writers

7.2.1 aff_writer()

Synopsis

`struct AffWriter_s *aff_writer (const char *fname);`

Description

Allocate a writer, and initialize it. Open a file for writing, initialize empty tables. If the file already exists, it is removed first. To query the status of `aff_writer()` one calls `aff_writer_errstr()` on the result. If `aff_writer_errstr()` returns `NULL`, the object has been successfully created, otherwise `aff_writer_errstr()` returns a description of the error. Any pointer returned from `aff_writer()` should be passed to `aff_writer_close()` to free resources.

Return Value

Return a pointer to a `struct AffWriter_s`. The status must be checked by calling `aff_writer_errstr()`.

7.2.2 aff_writer_close()

Synopsis

`const char *aff_writer_close (struct AffWriter_s *aff);`

Description

Finalize writing, calculate MD5 sums, write all meta tables and the header, and close the file.

Return Value

Return `NULL` on success, and a pointer to an error string on failure.

7.2.3 aff_writer_errstr()

Synopsis

`const char *aff_writer_errstr (struct AffWriter_s *aff);`

Description

Return a description of the error associated with the writer object. AFF implements latching errors: if an error occurs on a writer object, this object will signal errors on all subsequent calls. The first error message is stored in the object and is accessible via `aff_writer_errstr()` call.

Return Value

Return the string describing the error recorded in the writer object, or `NULL` if there were no errors.

7.2.4 aff_writer_stable()

Synopsis

```
struct AffSTable_s *aff_writer_stable
          (struct AffWriter_s *aff);
```

Description

Get the pointer the symbol table of the writer.

Return Value

The pointer on success, or `NULL` if the writer is not initialized.

7.2.5 aff_writer_tree()

Synopsis

```
struct AffTree_s *aff_writer_tree (struct AffWriter_s *aff);
```

Description

Get the pointer to the tree table of the writer

Return Value

The pointer on success, or `NULL` if the writer is not initialized.

7.2.6 aff_writer_root()

Synopsis

```
struct AffNode_s *aff_writer_root (struct AffWriter_s *aff);
```

Description

Get the handler to the root node. Any initialized writer always have a root node, even if it contains no data.

Return Value

The pointer on success, or `NULL` if the writer is not initialized.

7.2.7 aff_writer_mkdir()

Synopsis

```
struct AffNode_s *aff_writer_mkdir (struct AffWriter_s *aff,
          struct AffNode_s *dir, const char *name);
```

Description

Create a new subkey `name` in the key node `dir` with type `affNodeVoid` (no associated data). The type may be changed at most once later. The function calls `aff_name_check()` to check that `name` is a legal name and reports an error if it is not.

Return Value

Return the pointer to the new key node on success, and `NULL` on failures, i.e. the writer is not initialized, the name already exists, or not enough memory.

7.2.8 `aff_node_put_`*type*`()`

Synopsis

```
int aff_node_put_char (struct AffWriter_s *aff,
          struct AffNode_s *n, const char *d, uint32_t s);
int aff_node_put_int (struct AffWriter_s *aff,
          struct AffNode_s *n, const uint32_t *d, uint32_t s);
int aff_node_put_double (struct AffWriter_s *aff,
          struct AffNode_s *n, const double *d, uint32_t s);
int aff_node_put_complex (struct AffWriter_s *aff,
          struct AffNode_s *n, const double _Complex *d,
          uint32_t s);
```

Description

Put an array `d` of *type* of size `s` into AFF file `aff` in the key node `n`. *Type* may be `char`, `int`(32 bits), `double` or `complex`.

Return Value

Return zero on success, and non-zero on failure.

7.3 Reader

7.3.1 `aff_reader()`

Synopsis

```
struct AffReader_s *aff_reader (const char *file_name);
```

Description

Allocate a reader and initialize it. Open a file for reading, read all tables. To check the status of `aff_reader()`, call `aff_reader_errstr()`. `aff_reader_errstr()` returns `NULL` on success, or a problem description otherwise. Any pointer returned by `aff_reader()` must be passed later to `aff_reader_close()` to free resources.

Return Value

Return a pointer to `struct AffWriter_s`. The status must be checked by calling `aff_writer_errstr()`.

7.3.2 `aff_reader_close()`

Synopsis

```
void aff_reader_close (struct AffReader_s *aff);
```

Description

Close a file, deallocate a reader and all its tables.

7.3.3 aff_reader_errstr()

Synopsis

```
const char *aff_reader_errstr (struct AffReader_s *aff);
```

Description

Get an error string from the last failure.

Return Value

Return a pointer to a string, or `NULL` if no errors have occurred.

7.3.4 aff_reader_stable()

Synopsis

```
struct AffSTable_s *aff_reader_stable
         (const struct AffReader_s *aff);
```

Description

Get reader's symbol table.

Return Value

Return a pointer to the symbol table, or `NULL` if `aff` is not initialized.

7.3.5 aff_reader_tree()

Synopsis

```
struct AffTree_s *aff_reader_tree (struct AffReader_s *aff);
```

Description

Get the reader's tree table.

Return Value

Return a pointer to the tree table, or `NULL` if `aff` is not initialized.

7.3.6 aff_reader_root()

Synopsis

```
struct AffNode_s *aff_reader_root (struct AffReader_s *aff);
```

Description

Get the root node handler of the reader. Root node is always defined, even if the reader is empty.

Return Value

Return a pointer to the root node handler, or `NULL` if `aff` is not initialized.

7.3.7 aff_reader_chdir()

Synopsis

```
struct AffNode_s *aff_reader_chdir (struct AffReader_s *aff,
          struct AffNode_s *dir, const char *name);
```

Description

Get the handler to the subkey `name` in the key node `dir`. If the node does not exist, an error will be set in the reader object. Note that this function should not be used to probe for presence of a subkey because of failure it will render the reader unusable.

Return Value

Return a pointer to the handler or `NULL` if it does not exist or there is other failure.

7.3.8 aff_node_get_*type*()

Synopsis

```
int aff_node_get_char (const struct AffReader_s *aff,
          const struct AffNode_s *n, char *d, uint32_t s);
int aff_node_get_int (const struct AffReader_s *aff,
          const struct AffNode_s *n, int32_t *d, uint32_t s);
int aff_node_get_double (const struct AffReader_s *aff,
          const struct AffNode_s *n, double *d, uint32_t s);
int aff_node_get_complex (const struct AffReader_s *aff,
          const struct AffNode_s *n, double _Complex *d,
          uint32_t s);
```

Description

Get an array of *type* of size `s` from AFF file `aff` in the key node `n` and store it to `d`. Type may be `char`, `int`(32 bits), `double` or `complex`. If the data type does not match, an error will be set in the reader object. The size `s` may differ from the size of the node. If `s` is smaller than the node size, `d` will receive the initial portion of the node data. If `s` is larger than the node data, its initial portion of `d` will be filled with the node data. Values in the rest of the buffer are unspecified in this case.

Return Value

Return zero on success, and non-zero on failure. An failure causes an error to be stored in the reader object.

8 AFF low level interfaces

The rest of AFF provides low level access to the library structures. Some of them are exported only because they are perceived to be generally useful, other are needed for non-trivial manipulation with the AFF objects. The gentle User is advised to treat the functions below with respect.

8.1 Reader and writer tree navigation

8.1.1 `aff_node_foreach()`

Synopsis

```
void aff_node_foreach (struct AffNode_s *n,
         void (*proc)(struct AffNode_s *child, void *arg),
         void *arg);
```

Description

Call function `proc` for each child of the node `n`, and transfer `arg` as an argument. If `n` is `NULL` nothing is done.

8.1.2 `aff_node_id()`

Synopsis

```
uint64_t aff_node_id (const struct AffNode_s *tn);
```

Description

Get 64-bit node ID.

Return Value

Return the node ID. If `tn` is `NULL` return a special value with all bits set.

8.1.3 `aff_node_name()`

Synopsis

```
const struct AffSymbol_s *aff_node_name
         (const struct AffNode_s *n);
```

Description

Get the key name associated with the node.

Return Value

Return a pointer to a string containing key name. The string is internal to the reader(writer) and must not be freed. If `n` is `NULL`, return `NULL`.

8.1.4 `aff_node_parent()`

Synopsis

```
struct AffNode_s *aff_node_parent (const struct AffNode_s *n);
```

Description

Get the handler of node's parent. The parent of the root node is the root itself.

Return Value

Return the pointer to the handler of parent node. If `n` is `NULL`, return `NULL`.

8.1.5 `aff_node_type()`

Synopsis

```
enum AffNodeType_e aff_node_type (const struct AffNode_s *n);
```

Description

Determine the type of data stored in node `n`.

Return Value

Return type of data. If `n` is `NULL`, return `affNodeInvalid`.

8.1.6 `aff_node_size()`

Synopsis

```
uint32_t aff_node_size (const struct AffNode_s *n);
```

Description

Get the size of the data array stored in the node `n`.

Return Value

Return size of data in data type units. Return zero if `n` is `NULL`.

8.1.7 `aff_node_offset()`

Synopsis

```
uint64_t aff_node_offset (const struct AffNode_s *tn);
```

Description

Get the 64-bit file offset of the stored data of node `tn`.

Return Value

Return the byte offset of data. Return zero if `tn` is `NULL`.

8.1.8 `aff_node_assign()`

Synopsis

```
int aff_node_assign (struct AffNode_s *node,
          enum AffNodeType_e type, uint32_t size,
          uint64_t offset);
```

Description

Assign `type` to the node `node`.

Return Value

Return zero on success, and non-zero on failure.

8.1.9 aff_node_*chdir*()

Synopsis

```
struct AffNode_s *aff_node_chdir (struct AffTree_s *tree,
          struct AffSTable_s *stable, struct AffNode_s *n,
          int create, const char *p);
struct AffNode_s *aff_node_cda (struct AffTree_s *tree,
          struct AffSTable_s *stable, struct AffNode_s *n,
          int create, const char *p[]);
struct AffNode_s *aff_node_cdv (struct AffTree_s *tree,
          struct AffSTable_s *stable, struct AffNode_s *n,
          int create, va_list va);
struct AffNode_s *aff_node_cd (struct AffTree_s *tree,
          struct AffSTable_s *stable, struct AffNode_s *n,
          int create, ...);
```

Description

`aff_node_chdir` returns the subkey of node `n` in the `tree` with name `p`. `aff_node_cda`, `aff_node_cdv`, `aff_node_cd` descend the tree into subkeys with names transferred as `NULL`-terminated array, `va_list` and a `NULL`-terminated argument list respectively. If `create` is non-zero, all absent directories are created.

Return Value

Returns the handler of the target key on success. Returns `NULL` if the target key is absent and `create` is zero, or attempt to create keys failed.

8.2 Tree data structure

8.2.1 aff_tree_init()

Synopsis

```
struct AffTree_s *aff_tree_init (struct AffSTable_s *stable,
          uint64_t size);
```

Description

Allocate and initialize an AFF tree structure with only one node, which is the root. The name of the root is an empty string `""`. Previously allocated `stable` is provided to keep associated key data in. The initial size of the tree is supplied by `size` which will be adjusted if unreasonable. If `size` is zero, a default value will be used.

Return Value

Return a pointer to a new AFF tree, or `NULL` if allocation failed.

8.2.2 aff_tree_fini()

Synopsis

```
void *aff_tree_fini (struct AffTree_s *tree);
```

Description

Free AFF data structure.

Return Value

Return `NULL`. This helps with the following programming pattern:

`tree = aff_free_fini(tree);`

– clean up the tree and guard against stray accesses by setting it to `NULL`.

8.2.3 aff_tree_foreach()

Synopsis

```
void aff_tree_foreach (const struct AffTree_s *tree,
         void (*proc)(struct AffNode_s *node, void *arg),
         void *arg);
```

Description

Call function `proc` for each node of the tree in order of their ID numbers and pass `arg` as the argument. If `tree` is `NULL`, nothing is done.

8.2.4 aff_tree_print()

Synopsis

```
void aff_tree_print (struct AffTree_s *tree);
```

Description

Print the AFF tree for debug.

8.2.5 aff_tree_root()

Synopsis

```
struct AffNode_s *aff_tree_root (const struct AffTree_s *tree);
```

Description

Get the root of the `tree`. The root is always present.

Return Value

Return a pointer to the root, or `NULL` if `tree` is `NULL`.

8.2.6 `aff_tree_lookup()`

Synopsis

```
struct AffNode_s *aff_tree_lookup
          (const struct AffTree_s *tree,
          const struct AffNode_s *parent,
          const struct AffSymbol_s *name);
```

Description

Find the child of node `parent` with name `name`.

Return Value

Return a pointer to the child node handler, or `NULL` if `tree` is `NULL`or no such child is found.

8.2.7 `aff_tree_index()`

Synopsis

```
struct AffNode_s *aff_tree_index (const struct AffTree_s *tree,
          uint64_t index);
```

Description

Get the node handler by its index. The index starts from zero, which is reserved for the root node.

Return Value

Return a pointer to the node handler, or `NULL` if `tree` is `NULL` or no such node is found.

8.2.8 `aff_tree_insert()`

Synopsis

```
struct AffNode_s *aff_tree_insert (struct AffTree_s *tree,
          struct AffNode_s *parent,
          const struct AffSymbol_s *name);
```

Description

Insert a child with name `name` to the node `parent`.

Return Value

Return a pointer to the new child node handler, or `NULL` if such node have already been present, `tree` is `NULL` or the insertion failed.

8.3 Symbol table

8.3.1 `aff_stable_init()`

Synopsis

```
struct AffSTable_s *aff_stable_init (uint64_t size);
```

Description

Allocate and initialize an empty symbol table. Suggested initial table size is `size`. The library will adjust the size if unreasonable. If `size` is zero, a default value will be used.

Return Value

Return a pointer to a new symbol table, or `NULL` on failure.

8.3.2 `aff_stable_fini()`

Synopsis

```
void *aff_stable_fini (struct AffSTable_s *st);
```

Description

Free a symbol table.

8.3.3 `aff_stable_print()`

Synopsis

```
void aff_stable_print (const struct AffSTable_s *st);
```

Description

Print symbol table for debug.

8.3.4 `aff_stable_lookup()`

Synopsis

```
const struct AffSymbol_s *aff_stable_lookup
         (const struct AffSTable_s *st, const char *name);
```

Description

Lookup a symbol in the table by its string name

Return Value

Return a pointer to symbol, or `NULL` if there is no such symbol or `st` is zero.

8.3.5 `aff_stable_index()`

Synopsis

```
const struct AffSymbol_s *aff_stable_index (const
         struct AffSTable_s *st, uint32_t index);
```

Description

Lookup a symbol in the table by its index. The index starts from zero.

Return Value

Return a pointer to the symbol, or `NULL` if there is no such symbol or `st` is `NULL`.

8.3.6 aff_stable_insert()

Synopsis

```
const struct AffSymbol_s *aff_stable_insert
        (struct AffSTable_s *st, const char *name);
```

Description

Insert a new string into the symbol table. The string is duplicated by the library to allow the user to free space used by `name` without breaking the stable.

Return Value

Return a pointer to the new symbol, or a pointer to the symbol with the same string inserted before. Return `NULL` if `st` is `NULL`.

8.3.7 aff_stable_foreach()

Synopsis

```
void aff_stable_foreach (const struct AffSTable_s *st,
        void (*proc)(const struct AffSymbol_s *sym,
        void *arg), void *arg);
```

Description

Call the function `proc` for each symbol in the table in order of their index passing `arg` as an argument. If `st` is zero, nothing is done.

8.4 Symbols

8.4.1 aff_symbol_name()

Synopsis

```
const char *aff_symbol_name (const struct AffSymbol_s *sym);
```

Description

Get the name of the symbol. The string is stored internally in the symbol table and should not be freed or modified.

Return Value

Return a pointer to the null-terminated string, or `NULL` if `sym` is `NULL`.

8.4.2 aff_symbol_id()

Synopsis

```
uint32_t aff_symbol_id (const struct AffSymbol_s *sym);
```

Description

Get the index of a symbol.

Return Value

Return the index, or `0xffffffff` if `sym` is zero.

8.5 Treap structure

8.5.1 aff_treap_init()

Synopsis

```
struct AffTreap_s *aff_treap_init (void);
```

Description

Allocate and initialize an empty treap.

Return Value

Return a pointer to a treap, or `NULL` on failure.

8.5.2 aff_treap_fini()

Synopsis

```
void *aff_treap_fini (struct AffTreap_s *h);
```

Description

Free a treap.

Return Value

Return `NULL`. This helps with the following programming pattern:

`treap = aff_treap_fini(treap);`

– clean up the treap and guard against stray accesses by setting it to `NULL`.

8.5.3 aff_treap_cmp()

Synopsis

```
int aff_treap_cmp (const void *a_ptr, unsigned int a_size,
         const void *b_ptr, unsigned int b_size);
```

Description

Compare key `a_ptr` of length `a_size` with key `b_ptr` of length `b_size`. This function defines the ordering used by the treap internaly. It is probably of little use to the user.

Return Value

Return `-1` if key `a_ptr` is less than `b_ptr`, `+1` if key `a_ptr` is greater than `b_ptr`, and zero if they are equal.

8.5.4 aff_treap_lookup()

Synopsis

```
void *aff_treap_lookup (const struct AffTreap_s *h,
         const void *key, int ksize);
```

Description

Lookup the the key `key` of length `ksize` in the treap `h`.

Return Value

Return the pointer to the data associated with the `key`, or `NULL` if there is no such key or `h` is `NULL`.

8.5.5 aff_treap_insert()

Synopsis

```
int aff_treap_insert (struct AffTreap_s *h, const void *key,
         int ksize, void *data);
```

Description

Insert the pair `key` and `data` into the treap `h`. The `ket` must be unique. The `data` is not managed by the treap and should be maintained by the user.

Return Value

Return zero on successful insertion, or non-zero if the key is already present in the treap, insertion failed, or `h` is `NULL`.

8.5.6 aff_treap_print()

Synopsis

```
void aff_treap_print (struct AffTreap_s *h,
         int (*get_vsize)(const void *));
```

Description

Print the treap for debug.

9 MD5 sum functions

This functions implement the MD5 cryptographic checksum as described in RFC 1321. The implementation is taken from the RFC, only the naming conventions were changed to confirm to the rest of the library.

9.1 The Interface

9.1.1 aff_md5_init()

Synopsis

```
void aff_md5_init (struct AffMD5_s *);
```

Description

Initialize MD5 sum state.

9.1.2 aff_md5_update()

Synopsis

```
void aff_md5_update (struct AffMD5_s *, const uint8_t *, uint32_t);
```

Description

Update MD5 state when new data is added to a buffer.

9.1.3 aff_md5_final()

Synopsis

```
void aff_md5_final (uint8_t [16], struct AffMD5_s *);
```

Description

Produce the final value of MD5 sum.

10 AFF Mathematica Interface

By popular demand, there is also a read AFF interface for Wolfram's Mathematica. The interface consists of the following three functions; please note, however, that it is not as bullet-proof as the C interface—it does not do error checking to the same extend.

You will need to load file `$(prefix)/math/aff.m` into Mathematica to use the interface.

10.1 The Interface

10.1.1 `affOpen[]`

Synopsis
`affHandle = affOpen [fileName];`
Description
Open `fileName` as an AFF and prepare for reading data from it.
Return Value
A handle to a Mathematica AFF object is returned.

10.1.2 `affGet[]`

Synopsis
`{type, {data, ...}} = affGet [affHandle, keyPath];`
Description
Read data stored under the `keyPath` in the AFF handle `affHandle` which should have been previosly opened with `affOpen[]`.
Return Value
If `keyPath` is not present, return `$Failure`; otherwise, the type of data and the data itself are returned. The data is returned as a list of values.

10.1.3 `affClose[]`

Synopsis
`affClose [affHandle];`
Description
Close the file associated with `affHandle` and free all allocated resources.

INDEX