Eet Reference Manual

Generated by Doxygen 1.5.1

Wed Mar 28 00:01:04 2007

Contents

Chapter 1

Eet Library Documentation

Version:

@

Author:

Carsten Haitzler <raster@rasterman.com>

Date:

2000-2004

1.1 What is Eet?

It is a tiny library designed to write an arbitary set of chunks of data to a file and optionally compress each chunk (very much like a zip file) and allow fast random-access reading of the file later on. It does not do zip as a zip itself has more complexity than is needed, and it was much simpler to impliment this once here.

Eet is extremely fast, small and simple. Eet files can be very small and highly compressed, making them very optimal for just sending across the internet without having to archive, compress or decompress and install them. They allow for lightning-fast random-acess reads once created, making them perfect for storing data that is written once (or rarely) and read many times, but the program does not want to have to read it all in at once.

It also can encode and decode data structures in memory, as well as image data for saving to Eet files or sending across the network to other machines, or just writing to arbitary files on the system. All data is encoded in a platform independant way and can be written and read by any architecture.

1.2 A simple example on using Eet

Here is a simple example on how to use Eet to save a series of strings to a file and load them again. The advantage of using Eet over just fprintf() and fscanf() is that not only can these entries

be strings, they need no special parsing to handle delimiter characters or escaping, they can be binary data, image data, data structures containing integers, strings, other data structures, linked lists and much more, without the programmer having to worry about parsing, and best of all, Eet is very fast.

```
#include <Eet.h>

int
main(int argc, char **argv)
{
  Eet_File *ef;
  int       i;
  char      buf[32];
  char     *ret;
  int       size;
  char    **entries =
    {
      "Entry 1",
      "Big text string here compared to others",
      "Eet is cool"
    };

  eet_init();

  // blindly open an file for output and write strings with their NUL char
  ef = eet_open("test.eet", EET_FILE_MODE_WRITE);
  eet_write(ef, "Entry 1", entries[0], strlen(entries[0]) + 1, 0);
  eet_write(ef, "Entry 2", entries[1], strlen(entries[1]) + 1, 1);
  eet_write(ef, "Entry 3", entries[2], strlen(entries[2]) + 1, 0);
  eet_close(ef);

  // open the file again and blindly get the entries we wrote
  ef = eet_open("test.eet", EET_FILE_MODE_READ);
  ret = eet_read(ef, "Entry 1", &size);
  printf("%s\n", ret);
  ret = eet_read(ef, "Entry 2", &size);
  printf("%s\n", ret);
  ret = eet_read(ef, "Entry 3", &size);
  printf("%s\n", ret);
  eet_close(ef);

  eet_shutdown();
}
```

1.3 What does an Eet file look like?

The file format is very simple. There is a directory block at the start of the file listing entries and offsets into the file where they are stored, their sizes, compression flags etc. followed by all the entry data strung one element after the other.

All Eet files start with t a 4 byte magic number. It is written using network byte-order (big endian, or from most significant byte first to least significant byte last) and is 0x1ee7ff00 (or byte by byte 0:1e 1:e7 2:ff 3:00). The next 4 bytes are an integer (in big endian notation) indicating how many entries are stored in the Eet file. 0 indicates it is empty. This is a signed integer and thus values less than 0 are invalid, limiting the number of entries in an Eet file to 0x7fffffff entries at most. The next 4 bytes is the size of the directory table, in bytes, encoded in big-endian format. This is a signed integer and cannot be less than 0.

The directory table for the file follows immediately, with a continuous list of all entries in the Eet

file, their offset in the file etc. The order of these entries is not important, but convention would have them be from first to last entry in the file. Each directory entry consiste of 5 integers, one after the other, each stored as a signed, big endian integer. The first is the offset in the file that the data for this entry is stored at (based from the very start of the file, not relative to the end of the directory block). The second integer holds flags for the entry. currently only the least significant bit (bit 0) holds any useful information, and it is set to 1 if the entry is compressed using zlib compression calls, or 0 if it is not compressed. The next integer is the size of the entry in bytes stored in the file. The next integer is the size of the data when decompressed (if it was compressed) in bytes. This may be the same as the previous integer if the entry was not compressed. The final integer is the number of bytes used by the string identifier for the entry, without the NUL byte terminator, which is not stored. The next series of bytes is the string name of the entry, with the number of bytes being the same as specified in the last integer above. This list of entries continues until there are no more entries left to list. To read an entry from an Eet file, simply find the appropriate entry in the directory table, find it's offset and size, and read it into memory. If it is compressed, decompress it using zlib and then use that data.

Here is a data map of an Eet file. All integers are encoded using big-endian notation (most significant byte first) and are signed. There is no alignment of data, so all data types follow immediately on, one after the other. All compressed data is compressed using the zlib compress2() function, and decompressed using the zlib uncompress() function. Please see zlib documentation for more information as to the encoding of compressed data.

```
HEADER:
[INT] Magic number (0x1ee7ff00)
[INT] Number of entries in the directory table
[INT] The size of the directory table, in bytes

DIRECTORY TABLE ENTRIES (as many as specified in the header):
[INT] Offest from file start at which entry is stored (in bytes)
[INT] Entry flags (1 = compressed, 0 = not compressed)
[INT] Size of data chunk in file (in bytes)
[INT] Size of the data chunk once decompressed (or the same as above, if not)
[INT] The length of the string itendifier, in bytes, without NUL terminator
[STR] Series of bytes for the string identifier, no NUL terminator
... more directory entries

DATA STORED, ONE AFTER ANOTHER:
[DAT] DATA ENTRY 1...
[DAT] DATA ENTRY 2...
[DAT] DATA ENTRY 3...
... more data chunks
```

The contents of each entry in an Eet file has no defined format as such. It is an opaque chunk of data, that is up to the application to deocde, unless it is an image, ecoded by Eet, or a data structure encoded by Eet. The data itself for these entries can be encoded and decoded by Eet with extra helper functions in Eet. eet_data_image_read() and eet_data_image_write() are used to handle reading and writing image data from a known Eet file entry name. eet_data_read() and eet_data_write() are used to decode and encode program data structures from an Eet file, making the loading and saving of program information stored in data structures a simple 1 function call process.

Please see src/lib/eet_data.c for information on the format of these specially encoded data entries in an Eet file (for now).

Todo

Add hash table, fixed and variable array encode/decode support.

Todo

Document data format for images and data structures.

Chapter 2

Eet File Index

2.1 Eet File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Eet Page Index

3.1 Eet Related Pages

Here is a list of all related documentation pages:

Chapter 4

Eet File Documentation

4.1 eet.c File Reference

Eet Data Handling Library Public API Calls.

Defines

- #define EET_T_UNKNOW 0
  Unknown data encding type.
- #define EET_T_CHAR 1
  Data type: char.
- #define EET_T_SHORT 2
  Data type: short.
- #define EET_T_INT 3
  Data type: int.
- #define EET_T_LONG_LONG 4
  Data type: long long.
- #define EET_T_FLOAT 5
  Data type: float.
- #define EET_T_DOUBLE 6
  Data type: double.
- #define EET_T_UCHAR 7
  Data type: unsigned char.
- #define EET_T_USHORT 8

Data type: unsigned short.

- #define EET_T_UINT 9

  Data type: unsigned int.

- #define EET_T_ULONG_LONG 10

  Data type: unsigned long long.

- #define EET_T_STRING 11

  *Data type: char *.*

- #define EET_T_LAST 12

  Last data type.

- #define EET_G_UNKNOWN 100

  Unknown group data encoding type.

- #define EET_G_ARRAY 101

  Fixed size array group type.

- #define EET_G_VAR_ARRAY 102

  Variable size array group type.

- #define EET_G_LIST 103

  Linked list group type.

- #define EET_G_HASH 104

  Hash table group type.

- #define EET_G_LAST 105

  Last group type.

- #define EET_DATA_DESCRIPTOR_ADD_BASIC(edd, struct_type, name, member, type)

  Add a basic data element to a data descriptor.

- #define EET_DATA_DESCRIPTOR_ADD_SUB(edd, struct_type, name, member, subtype)

  Add a sub-element type to a data descriptor.

- #define EET_DATA_DESCRIPTOR_ADD_LIST(edd, struct_type, name, member, subtype)

  Add a linked list type to a data descriptor.

- #define EET_DATA_DESCRIPTOR_ADD_HASH(edd, struct_type, name, member, subtype)

  Add a hash type to a data descriptor.

Functions

- EAPI int eet_init (void)

  Initialize the EET library.

- EAPI int eet_shutdown (void)

  Shut down the EET library.

- EAPI void eet_clearcache (void)

  Clear eet cache.

- EAPI Eet_File * eet_open (const char *file, Eet_File_Mode mode)

  Open an eet file on disk, and returns a handle to it.

- EAPI Eet_File_Mode eet_mode_get (Eet_File *ef)

  Get the mode an Eet_File was opened with.

- EAPI Eet_Error eet_close (Eet_File *ef)

  Close an eet file handle and flush and writes pending.

- EAPI void * eet_read (Eet_File *ef, const char *name, int *size_ret)

  Read a specified entry from an eet file and return data.

- EAPI void * eet_read_direct (Eet_File *ef, const char *name, int *size_ret)

  Read a specified entry from an eet file and return data.

- EAPI int eet_write (Eet_File *ef, const char *name, const void *data, int size, int compress)

  Write a specified entry to an eet file handle.

- EAPI int eet_delete (Eet_File *ef, const char *name)

  Delete a specified entry from an Eet file being written or re-written.

- EAPI char ** eet_list (Eet_File *ef, const char *glob, int *count_ret)

  List all entries in eet file matching shell glob.

- EAPI int eet_num_entries (Eet_File *ef)

  Return the number of entries in the specified eet file.

- EAPI int eet_data_image_header_read (Eet_File *ef, const char *name, unsigned int *w, unsigned int *h, int *alpha, int *compress, int *quality, int *lossy)

  Read just the header data for an image and dont decode the pixels.

- EAPI void * eet_data_image_read (Eet_File *ef, const char *name, unsigned int *w, unsigned int *h, int *alpha, int *compress, int *quality, int *lossy)

  Read image data from the named key in the eet file.

- EAPI int eet_data_image_write (Eet_File *ef, const char *name, const void *data, unsigned int w, unsigned int h, int alpha, int compress, int quality, int lossy)

  Write image data to the named key in an eet file.

- EAPI int eet_data_image_header_decode (const void *data, int size, unsigned int *w, unsigned int *h, int *alpha, int *compress, int *quality, int *lossy)

  Decode Image data header only to get information.

- EAPI void * eet_data_image_decode (const void *data, int size, unsigned int *w, unsigned int *h, int *alpha, int *compress, int *quality, int *lossy)

  Decode Image data into pixel data.

- EAPI void * eet_data_image_encode (const void *data, int *size_ret, unsigned int w, unsigned int h, int alpha, int compress, int quality, int lossy)

  Encode image data for storage or transmission.

- EAPI Eet_Data_Descriptor * eet_data_descriptor_new (const char *name, int size, void *(*func_list_next)(void *l), void *(*func_list_append)(void *l, void *d), void *(*func_list_data)(void *l), void *(*func_list_free)(void *l), void(*func_hash_foreach)(void *h, int(*func)(void *h, const char *k, void *dt, void *fdt), void *fdt), void *(*func_hash_add)(void *h, const char *k, void *d), void(*func_hash_free)(void *h))

  Create a new empty data structure descriptor.

- EAPI void eet_data_descriptor_free (Eet_Data_Descriptor *edd)

  This function frees a data descriptor when it is not needed anymore.

- EAPI void eet_data_descriptor_element_add (Eet_Data_Descriptor *edd, const char *name, int type, int group_type, int offset, int count, const char *counter_name, Eet_Data_Descriptor *subtype)

  This function is an internal used by macros.

- EAPI void * eet_data_read (Eet_File *ef, Eet_Data_Descriptor *edd, const char *name)

  Read a data structure from an eet file and decodes it.

- EAPI int eet_data_write (Eet_File *ef, Eet_Data_Descriptor *edd, const char *name, const void *data, int compress)

  Write a data structure from memory and store in an eet file.

- EAPI void * eet_data_descriptor_decode (Eet_Data_Descriptor *edd, const void *data_in, int size_in)

  Decode a data structure from an arbitary location in memory.

- EAPI void * eet_data_descriptor_encode (Eet_Data_Descriptor *edd, const void *data_in, int *size_ret)

  Encode a dsata struct to memory and return that encoded data.

4.1.1 Detailed Description

Eet Data Handling Library Public API Calls.

These routines are used for Eet Library interaction

4.1.2 Define Documentation

4.1.2.1 #define EET_DATA_DESCRIPTOR_ADD_BASIC(edd, struct_type, name, member, type)

Value:

```
{ \
  struct_type ___ett; \
  \
  eet_data_descriptor_element_add(edd, name, type, EET_G_UNKNOWN, \
         (char *)(&(___ett.member)) - (char *)(&(___ett)), \
         0, NULL, NULL); \
    }
```

Add a basic data element to a data descriptor.

Parameters:

edd The data descriptor to add the type to.

struct_type The type of the struct.

name The string name to use to encode/decode this member (must be a constant global and never change).

member The struct member itself to be encoded.

type The type of the member to encode.

This macro is a convenience macro provided to add a member to the data descriptor `edd`. The type of the structure is provided as the `struct_type` parameter (for example: struct my_struct). The `name` parameter defines a string that will be used to uniquely name that member of the struct (it is suggested to use the struct member itself). The `member` parameter is the actual struct member itself (for example: values), and `type` is the basic data type of the member which must be one of: EET_T_CHAR, EET_T_SHORT, EET_T_INT, EET_T_LONG_LONG, EET_T_FLOAT, EET_T_DOUBLE, EET_T_UCHAR, EET_T_USHORT, EET_T_UINT, EET_T_ULONG_LONG or EET_T_STRING.

4.1.2.2 #define EET_DATA_DESCRIPTOR_ADD_HASH(edd, struct_type, name, member, subtype)

Value:

```
{ \
  struct_type ___ett; \
  \
  eet_data_descriptor_element_add(edd, name, EET_T_UNKNOW, EET_G_HASH, \
         (char *)(&(___ett.member)) - (char *)(&(___ett)), \
         0, NULL, subtype); \
    }
```

Add a hash type to a data descriptor.

Parameters:

edd The data descriptor to add the type to.

struct_ type The type of the struct.

name The string name to use to encode/decode this member (must be a constant global and never change).

member The struct member itself to be encoded.

subtype The type of hash member to add.

This macro lets you easily add a hash of other data types. All the parameters are the same as for EET_DATA_DESCRIPTOR_ADD_BASIC(), with the `subtype` being the exception. This must be the data descriptor of the element that is in each member of the hash to be stored.

4.1.2.3 #define EET_DATA_DESCRIPTOR_ADD_LIST(edd, struct_type, name, member, subtype)

Value:

```
{ \
  struct_type ___ett; \
  \
  eet_data_descriptor_element_add(edd, name, EET_T_UNKNOW, EET_G_LIST, \
          (char *)(&(___ett.member)) - (char *)(&(___ett)), \
          0, NULL, subtype); \
     }
```

Add a linked list type to a data descriptor.

Parameters:

edd The data descriptor to add the type to.

struct_ type The type of the struct.

name The string name to use to encode/decode this member (must be a constant global and never change).

member The struct member itself to be encoded.

subtype The type of linked list member to add.

This macro lets you easily add a linked list of other data types. All the parameters are the same as for EET_DATA_DESCRIPTOR_ADD_BASIC(), with the `subtype` being the exception. This must be the data descriptor of the element that is in each member of the linked list to be stored.

4.1.2.4 #define EET_DATA_DESCRIPTOR_ADD_SUB(edd, struct_type, name, member, subtype)

Value:

```
{ \
  struct_type ___ett; \
  \
  eet_data_descriptor_element_add(edd, name, EET_T_UNKNOW, EET_G_UNKNOWN, \
          (char *)(&(___ett.member)) - (char *)(&(___ett)), \
          0, NULL, subtype); \
     }
```

Add a sub-element type to a data descriptor.

Parameters:

edd The data descriptor to add the type to.

struct_ type The type of the struct.

name The string name to use to encode/decode this member (must be a constant global and never change).

member The struct member itself to be encoded.

subtype The type of sub-type struct to add.

This macro lets you easily add a sub-type (a struct that's pointed to by this one). All the parameters are the same as for EET_DATA_DESCRIPTOR_ADD_BASIC(), with the `subtype` being the exception. This must be the data descriptor of the struct that is pointed to by this element.

4.1.3 Function Documentation

4.1.3.1 EAPI void eet_clearcache (void)

Clear eet cache.

Eet didn't free items by default. If you are under memory presure, just call this function to recall all memory that are not yet referenced anymore. The cache take care of modification on disk.

4.1.3.2 EAPI Eet_Error eet_close (Eet_File * *ef*)

Close an eet file handle and flush and writes pending.

Parameters:

ef A valid eet file handle.

This function will flush any pending writes to disk if the eet file was opened for write, and free all data associated with the file handle and file, and close the file.

If the eet file handle is not valid nothing will be done.

4.1.3.3 EAPI void* eet_data_descriptor_decode (Eet_Data_Descriptor * *edd*, const void * *data_in*, int *size_in*)

Decode a data structure from an arbitary location in memory.

Parameters:

edd The data descriptor to use when decoding.

data_in The pointer to the data to decode into a struct.

size_in The size of the data pointed to in bytes.

Returns:

NULL on failure, or a valid decoded struct pointer on success.

This function will decode a data structure that has been encoded using eet_data_descriptor_encode(), and return a data structure with all its elements filled out, if successful, or NULL on failure.

The data to be decoded is stored at the memory pointed to by `data_in`, and is described by the descriptor pointed to by `edd`. The data size is passed in as the value to `size_in`, ande must be greater than 0 to succeed.

This function is useful for decoding data structures delivered to the application by means other than an eet file, such as an IPC or socket connection, raw files, shared memory etc.

Please see eet_data_read() for more information.

4.1.3.4 EAPI void eet_data_descriptor_element_add (Eet_Data_Descriptor * *edd*, const char * *name*, int *type*, int *group_type*, int *offset*, int *count*, const char * *counter_name*, Eet_Data_Descriptor * *subtype*)

This function is an internal used by macros.

This function is used by macros EET_DATA_DESCRIPTOR_ADD_BASIC(), EET_DATA_DESCRIPTOR_ADD_SUB() and EET_DATA_DESCRIPTOR_ADD_LIST(). It is complex to use by hand and should be left to be used by the macros, and thus is not documented.

4.1.3.5 EAPI void* eet_data_descriptor_encode (Eet_Data_Descriptor * *edd*, const void * *data_in*, int * *size_ret*)

Encode a dsata struct to memory and return that encoded data.

Parameters:

edd The data descriptor to use when encoding.

data_in The pointer to the struct to encode into data.

size_ret A pointer to the an int to be filled with the decoded size.

Returns:

NULL on failure, or a valid encoded data chunk on success.

This function takes a data structutre in memory and encodes it into a serialised chunk of data that can be decoded again by eet_data_descriptor_decode(). This is useful for being able to transmit data structures across sockets, pipes, IPC or shared file mechanisms, without having to worry about memory space, machine type, endianess etc.

The parameter `edd` must point to a valid data descriptor, and `data_in` must point to the right data structure to encode. If not, the encoding may fail.

On success a non NULL valid pointer is returned and what `size_ret` points to is set to the size of this decoded data, in bytes. When the encoded data is no longer needed, call free() on it. On failure NULL is returned and what `size_ret` points to is set to 0.

Please see eet_data_write() for more information.

4.1.3.6 EAPI void eet_data_descriptor_free (Eet_Data_Descriptor * *edd*)

This function frees a data descriptor when it is not needed anymore.

Parameters:

edd The data descriptor to free.

This function takes a data descriptor handle as a parameter and frees all data allocated for the data descriptor and the handle itself. After this call the descriptor is no longer valid.

4.1.3.7 EAPI Eet_Data_Descriptor* eet_data_descriptor_new (const char * *name*, int *size*, void *(*)(void *l) *func_list_next*, void *(*)(void *l, void *d) *func_list_append*, void *(*)(void *l) *func_list_data*, void *(*)(void *l) *func_list_free*, void(*)(void *h, int(*func)(void *h, const char *k, void *dt, void *fdt), void *fdt) *func_hash_foreach*, void *(*)(void *h, const char *k, void *d) *func_hash_add*, void(*)(void *h) *func_hash_free*)

Create a new empty data structure descriptor.

Parameters:

name The string name of this data structure (most be a global constant and never change).

size The size of the struct (in bytes).

func_list_next The function to get the next list node.

func_list_append The function to append a member to a list.

func_list_data The function to get the data from a list node.

func_list_free The function to free an entire linked list.

func_hash_foreach The function to iterate through all hash table entries.

func_hash_add The function to add a member to a hash table.

func_hash_free The function to free an entire hash table.

Returns:

A new empty data descriptor.

This function creates a new data descriptore and returns a handle to the new data descriptor. On creation it will be empty, containing no contents describing anything other than the shell of the data structure.

You add structure members to the data descriptor using the macros EET_DATA_DESCRIPTOR_ADD_BASIC(), EET_DATA_DESCRIPTOR_ADD_SUB() and EET_DATA_DESCRIPTOR_ADD_LIST(), depending on what type of member you are adding to the description.

Once you have described all the members of a struct you want loaded, or saved eet can load and save those members for you, encode them into endian-independant serialised data chunks for transmission across a a network or more.

Example:

```
#include <Eet.h>
#include <Evas.h>

typedef struct _blah2
{
   char *string;
}
Blah2;
```

```
typedef struct _blah3
{
   char *string;
}
Blah3;

typedef struct _blah
{
   char character;
   short sixteen;
   int integer;
   long long lots;
   float floating;
   double floating_lots;
   char *string;
   Blah2 *blah2;
   Evas_List *blah3;
}
Blah;

int
main(int argc, char **argv)
{
   Blah blah;
   Blah2 blah2;
   Blah3 blah3;
   Eet_Data_Descriptor *edd, *edd2, *edd3;
   void *data;
   int size;
   FILE *f;
   Blah *blah_in;

   edd3 = eet_data_descriptor_new("blah3", sizeof(Blah3),
                                  evas_list_next,
                                  evas_list_append,
                                  evas_list_data,
                                  evas_list_free,
                                  evas_hash_foreach,
                                  evas_hash_add,
                                  evas_hash_free);
   EET_DATA_DESCRIPTOR_ADD_BASIC(edd3, Blah3, "string3", string, EET_T_STRING);

   edd2 = eet_data_descriptor_new("blah2", sizeof(Blah2),
                                  evas_list_next,
                                  evas_list_append,
                                  evas_list_data,
                                  evas_list_free,
                                  evas_hash_foreach,
                                  evas_hash_add,
                                  evas_hash_free);
   EET_DATA_DESCRIPTOR_ADD_BASIC(edd2, Blah2, "string2", string, EET_T_STRING);

   edd = eet_data_descriptor_new("blah", sizeof(Blah),
                                  evas_list_next,
                                  evas_list_append,
                                  evas_list_data,
                                  evas_list_free,
                                  evas_hash_foreach,
                                  evas_hash_add,
                                  evas_hash_free);
   EET_DATA_DESCRIPTOR_ADD_BASIC(edd, Blah, "character", character, EET_T_CHAR);
   EET_DATA_DESCRIPTOR_ADD_BASIC(edd, Blah, "sixteen", sixteen, EET_T_SHORT);
   EET_DATA_DESCRIPTOR_ADD_BASIC(edd, Blah, "integer", integer, EET_T_INT);
   EET_DATA_DESCRIPTOR_ADD_BASIC(edd, Blah, "lots", lots, EET_T_LONG_LONG);
   EET_DATA_DESCRIPTOR_ADD_BASIC(edd, Blah, "floating", floating, EET_T_FLOAT);
   EET_DATA_DESCRIPTOR_ADD_BASIC(edd, Blah, "floating_lots", floating_lots, EET_T_DOUBLE);
```

```
   EET_DATA_DESCRIPTOR_ADD_BASIC(edd, Blah, "string", string, EET_T_STRING);
   EET_DATA_DESCRIPTOR_ADD_SUB(edd, Blah, "blah2", blah2, edd2);
   EET_DATA_DESCRIPTOR_ADD_LIST(edd, Blah, "blah3", blah3, edd3);

   blah3.string="PANTS";

   blah2.string="subtype string here!";

   blah.character='7';
   blah.sixteen=0x7777;
   blah.integer=0xc0def00d;
   blah.lots=0xdeadbeef31337777;
   blah.floating=3.141592654;
   blah.floating_lots=0.777777777777777;
   blah.string="bite me like a turnip";
   blah.blah2 = &blah2;
   blah.blah3 = evas_list_append(NULL, &blah3);
   blah.blah3 = evas_list_append(blah.blah3, &blah3);
   blah.blah3 = evas_list_append(blah.blah3, &blah3);
   blah.blah3 = evas_list_append(blah.blah3, &blah3);
   blah.blah3 = evas_list_append(blah.blah3, &blah3);
   blah.blah3 = evas_list_append(blah.blah3, &blah3);
   blah.blah3 = evas_list_append(blah.blah3, &blah3);

   data = eet_data_descriptor_encode(edd, &blah, &size);
   f = fopen("out", "w");
   if (f)
     {
        fwrite(data, size, 1, f);
        fclose(f);
     }
   printf("-----DECODING\n");
   blah_in = eet_data_descriptor_decode(edd, data, size);

   printf("-----DECODED!\n");
   printf("%c\n", blah_in->character);
   printf("%x\n", (int)blah_in->sixteen);
   printf("%x\n", blah_in->integer);
   printf("%lx\n", blah_in->lots);
   printf("%f\n", (double)blah_in->floating);
   printf("%f\n", (double)blah_in->floating_lots);
   printf("%s\n", blah_in->string);
   printf("%p\n", blah_in->blah2);
   printf("  %s\n", blah_in->blah2->string);
     {
        Evas_List *l;

        for (l = blah_in->blah3; l; l = l->next)
          {
             Blah3 *blah3_in;

             blah3_in = l->data;
             printf("%p\n", blah3_in);
             printf("  %s\n", blah3_in->string);
          }
     }
   eet_data_descriptor_free(edd);
   eet_data_descriptor_free(edd2);
   eet_data_descriptor_free(edd3);

  return 0;
}
```

4.1.3.8 EAPI void* eet_data_image_decode (const void * *data*, int *size*, unsigned int * *w*, unsigned int * *h*, int * *alpha*, int * *compress*, int * *quality*, int * *lossy*)

Decode Image data into pixel data.

Parameters:

data The encoded pixel data.

size The size, in bytes, of the encoded pixel data.

w A pointer to the unsigned int to hold the width in pixels.

h A pointer to the unsigned int to hold the height in pixels.

alpha A pointer to the int to hold the alpha flag.

compress A pointer to the int to hold the compression amount.

quality A pointer to the int to hold the quality amount.

lossy A pointer to the int to hold the lossiness flag.

Returns:

The image pixel data decoded

This function takes encoded pixel data and decodes it into raw RGBA pixels on success.

The other parameters of the image (width, height etc.) are placed into the values pointed to (they must be supplied). The pixel data is a linear array of pixels starting from the top-left of the image scanning row by row from left to right. Each pixel is a 32bit value, with the high byte being the alpha channel, the next being red, then green, and the low byte being blue. The width and height are measured in pixels and will be greater than 0 when returned. The alpha flag is either 0 or 1. 0 denotes that the alpha channel is not used. 1 denotes that it is significant. Compress is filled with the compression value/amount the image was stored with. The quality value is filled with the quality encoding of the image file (0 - 100). The lossy flags is either 0 or 1 as to if the image was encoded lossily or not.

On success the function returns a pointer to the image data decoded. The calling application is responsible for calling free() on the image data when it is done with it. On failure NULL is returned and the parameter values may not contain any sensible data.

4.1.3.9 EAPI void* eet_data_image_encode (const void * *data*, int * *size_ret*, unsigned int *w*, unsigned int *h*, int *alpha*, int *compress*, int *quality*, int *lossy*)

Encode image data for storage or transmission.

Parameters:

data A pointer to the image pixel data.

size_ret A pointer to an int to hold the size of the returned data.

w The width of the image in pixels.

h The height of the image in pixels.

alpha The alpha channel flag.

compress The compression amount.

quality The quality encoding amount.

lossy The lossiness flag.

Returns:

The encoded image data.

This function stakes image pixel data and encodes it with compression and possible loss of quality (as a trade off for size) for storage or transmission to another system.

The data expected is the same format as returned by eet_data_image_read. If this is not the case weird things may happen. Width and height must be between 1 and 8000 pixels. The alpha flags can be 0 or 1 (0 meaning the alpha values are not useful and 1 meaning they are). Compress can be from 0 to 9 (0 meaning no compression, 9 meaning full compression). This is only used if the image is not lossily encoded. Quality is used on lossy compression and should be a value from 0 to 100. The lossy flag can be 0 or 1. 0 means encode losslessly and 1 means to encode with image quality loss (but then have a much smaller encoding).

On success this function returns a pointer to the encoded data that you can free with free() when no longer needed.

4.1.3.10 EAPI int eet_data_image_header_decode (const void * *data*, int *size*, unsigned int * *w*, unsigned int * *h*, int * *alpha*, int * *compress*, int * *quality*, int * *lossy*)

Decode Image data header only to get information.

Parameters:

data The encoded pixel data.

size The size, in bytes, of the encoded pixel data.

w A pointer to the unsigned int to hold the width in pixels.

h A pointer to the unsigned int to hold the height in pixels.

alpha A pointer to the int to hold the alpha flag.

compress A pointer to the int to hold the compression amount.

quality A pointer to the int to hold the quality amount.

lossy A pointer to the int to hold the lossiness flag.

Returns:

1 on success, 0 on failure.

This function takes encoded pixel data and decodes it into raw RGBA pixels on success.

The other parameters of the image (width, height etc.) are placed into the values pointed to (they must be supplied). The pixel data is a linear array of pixels starting from the top-left of the image scanning row by row from left to right. Each pixel is a 32bit value, with the high byte being the alpha channel, the next being red, then green, and the low byte being blue. The width and height are measured in pixels and will be greater than 0 when returned. The alpha flag is either 0 or 1. 0 denotes that the alpha channel is not used. 1 denotes that it is significant. Compress is filled with the compression value/amount the image was stored with. The quality value is filled with the quality encoding of the image file (0 - 100). The lossy flags is either 0 or 1 as to if the image was encoded lossily or not.

On success the function returns 1 indicating the header was read and decoded properly, or 0 on failure.

4.1.3.11 EAPI int eet_data_image_header_read (Eet_File * *ef*, const char * *name*, unsigned int * *w*, unsigned int * *h*, int * *alpha*, int * *compress*, int * *quality*, int * *lossy*)

Read just the header data for an image and dont decode the pixels.

Parameters:

ef A valid eet file handle opened for reading.

name Name of the entry. eg: "/base/file_i_want".

w A pointer to the unsigned int to hold the width in pixels.

h A pointer to the unsigned int to hold the height in pixels.

alpha A pointer to the int to hold the alpha flag.

compress A pointer to the int to hold the compression amount.

quality A pointer to the int to hold the quality amount.

lossy A pointer to the int to hold the lossiness flag.

Returns:

1 on successfull decode, 0 otherwise

This function reads an image from an eet file stored under the named key in the eet file and return a pointer to the decompressed pixel data.

The other parameters of the image (width, height etc.) are placed into the values pointed to (they must be supplied). The pixel data is a linear array of pixels starting from the top-left of the image scanning row by row from left to right. Each pile is a 32bit value, with the high byte being the alpha channel, the next being red, then green, and the low byte being blue. The width and height are measured in pixels and will be greater than 0 when returned. The alpha flag is either 0 or 1. 0 denotes that the alpha channel is not used. 1 denotes that it is significant. Compress is filled with the compression value/amount the image was stored with. The quality value is filled with the quality encoding of the image file (0 - 100). The lossy flags is either 0 or 1 as to if the image was encoded lossily or not.

On success the function returns 1 indicating the header was read and decoded properly, or 0 on failure.

4.1.3.12 EAPI void* eet_data_image_read (Eet_File * *ef*, const char * *name*, unsigned int * *w*, unsigned int * *h*, int * *alpha*, int * *compress*, int * *quality*, int * *lossy*)

Read image data from the named key in the eet file.

Parameters:

ef A valid eet file handle opened for reading.

name Name of the entry. eg: "/base/file_i_want".

w A pointer to the unsigned int to hold the width in pixels.

h A pointer to the unsigned int to hold the height in pixels.

alpha A pointer to the int to hold the alpha flag.

compress A pointer to the int to hold the compression amount.

quality A pointer to the int to hold the quality amount.

lossy A pointer to the int to hold the lossiness flag.

Returns:

The image pixel data decoded

This function reads an image from an eet file stored under the named key in the eet file and return a pointer to the decompressed pixel data.

The other parameters of the image (width, height etc.) are placed into the values pointed to (they must be supplied). The pixel data is a linear array of pixels starting from the top-left of the image scanning row by row from left to right. Each pile is a 32bit value, with the high byte being the alpha channel, the next being red, then green, and the low byte being blue. The width and height are measured in pixels and will be greater than 0 when returned. The alpha flag is either 0 or 1. 0 denotes that the alpha channel is not used. 1 denotes that it is significant. Compress is filled with the compression value/amount the image was stored with. The quality value is filled with the quality encoding of the image file (0 - 100). The lossy flags is either 0 or 1 as to if the image was encoded lossily or not.

On success the function returns a pointer to the image data decoded. The calling application is responsible for calling free() on the image data when it is done with it. On failure NULL is returned and the parameter values may not contain any sensible data.

4.1.3.13 EAPI int eet_data_image_write (Eet_File * *ef*, const char * *name*, const void * *data*, unsigned int *w*, unsigned int *h*, int *alpha*, int *compress*, int *quality*, int *lossy*)

Write image data to the named key in an eet file.

Parameters:

ef A valid eet file handle opened for writing.

name Name of the entry. eg: "/base/file_i_want".

data A pointer to the image pixel data.

w The width of the image in pixels.

h The height of the image in pixels.

alpha The alpha channel flag.

compress The compression amount.

quality The quality encoding amount.

lossy The lossiness flag.

Returns:

Success if the data was encoded and written or not.

This function takes image pixel data and encodes it in an eet file stored under the supplied name key, and returns how many bytes were actually written to encode the image data.

The data expected is the same format as returned by eet_data_image_read. If this is not the case weird things may happen. Width and height must be between 1 and 8000 pixels. The alpha flags can be 0 or 1 (0 meaning the alpha values are not useful and 1 meaning they are). Compress can be from 0 to 9 (0 meaning no compression, 9 meaning full compression). This is only used if

the image is not lossily encoded. Quality is used on lossy compression and should be a value from 0 to 100. The lossy flag can be 0 or 1. 0 means encode losslessly and 1 means to encode with image quality loss (but then have a much smaller encoding).

On success this function returns the number of bytes that were required to encode the image data, or on failure it returns 0.

4.1.3.14 EAPI void* eet_data_read (Eet_File * *ef*, Eet_Data_Descriptor * *edd*, const char * *name*)

Read a data structure from an eet file and decodes it.

Parameters:

ef The eet file handle to read from.

edd The data descriptor handle to use when decoding.

name The key the data is stored under in the eet file.

Returns:

A pointer to the decoded data structure.

This function decodes a data structure stored in an eet file, returning a pointer to it if it decoded successfully, or NULL on failure. This can save a programmer dozens of hours of work in writing configuration file parsing and writing code, as eet does all that work for the program and presents a program-friendly data structure, just as the programmer likes. Eet can handle members being added or deleted from the data in storage and safely zero-fills unfilled members if they were not found in the data. It checks sizes and headers whenever it reads data, allowing the programmer to not worry about corrupt data.

Once a data structure has been described by the programmer with the fields they wish to save or load, storing or retrieving a data structure from an eet file, or from a chunk of memory is as simple as a single function call.

4.1.3.15 EAPI int eet_data_write (Eet_File * *ef*, Eet_Data_Descriptor * *edd*, const char * *name*, const void * *data*, int *compress*)

Write a data structure from memory and store in an eet file.

Parameters:

ef The eet file handle to write to.

edd The data descriptor to use when encoding.

name The key to store the data under in the eet file.

data A pointer to the data structure to ssave and encode.

compress Compression flags for storage.

Returns:

1 on successful write, 0 on failure.

This function is the reverse of eet_data_read(), saving a data structure to an eet file.

4.1.3.16 EAPI int eet_delete (Eet_File * *ef*, const char * *name*)

Delete a specified entry from an Eet file being written or re-written.

Parameters:

ef A valid eet file handle opened for writing.

name Name of the entry. eg: "/base/file_i_want".

Returns:

Success or failure of the delete.

This function will delete the specified chunk of data from the eet file and return greater than 0 on success. 0 will be returned on failure.

The eet file handle must be a valid file handle for an eet file opened for writing. If it is not, 0 will be returned and no action will be performed.

Name, must not be NULL, otherwise 0 will be returned.

4.1.3.17 EAPI int eet_init (void)

Initialize the EET library.

Returns:

The new init count.

4.1.3.18 EAPI char** eet_list (Eet_File * *ef*, const char * *glob*, int * *count_ret*)

List all entries in eet file matching shell glob.

Parameters:

ef A valid eet file handle.

glob A shell glob to match against.

count_ret Number of entries found to match.

Returns:

Pointer to an array of strings.

This function will list all entries in the eet file matching the supplied shell glob and return an allocated list of their names, if there are any, and if no memory errors occur.

The eet file handle must be valid and glob must not be NULL, or NULL will be returned and count_ret will be filled with 0.

The calling program must call free() on the array returned, but NOT on the string pointers in the array. They are taken as read-only internals from the eet file handle. They are only valid as long as the file handle is not closed. When it is closed those pointers in the array are now not valid and should not be used.

On success the array returned will have a list of string pointers that are the names of the entries that matched, and count_ret will have the number of entries in this array placed in it.

Hint: an easy way to list all entries in an eet file is to use a glob value of "*".

4.1.3.19 EAPI Eet_File_Mode eet_mode_get (Eet_File * *ef*)

Get the mode an Eet_File was opened with.

Parameters:

ef A valid eet file handle.

Returns:

The mode ef was opened with.

4.1.3.20 EAPI int eet_num_entries (Eet_File * *ef*)

Return the number of entries in the specified eet file.

Parameters:

ef A valid eet file handle.

Returns:

Number of entries in ef or -1 if the number of entries cannot be read due to open mode restrictions.

4.1.3.21 EAPI Eet_File* eet_open (const char * *file*, Eet_File_Mode *mode*)

Open an eet file on disk, and returns a handle to it.

Parameters:

file The file path to the eet file. eg: "/tmp/file.eet".

mode The mode for opening. Either EET_FILE_MODE_READ or EET_FILE_MODE_WRITE, but not both.

Returns:

An opened eet file handle.

This function will open an exiting eet file for reading, and build the directory table in memory and return a handle to the file, if it exists and can be read, and no memory errors occur on the way, otherwise NULL will be returned.

It will also open an eet file for writing. This will, if successful, delete the original file and replace it with a new empty file, till the eet file handle is closed or flushed. If it cannot be opened for writing or a memory error occurs, NULL is returned.

Example:

```
#include <Eet.h>
#include <stdio.h>

int
main(int argc, char **argv)
{
  Eet_File *ef;
  char buf[1024], *ret, **list;
  int size, num, i;

  strcpy(buf, "Here is a string of data to save!");

  ef = eet_open("/tmp/my_file.eet", EET_FILE_MODE_WRITE);
  if (!ef) return -1;
  if (!eet_write(ef, "/key/to_store/at", buf, 1024, 1))
    fprintf(stderr, "Error writing data!\n");
  eet_close(ef);

  ef = eet_open("/tmp/my_file.eet", EET_FILE_MODE_READ);
  if (!ef) return -1;
  list = eet_list(ef, "*", &num);
  if (list)
    {
      for (i = 0; i < num; i++)
        printf("Key stored: %s\n", list[i]);
      free(list);
    }
  ret = eet_read(ef, "/key/to_store/at", &size);
  if (ret)
    {
      printf("Data read (%i bytes):\n%s\n", size, ret);
      free(ret);
    }
  eet_close(ef);

  return 0;
}
```

4.1.3.22 EAPI void* eet_read (Eet_File * *ef*, const char * *name*, int * *size_ret*)

Read a specified entry from an eet file and return data.

Parameters:

ef A valid eet file handle opened for reading.

name Name of the entry. eg: "/base/file_i_want".

size_ret Number of bytes read from entry and returned.

Returns:

The data stored in that entry in the eet file.

This function finds an entry in the eet file that is stored under the name specified, and returns that data, decompressed, if successful. NULL is returned if the lookup fails or if memory errors are encountered. It is the job of the calling program to call free() on the returned data. The number of bytes in the returned data chunk are placed in size_ret.

If the eet file handle is not valid NULL is returned and size_ret is filled with 0.

4.1.3.23 EAPI void* eet_read_direct (Eet_File * *ef*, const char * *name*, int * *size_ret*)

Read a specified entry from an eet file and return data.

Parameters:

ef A valid eet file handle opened for reading.

name Name of the entry. eg: "/base/file_i_want".

size_ret Number of bytes read from entry and returned.

Returns:

The data stored in that entry in the eet file.

This function finds an entry in the eet file that is stored under the name specified, and returns that data if not compressed and successful. NULL is returned if the lookup fails or if memory errors are encountered or if the data is comrpessed. The calling program must never call free() on the returned data. The number of bytes in the returned data chunk are placed in size_ret.

If the eet file handle is not valid NULL is returned and size_ret is filled with 0.

4.1.3.24 EAPI int eet_shutdown (void)

Shut down the EET library.

Returns:

The new init count.

4.1.3.25 EAPI int eet_write (Eet_File * *ef*, const char * *name*, const void * *data*, int *size*, int *compress*)

Write a specified entry to an eet file handle.

Parameters:

ef A valid eet file handle opened for writing.

name Name of the entry. eg: "/base/file_i_want".

data Pointer to the data to be stored.

size Length in bytes in the data to be stored.

compress Compression flags (1 == compress, 0 = don't compress).

Returns:

Success or failure of the write.

This function will write the specified chunk of data to the eet file and return greater than 0 on success. 0 will be returned on failure.

The eet file handle must be a valid file handle for an eet file opened for writing. If it is not, 0 will be returned and no action will be performed.

Name, and data must not be NULL, and size must be > 0. If these conditions are not met, 0 will be returned.

The data will be copied (and optionally compressed) in ram, pending a flush to disk (it will stay in ram till the eet file handle is closed though).

Chapter 5

Eet Page Documentation

5.1 Todo List

page Eet Library Documentation Add hash table, fixed and variable array encode/decode support.

page Eet Library Documentation Document data format for images and data structures.

Index