Notes on "EP" Library

Author Eric Allman

Date 2017-06-28

1 INTRODUCTION

This document describes the Enhanced Portability library. This is a reduced version of Eric's Portability library (designed for sendmail), removing many things that didn't work out or proved unnecessary, e.g., the entire I/O subsystem. It was originally intended to be used in sendmail, so some of the terminology is geared toward the email world; none the less, it should be generally useful.

Programs using `libep` should be able to minimize `#ifdef` since often non-portable functionality is wrapped in portability routines; for example, BSD-derived systems and Linux-derived systems differ in how you get the name of the currently running program. These differences are hidden if the application calls `ep_app_getprogname`.

1.1 Design Goals

- Portable, to the extent possible. Where not possible, there needs to be a clearly codified way to represent the externally visible semantic differences.
- Efficient.
- Customizable -- try to implement mechanism, not policy.
- TBD: It should be entirely UTF-8 internally. Any translations to other character sets should be done on input or output, and then only as strictly necessary.

1.2 Assumptions

Use of this library requires that you have a C compiler that is compliant with ANSI C as defined by ANSI/ISO 9899-1999. Also requires an environment that is at least Posix based on Posix.1-2008.

1.3 Conventions

- All externally visible names (i.e., those not declared "static" in a file) shall be named ep_* (for routine names) or Ep* (for variable names). In a few cases, the names may begin with __ep or __Ep; such names would be in the global namespace, but would be intended for use internal to the library only. There are a few cases where "standard" names (such as `strlcpy`) are defined if they are not included in the standard library.

1.4 Terminology

1.4.1 Warning, Error, Severe, Abort

These words get used fairly loosely, so they are worth defining. In the context of libep:

- Warning means a condition that is expected in normal operations, but is not the usual case. Reading an end-of-file on a file might be a warning. Applications need to be aware of these, but are expected to either ignore them or recover easily. This can also be used for temporary errors which are likely to recover

after a delay. For example, the inability to open a connection to a remote server might recover automatically if that server is re-started. However, this sort of warnings that persist should turn into permanent errors in a fashion appropriate for the application.

- Error means a situation that should not occur, but isn't terribly unusual. For example, an attempt to open a file that isn't accessible would be an error. Applications must be aware of such conditions and handle them gracefully.
- Severe means a situation that should not occur iand requires exceptional handling. Severe errors are drastic conditions, but are not so severe that the application can't take some reasonable backout action.
- Abort means a situation so drastic that an application cannot be expected to make any reasonable recovery. These might include assertion errors and memory allocation failures during a critical step (e.g., something where backing out a single thread won't solve the problem). About the only thing an application can reasonably do is log an abort error and exit. In particular, an abort is appropriate when an attempt for an application to recover is likely to do additional damage. These should be extremely rare.

2 GENERAL ISSUES

All files using this library must use "`#include <ep/ep.h>`".

3 STATUS CODES

Almost all functions return an `EP_STAT` value. This is a short (integer-encoded) status value that gives you a brief idea of how severe the problem was and some idea of what it was, but not much else. Think of it as an errno equivalent. Functions returning any status other than OK are expected to provide some other way of returning detailed data.

`EP_STAT`s are also used as message identifiers for logging (below).

Status codes are defined in `<ep/ep_stat.h>`.

3.1 Severities

Severities are:

`EP_STAT_SEV_OK`
: Everything is fine. Detail may contain info. For messages, can be used for debugging.

`EP_STAT_SEV_WARN`
: The function partially succeeded, but there is something that the application should be aware of, e.g., an end of file or a short data read. Alternatively, the functionfailed, but it might work again on a later try.

`EP_STAT_SEV_ERROR`
: A normal error status. The call failed.

`EP_STAT_SEV_SEVERE`
: A severe error status. The call failed, and the caller should try to back out.

`EP_STAT_SEV_ABORT`
: A critical error occured — you should clean up and exit as soon as possible; the program cannot be expected to operate correctly.

Some functions for testing values:

`EP_STAT_SEV_ISOK(st)`
: Returns true if this is an `EP_STAT_SEV_OK` status code.

EP_STAT_SEV_WARN(st)
Returns true if this is an "warning" severity status code: EP_STAT_SEV_WARN.

EP_STAT_SEV_ISERROR(st)
Returns true if this is an "error" severity status code: EP_STAT_SEV_ERROR

EP_STAT_SEV_ISFAIL(st)
Returns true if this message is a "failure" severity status code: EP_STAT_SEV_ERROR or higher.

EP_STAT_SEV_ISSEVERE(st)
Returns true if this is an "severe" severity status code: EP_STAT_SEV_SEVERE

EP_STAT_SEV_ISSFAIL(st)
Returns true if this message is a "major" severity status code: EP_STAT_SEV_SEVERE or higher

EP_STAT_SEV_ISABORT(st)
Returns true if this is an "abort" severity status code: EP_STAT_SEV_ABORT

3.2 Status Code Representation

Status codes are represented as four-part values: severity, registry, module, and detail. The severities are described above. Registries are globally registered by neophilic.com and are defined in ep_registry.h. There are some registries for general use; in particular, registry numbers between 0x001 and 0x1FF are available for local (non-global) registry at the corporate or local level. Modules are defined by registries, and detail is defined by module. It is *never* acceptable to look at detail unless you recognize the module. (OK, you can print it out for debugging.) Severity = 3 bits, registry = 13 bits, module = 6 bits, detail = 10 bits.

Any severity where the top bit is zero is considered "OK", and the rest of the word is available to encode a non-negative integer.

Status codes are represented as structures to ensure type safety. Occassionally you might want to convert a status to or from a long int:

```
int      EP_STAT_TO_INT(EP_STAT stat)          // convert status to unsigned int
EP_STAT  EP_STAT_FROM_INT(unsigned int istat)  // convert unsigned integer to status
```

The constituent parts of the status code can also be extracted:

EP_STAT_SEV(st)
Returns the severity part of the status code.

EP_STAT_REGISTRY(st)
Returns the registry part of the status code.

EP_STAT_MODULE(st)
Returns the module part of the status code.

EP_STAT_DETAIL(st)
Returns the detail part of the status code.

To compare two statuses for equality, use EP_STAT_IS_SAME(a, b).

As a special case, if the severity is EP_STAT_SEV_OK the rest of the word is ignored; this can be used to pass small integers (no more than 31 bits) of information.

3.3 Predefined Status Codes

All status codes from this library are in the EP_REGISTRY_EPLIB registery. There are several predefined status

codes for generic use, all using module EP_STAT_MOD_GENERIC:

EP_STAT_OK	No error (also integer 0)
EP_STAT_WARN	Generic warning status
EP_STAT_ERROR	Generic error status
EP_STAT_SEVERE	Generic severe error status
EP_STAT_ABORT	Generic abortive status
EP_STAT_OUT_OF_MEMORY	Out of memory
EP_STAT_ARG_OUT_OF_RANGE	An argument was out of range
EP_STAT_END_OF_FILE	End of input
EP_STAT_TIME_BADFORMAT	Couldn't parse a date/time string
EP_STAT_BUF_OVERFLOW	Buffer overflow averted
EP_STAT_ASSERT_ABORT	Assertion failiure: backout now!

There is also a special module EP_STAT_MOD_ERRNO that encodes Posix-style errnos (i.e., use EP_STAT_DETAIL on codes returned by that module to get the Posix errno code).

3.4 Manipulating Status Codes

There are several routines to print error codes or create them on the fly. Note that ep_stat_tostr returns the buffer itself.

```
// create status code from UNIX errno
extern EP_STAT  ep_stat_from_errno(
                        int uerrno);

// return string representation of status
char            *ep_stat_tostr(
                        EP_STAT estat,
                        char *buf,
                        size_t bsize);

// return string representation of severity (in natural language)
const char      *ep_stat_sev_tostr(
                        int sev);

// print a status code and abort (never returns)
void            ep_stat_abort(
                        EP_STAT estat);
```

3.5 Creating New Status Codes

Libraries and applications can create their own specific error codes. There are four steps to do this:

1. Determine the registry. The registry name space is divided as follows:

0x000 (EP_REGISTRY_GENERIC)	reserved for generic status codes
0x001 (EP_REGISTRY_USER)	available for internal use to an application
0x002-0x07F	available for local, unregistered use, such as separate applications within an application suite

0x080-0x0FF	available for internal corporate registry, but not registered globally; conflicts may occur between organizations but not within an organization
0x100 (EP_REGISTRY_EPLIB)	reserved for libep
0x101-0x6FF	available for centrally managed global registry — contact the libep maintainers for an allocation
0x700-0x7FF	reserved

2. Determine the module(s) in which the error code should exist. These must be unique within a registry and in the range 0x00-0xFF.
3. Define the error codes you want to use using `EP_STAT_NEW`, e.g.,

```
#define FOO_STAT_ELEPHANT    EP_STAT_NEW(ERROR, EP_REGISTRY_FOO, MOD_BAR, 1)
#define FOO_STAT_GIRAFFE     EP_STAT_NEW(SEVERE, EP_REGISTRY_FOO, MOD_BAR, 2)
```

4. (Optional step) Define the strings associated with the error codes when they are printed. These are done by populating a table and then calling `ep_stat_register_strings`. For example:

```
struct ep_stat_reg_strings  FooStatusCodes[] =
{
    FOO_STAT_ELEPHANT,    "elephant in the room",        },
    FOO_STAT_GIRAFFE,     "too tall",                    },
    EP_STAT_OK,           NULL,                          }
}

ep_stat_reg_strings(FooStatusCodes);
```

4 INITIALIZATION

Although libep will generally work without initialization, in some cases you may need to give it information about your usage. To do this call `ep_lib_init`:

```
#include <ep/ep.h>

EP_STAT
ep_lib_init(uint32_t flags)
```

Flags can be:

EP_LIB_USEPTHREADS	Initialize the thread support

5 MEMORY ALLOCATION AND RESOURCE POOLS

5.1 Memory

Memory support is much like malloc/free, but with some additional functionality. One crucial difference is that most of these routines do not return if memory is exhausted; instead they can call a cleanup routine that might (for example) eliminate some old cache entries, or pick a "victim" thread to kill and reclaim its memory. If successful they can continue, otherwise the process is aborted.

```
    #include <ep/ep_mem.h>

    void *
    ep_mem_malloc(size_t nbytes)       // allocate uninitialized memory
```

```
void *
ep_mem_zalloc(size_t nbytes)       // allocate zeroed memory

void *
ep_mem_ralloc(size_t nbytes)       // allocate randomized memory

void *
ep_mem_ealloc(size_t nbytes)       // allocate memory, failure OK

void *
ep_mem_realloc(size_t nbytes,      // reallocate (extend) memory
            void *curmem)

void *
ep_mem_falloc(size_t nbytes,       // allocate memory (see flags)
            uint32_t flags)

void
ep_mem_mfree(void *mem)            // free indicated memory

struct ep_malloc_functions
{
    void    *(*m_malloc)(size_t);
    void    *(*m_realloc)(void*, size_t);
    void    *(*m_valloc)(size_t);
    void    (*m_free)(void*);
};

void
ep_mem_set_malloc_functions(       // set underlying malloc functions
            struct ep_malloc_functions *funcs)
```

The `ep_mem_malloc`, `ep_mem_zalloc`, `ep_mem_ralloc`, and `ep_mem_realloc` are all implemented in terms of `ep_mem_falloc`, which uses flags to tune the behavior (see below). The primary interface is `ep_mem_malloc`, which returns uninitialized data; `ep_mem_zalloc` returns zeroed memory, and `ep_mem_ralloc` returns memory that is initialized to random or some other nonsensical data. The last would probably be used only for debugging, and can be turned on at runtime using a debug flag *XXX TBD*.

In all allocation schemes, the function returns a pointer to the allocated data — they cannot normally return `NULL` (but see below). If they cannot allocate the memory, they *do error recovery (XXX describe)*. If recovery fails, the allocation system will abort the process. However, `ep_mem_ealloc` can return `NULL` on memory allocation failure, as can `ep_mem_falloc` if the `EP_MEM_F_FAILOK` flag bit is set (see below).

Flag bits are as follows:

`EP_MEM_F_FAILOK`
: Permits the routine to return `NULL` on failure. This modifies the behavior described above. Note that if this is set every call to the `ep_*malloc` routines may potentially fail.

`EP_MEM_F_ZERO`
: Zero any returned memory.

`EP_MEM_F_TRASH`
: Randomize any returned memory.

`EP_MEM_F_ALIGN`
: The application would prefer that the allocation is page-aligned. This is not available on all architectures, and other architectures do it automatically if the allocation is at least as large as a page.

`EP_MEM_F_WAIT`

If memory is unavailable, try to wait for it to become available (e.g., because another thread has released memory). *This is not yet implemented.*

Specifying `EP_MEM_F_ZERO` and `EP_MEM_F_TRASH` at the same time is undefined.

Since `ep_mem_[mzr]alloc` are implemented as macros, they can't be used as pointers to functions (e.g., for specifying a memory allocator callback to a third party app). For this reason, there are also `ep_mem_[mzr]alloc_f` "real" functions to be used in this context.

Generally, unthreaded code and most application code will probably be happy with the defaults. Threaded server code (which cannot be permitted to die) is expected to catch the out of memory condition, do some recovery operation such as terminating a task, and return `EP_MEM_STAT_TRYAGAIN` so the memory allocation can retry.

[[[XXX Document ep_set_malloc_functions XXX]]]

5.2 Resource Pools

Resources are allocatable global entities such as memory, file descriptors, etc. Resources can be collected together into pools and then freed in one call. Memory is specially handled to allow fast allocation from a pool --- specifically, a chunk of memory can be allocated from the heap to a pool and then sub-allocated as needed. Allocating memory from resource pools is particularly fast for small allocations. Also, pool allocations that are of a size that is a multiple of the page size are guaranteed to return a page-aligned pointer. This is particularly useful to allow the I/O level to implement zero-copy I/O.

The heap used is the one that is current when `ep_rpool_new` is invoked.

```
#include <ep/ep_mem.h>

EP_RPOOL *
ep_rpool_new(const char *name,              // for debugging
            size_t qsize)                   // min memory allocation quantum

EP_STAT
ep_rpool_free(EP_RPOOL *rp)                 // free pool and all resources

void *
ep_rpool_malloc(EP_RPOOL *rp,               // the pool to allocate from
            size_t nbytes)                  // number of bytes

void *
ep_rpool_zalloc(EP_RPOOL *rp,               // the pool to allocate from
            size_t nbytes)                  // number of bytes

void *
ep_rpool_xalloc(EP_RPOOL *rp,               // the pool to allocate from
            size_t nbytes,                  // number of bytes
            const char *filename,           // file name (for debugging)
            int lineno,                     // line number (for debugging)
            uint32_t flags)                 // flag bits (see below)

void *
ep_rpool_strdup(EP_RPOOL *rp,               // the pool to allocate from
            char *str)                      // the string to save

void *
ep_rpool_realloc(EP_RPOOL *rp,              // pool to allocate from
            void *old_mem,                  // old memory pointer
            size_t oldsize,                 // old allocation size
```

```
                size_t newsize)                 // new allocation size

    void
    ep_rpool_mfree(EP_RPOOL *rp,                // the pool to release to
                void *p);                       // the memory

    void
    ep_rpool_mfreeto(EP_RPOOL *rp,              // the pool to release to
                void *p);                       // restore up the pool to here

    EP_STAT
    ep_rpool_attach(EP_RPOOL *rp,               // the resource pool
                void freefunc(void *arg),       // a function to call on free
                void *arg)                      // argument to pass to it
```

The ep_rpool_mfreeto() routine lets you treat rpool memory like a stack; this call releases everything allocated back to (and including) the pointer given. If `p == NULL`, the entire memory contents of the rpool are freed, but the rpool itself is still active. Deep care needs to be taken here: if a subordinate routine is called that allocates memory from the rpool, you may end up deallocating memory that is still in use. *Not implemented at this time.*

The ep_rpool_attach() routine is used to associate other resources (such as files) with a pool. The corresponding free functions will be invoked when the pool is freed.

In most cases, passing in `rp == NULL` treats the call like the corresponding heap allocation. In this case the caller is responsible for freeing the memory. For example, ep_rpool_malloc(NULL, nbytes) is equivalent to ep_mem_malloc(nbytes).

The distinction between multiple heaps and resource pools are that heaps are not intended for application use other than for doing recovery for out-of-memory conditions. Pools are intended for general use. Pools are fast at allocation time (since they just grab space from the end of the pool) and fast at free time (since the entire pool can be deallocated at once); heaps are comparatively slow.

When any memory collections (heaps or pools) are freed, all objects allocated from that collection are freed (i.e., their destructors are automatically invoked).

5.3 Opening Memory as a File

```
    FILE *
    ep_fopen_smem(void *buf,            // block of memory to open
        size_t bsize,                   // size of that memory
        const char *mode)               // fopen(3) mode string
```

6 TIME

The ep library has a separate time abstraction. This is for two reasons: first, it guarantees that the number of seconds since January 1, 1970 will be sufficiently long to last past 2038 (this varies from system to system), and it includes a "tv_accuracy" (type float) to indicate the approximate accuracy of the clock relative to absolute time. For example, a clock synchronized from a GPS clock might be accurate within perhaps 100nsec, whereas a standard crystal clock synchronized once a day might only have an accuracy of a few seconds.

```
#include <ep/ep_time.h>

typedef struct
{
        int64_t     tv_sec;         // seconds since Jan 1, 1970
        int32_t     tv_nsec;        // nanoseconds
        float       tv_accuracy;    // clock accuracy in seconds
} EP_TIME_SPEC;
```

```
EP_STAT
ep_time_now(                        // return current time
        EP_TIME_SPEC *tv);

EP_STAT
ep_time_deltanow(                   // return time in the future (or past)
        uint64_t delta_nanoseconds,
        EP_TIME_SPEC *tv);

void
ep_time_add_delta(                  // add a delta to a time (delta may be negative)
        EP_TIME_SPEC *delta,
        EP_TIME_SPEC *tv);

bool
ep_time_before(                     // determine if A occurred before B
        EP_TIME_SPEC *a,
        EP_TIME_SPEC *b);

void
ep_time_from_nsec(                  // create a time from a scalar number of nanoseconds
        int64_t delta,
        EP_TIME_SPEC *tv);

float
ep_time_accuracy(void);             // return putative clock accuracy

void
ep_time_setaccuracy(                // set the clock accuracy (may not be available)
        float accuracy);

void
ep_time_format(                     // format a time string into a buffer
        EP_TIME_SPEC *tv,
        char *buf,
        size_t bufsize,
        uint32_t flags);

void
ep_time_print(                      // format a time string to a file
        EP_TIME_SPEC *tv,
        FILE *fp,
        uint32_t flags);

// values for ep_time_format and ep_time_print flags
#define EP_TIME_FMT_DEFAULT     0               // pseudo-flag
#define EP_TIME_FMT_HUMAN       0x00000001      // format for humans
#define EP_TIME_FMT_NOFUZZ      0x00000002      // suppress accuracy printing

EP_STAT
ep_time_parse(                      // parse a time string
        const char *timestr,
        EP_TIME_SPEC *tv,
        uint32_t flags);

// values for ep_time_parse flags
#define EP_TIME_USE_UTC         0x00000000      // assume UTC (default)
#define EP_TIME_USE_LOCALTIME   0x00000001      // assume times in local zone

EP_STAT
```

```
ep_time_nanosleep(                      // sleep for the indicated number of nanoseconds
        int64_t nanoseconds);

bool
EP_TIME_IS_VALID(                       // test to see if a timestamp is valid
        EP_TIME_SPEC *tv);

void
EP_TIME_INVALIDATE(                     // invalidate a timestamp
        EP_TIME_SPEC *tv);
```

"Human" formatted times are intended to be human readable, and may use non-ASCII characters. Otherwise the format is intended to be machine readable, e.g., using `ep_time_parse`.

7 DATA STRUCTURES

7.1 Property Lists

Not implemented at this time. A series of key=value pairs. Used for many things, including configuration files. For example, looking in the "configuration" property list for "`mailer.local.timeout.connect`" would return the connect timeout for the local mailer. *[[How does this deal with nested defaults — e.g., looking for timeout.connect if the full path cannot be found?]]*

```
    EP_PLIST *
    ep_plist_new(
                const char *name)               // for printing

    EP_STAT
    ep_plist_load(
                EP_PLIST *plp,                  // the list to read into
                FILE *sp,                       // the stream to load from
                const char *prefix)             // prefix added to all properties

    EP_STAT
    ep_plist_set(
                EP_PLIST *plp,                  // the plist in which to set
                const char *keyname,            // the name of the key to set
                const char *value)              // the value to set (will be copied)

    const char *
    ep_plist_get(
                EP_PLIST *plp,                  // the plist to search
                const char *keyname)            // the name of the key to get

    void
    ep_plist_dump(
                EP_PLIST *plp,                  // plist to print
                FILE *sp)                       // stream to print to

    void
    ep_plist_free(
                EP_PLIST *plp)                  // plist to free
```

A property list can be loaded from an external stream using `ep_plist_load`. The syntax of the file is a simple text file with "key=value" pairs on separate lines, with blank lines and those with # at the beginning of the line ignored. The values are strictly strings. *[[Does it make sense to type them?]]*

[[Note the overlap between plists and the `ep_adm` interface. Does this make sense?]]

Property lists can be printed using `ep_plist_dump`. The output format will be readable by `ep_plist_load`. For the time being, `flags` should always be `0`.

> **Warning** The property list is not guaranteed to be dumped in the same order items are inserted.

7.2 Hashes

```
#include <ep/ep_hash.h>

EP_HASH *
ep_hash_new(
            const char *name,               // for printing
            EP_HASH_HASH_FUNCP *hfunc,      // alternate hash function
            int tabsize)                    // hash table function size

void
ep_hash_free(
            EP_HASH *hp)                    // hash to free

void *
ep_hash_search(
            const EP_HASH *hp,              // hash to search
            size_t keylen,                  // length of key
            const void *key)                // pointer to key

void *
ep_hash_insert(                     // returns old value for key
            EP_HASH *hp,                    // hash to modify
            size_t keylen,                  // length of key
            const void *key,                // pointer to key
            void *val)                      // value to insert

ep_hash_forall(EP_HASH *hp,                 // hash to walk
            void (func)(                    // function to call
                    int keylen,                 // key length
                    const void *key,            // key value
                    void *val,                  // value
                    void *closure),             // from caller
            void *closure)                  // passed to func

ep_hash_dump(EP_TREE *tree,                 // tree to dump
            FILE *sp)                       // stream to print on
```

[[Should `ep_hash_dump` take the same parameters as the usual object print routine? For that matter, should there be a separate `ep_hash_dump` routine, or should it just be a generic `ep_obj_dump`? Note that `ep_hash_dump` is not implemented at this time, but an internal (object-based) dump is.]]

7.3 Function Lists

```
#include <ep/ep_funclist.h>

EP_FUNCLIST *
ep_funclist_new(
            const char *name)       // name for printing/debugging

void
ep_funclist_free(EP_FUNCLIST *fp)   // list to free

void
ep_funclist_push(EP_FUNCLIST *fp,   // list to push to
```

```
                void (*func)(void *),   // the function to invoke
                void *arg)              // an argument to pass to it

    void
    ep_funclist_pop(EP_FUNCLIST *fp)    // list to pop from, value discarded

    void
    ep_funclist_clear(EP_FUNCLIST *fp)  // list to clear

    void
    ep_funclist_invoke(EP_FUNCLIST *fp) // invoke all functions in list
```

8 CRYPTOGRAPHIC SUPPORT

The current implementation wraps the OpenSSL library, but it could be retargeted.

Before any cryptographic functions can be used, the library must be initialized:

```
void         ep_crypto_init(uint32_t flags)
```

At the moment flags is unused (just pass zero). There are also several general purpose definitions, useful for declaring buffers without memory allocation:

EP_CRYPTO_MAX_PUB_KEY	Maximum length of a public key
EP_CRYPTO_MAX_SEC_KEY	Maximum length of a secret key
EP_CRYPTO_MAX_DIGEST	Maximum length of a message digest
EP_CRYPTO_MAX_DER	Maximum length of a DER-encoded key

8.1 Key Management

Internally all keys are represented as `EP_CRYPTO_KEY` variables, defined in `ep_crypto.h`. External representations for keys may be either PEM (Privacy Enhanced Mail, represented as text) or DER (Distinguished Encoding Rules, represented in binary). PEM self identifies the type of key, but DER does not, so in some cases the key type needs to be pre-arranged.

```
// on-disk key formats
# define EP_CRYPTO_KEYFORM_UNKNOWN      0       // error
# define EP_CRYPTO_KEYFORM_PEM          1       // PEM (ASCII-encoded text)
# define EP_CRYPTO_KEYFORM_DER          2       // DER (binary ASN.1)
```

Internally, algorithms (e.g., for keys and hash/digest functions) are represented by a scalar value. Keys also have to be identified as public or secret.

[[Note: DH is not supported at this time.]]

```
// key types
# define EP_CRYPTO_KEYTYPE_UNKNOWN      0       // error
# define EP_CRYPTO_KEYTYPE_RSA          1       // RSA
# define EP_CRYPTO_KEYTYPE_DSA          2       // DSA
# define EP_CRYPTO_KEYTYPE_EC           3       // Elliptic curve
# define EP_CRYPTO_KEYTYPE_DH           4       // Diffie-Hellman

// flag bits
# define EP_CRYPTO_F_PUBLIC             0x0000  // public key (no flags set)
# define EP_CRYPTO_F_SECRET             0x0001  // secret key
```

Keys are represented as an `EP_CRYPTO_KEY`. New keys can be created by giving the type of the key desired,

the length of the key in bits, and two other values that are interpreted by the key type. The first is primarily for RSA and gives the exponent, and the second is primarily for EC and gives the curve name.

```
EP_CRYPTO_KEY           *ep_crypto_key_create(
                                int keytype,
                                int keylen,
                                int keyexp,
                                const char *curve);
```

Keys can be read from or written to named files, open files, or memory. All the read routines create and return a new key data structure. If the `keyform` is `EP_CRYPTO_KEYFORM_PEM` then the `keytype` need not be specified.

```
EP_CRYPTO_KEY           *ep_crypto_key_read_file(
                                const char *filename,
                                int keyform,
                                uint32_t flags);
EP_CRYPTO_KEY           *ep_crypto_key_read_fp(
                                FILE *fp,
                                const char *filename,
                                int keyform,
                                uint32_t flags);
EP_CRYPTO_KEY           *ep_crypto_key_read_mem(
                                const void *buf,
                                size_t buflen,
                                int keyform,
                                uint32_t flags);
EP_STAT                 ep_crypto_key_write_file(
                                EP_CRYPTO_KEY *key,
                                const char *filename,
                                int keyform,
                                int cipher,
                                uint32_t flags);
EP_STAT                 ep_crypto_key_write_fp(
                                EP_CRYPTO_KEY *key,
                                FILE *fp,
                                int keyform,
                                int cipher,
                                uint32_t flags);
EP_STAT                 ep_crypto_key_write_mem(
                                EP_CRYPTO_KEY *key,
                                void *buf,
                                size_t bufsize,
                                int keyform,
                                int cipher,
                                uint32_t flags);
```

When finished with a key it must be freed.

```
void                    ep_crypto_key_free(
                                EP_CRYPTO_KEY *key);
```

There are also some utility routines. A public and a secret key can be compared to see if they match each other (same algorithm, keysize, etc.) using `ep_crypto_key_compat`. Various conversions are also included: `ep_crypto_keyform_byname` converts a text string (e.g., "pem") to an internal code, `ep_crypto_keytype_fromkey` returns the type of a key, and `ep_crypto_keytype_byname` converts a text string to a type.

```
EP_STAT                 ep_crypto_key_compat(
                                const EP_CRYPTO_KEY *pubkey,
                                const EP_CRYPTO_KEY *seckey);
```

```
int                     ep_crypto_keyform_byname(
                                const char *fmt);
int                     ep_crypto_keytype_fromkey(
                                EP_CRYPTO_KEY *key);
int                     ep_crypto_keytype_byname(
                                const char *alg_name);
```

8.2 Message Digests (Hashes)

Several message digest (cryptographic hash) algorithms are supported. Text can be converted to one of these values, and the algorithm type can be extracted from the internal form.

```
// digest algorithms (no more than 4 bits)
# define EP_CRYPTO_MD_NULL      0
# define EP_CRYPTO_MD_SHA1      1
# define EP_CRYPTO_MD_SHA224    2
# define EP_CRYPTO_MD_SHA256    3
# define EP_CRYPTO_MD_SHA384    4
# define EP_CRYPTO_MD_SHA512    5

int                     ep_crypto_md_alg_byname(
                                const char *algname);
int                     ep_crypto_md_type(
                                EP_CRYPTO_MD *md);
```

Digests (type `EP_CRYPTO_MD`) can be created, freed, and cloned. Cloning lets an application compute the a fixed part of a digest (perhaps an unchanging header) and then produce separate digests for individual records.

```
EP_CRYPTO_MD            *ep_crypto_md_new(
                                int md_alg_id);
EP_CRYPTO_MD            *ep_crypto_md_clone(
                                EP_CRYPTO_MD *base_md);
void                    ep_crypto_md_free(
                                EP_CRYPTO_MD *md);
```

The typical lifetime of a digest is to be created (as above), updated with additional data, possibly multiple times, and then finalized to give the output hash.

```
EP_STAT                 ep_crypto_md_update(
                                EP_CRYPTO_MD *md,
                                void *data,
                                size_t dsize);
EP_STAT                 ep_crypto_md_final(
                                EP_CRYPTO_MD *md,
                                void *dbuf,
                                size_t *dbufsize);
```

8.3 Signing and Verification

Signing and verification are quite similar. A new internal structure is created, using the same type as a message digest, data is added to the existing hash, possibly multiple times, the signature is created or verified, and finally the structure is freed.

```
# define EP_CRYPTO_MAX_SIG      (1024 * 8)

EP_CRYPTO_MD            *ep_crypto_sign_new(
                                EP_CRYPTO_KEY *skey,
                                int md_alg_id);
void                    ep_crypto_sign_free(
```

```
                                EP_CRYPTO_MD *md);
EP_STAT                 ep_crypto_sign_update(
                                EP_CRYPTO_MD *md,
                                void *dbuf,
                                size_t dbufsize);
EP_STAT                 ep_crypto_sign_final(
                                EP_CRYPTO_MD *md,
                                void *sbuf,
                                size_t *sbufsize);

EP_CRYPTO_MD            *ep_crypto_vrfy_new(
                                EP_CRYPTO_KEY *pkey,
                                int md_alg_id);
void                    ep_crypto_vrfy_free(
                                EP_CRYPTO_MD *md);
EP_STAT                 ep_crypto_vrfy_update(
                                EP_CRYPTO_MD *md,
                                void *dbuf,
                                size_t dbufsize);
EP_STAT                 ep_crypto_vrfy_final(
                                EP_CRYPTO_MD *md,
                                void *obuf,
                                size_t obufsize);
```

8.4 Encryption and Decryption (Asymmetric)

To be supplied.

8.5 Encryption and Decryption (Symmetric Ciphers)

Symmetric Ciphers are driven by a Chaining Mode (how subsequent blocks have the key modified to prevent replay and brute force attacks) and the actual cipher itself. The chaining modes are:

EP_CRYPTO_MODE_CBC	Cipher Block Chaining
EP_CRYPTO_MODE_CFB	Cipher Feedback mode
EP_CRYPTO_MODE_OFB	Output Feedback mode

The various cipher algorithms (which is equivalent to the key type) are:

EP_CRYPTO_SYMKEY_NONE	Error/unencrypted
EP_CRYPTO_SYMKEY_AES128	Advanced Encr Std, 128-bit key
EP_CRYPTO_SYMKEY_AES192	Advanced Encr Std, 192-bit key
EP_CRYPTO_SYMKEY_AES256	Advanced Encr Std, 256-bit key
EP_CRYPTO_SYMKEY_CAMELLIA128	Camellia, 128-bit key
EP_CRYPTO_SYMKEY_CAMELLIA192	Camellia, 192-bit key
EP_CRYPTO_SYMKEY_CAMELLIA256	Camellia, 256-bit key
EP_CRYPTO_SYMKEY_DES	Data Encryption Standard, single, 56-bit key
EP_CRYPTO_SYMKEY_3DES	Data Encryption Standard, triple, 128-bit key (112-bit effective)
EP_CRYPTO_SYMKEY_IDEA	International Data Encryption Alg, 128-bit key

One value from each table are "or"ed together to specify a full symmetric cipher. The rest of the interface is as follows:

```
/*
**      The cipher is set to encrypt or decrypt when the context
**      is created.
**
**      ep_crypto_cipher_crypt is just shorthand for a single
**      call to ep_crypto_cipher_update followed by a single
**      call to ep_crypto_cipher_final.  Final pads out any
**      remaining block and returns that data.
*/

EP_CRYPTO_CIPHER_CTX    *ep_crypto_cipher_new(
                                uint32_t ciphertype,    // mode + keytype & len
                                uint8_t *key,           // the key
                                uint8_t *iv,            // initialization vector
                                bool enc);              // true => encrypt
void                    ep_crypto_cipher_free(
                                EP_CRYPTO_CIPHER_CTX *cipher);

EP_STAT                 ep_crypto_cipher_crypt(
                                EP_CRYPTO_CIPHER_CTX *cipher,
                                void *in,               // input data
                                size_t inlen,           // input length
                                void *out,              // output buffer
                                size_t outlen);         // output buf size
EP_STAT                 ep_crypto_cipher_update(
                                EP_CRYPTO_CIPHER_CTX *cipher,
                                void *in,               // input data
                                size_t inlen,           // input length
                                void *out,              // output buffer
                                size_t outlen);         // output buf size
EP_STAT                 ep_crypto_cipher_final(
                                EP_CRYPTO_CIPHER_CTX *cipher,
                                void *out,              // output buffer
                                size_t outlen);         // output buf size
```

8.6 Cryptography-specific Error Codes

There are several status codes that may be returned from the cryptography routines. These are all in module EP_STAT_MOD_CRYPTO.

EP_STAT_CRYPTO_DIGEST	Failed to update or finalize a digest (hash)
EP_STAT_CRYPTO_SIGN	Failed to update or finalize a digest for signing
EP_STAT_CRYPTO_VRFY	Failed to update or finalize a digest for verification
EP_STAT_CRYPTO_BADSIG	Signature did not match
EP_STAT_CRYPTO_KEYTYPE	Unknown key type
EP_STAT_CRYPTO_KEYFORM	Unknown key format
EP_STAT_CRYPTO_CONVERT	Couldn't read or write a key
EP_STAT_CRYPTO_KEYCREATE	Couldn't create a new key
EP_STAT_CRYPTO_KEYCOMPAT	Public and secret keys are incompatible
EP_STAT_CRYPTO_CIPHER	Symmetric cipher failure

9 APPLICATION SUPPORT

The following routines are intended to provide useful support to applications, but are not otherwise fundamental

9.1 Printing Flag Words, Etc.

```
#include <ep/ep_prflags.h>

void
ep_prflags(
            u_int32 flagword,               // the flags word to print
            EP_PRFLAGS_DESC *flaglist,      // descriptor of flags
            FILE *out)                      // output stream

typedef struct ep_prflags_desc
{
        u_int32     bits;           // bits to compare against
        u_int32     mask;           // mask against flagword
        char        *name;          // printable name
} EP_PRFLAGS_DESC;
```

For example, given a descriptor of:

```
    0x0000, 0x0003, "READ",
    0x0001, 0x0003, "WRITE",
    0x0002, 0x0003, "READWRITE",
    0x0003, 0x0003, "[INVALID MODE]",
    0x0004, 0x0004, "NONBLOCK",
    0x0008, 0x0008, "APPEND",
    0,      0,      NULL
```

then a flagword of 0x0009 would print:

```
0009<WRITE,APPEND>
```

9.2 Printing Helpers

A few routines to make it easier to create string versions of other type variables, e.g., for `ep_stat_post`.

```
#include <ep/ep_pcvt.h>

char *ep_pcvt_str(size_t osize,             // output buffer size
            char *obuf,                     // output buffer
            const char *val)                // value to convert

char *ep_pcvt_int(size_t osize,             // output buffer size
            char *obuf,                     // output buffer
            int base,                       // base of value
            int val)                        // value to convert
```

All of these return their input buffer.

The routine `ep_pcvt_str` truncates the value to the indicated size. If the value won't fit, it renders "*beginning*...*end*" where *end* is the last three bytes of the value.

9.3 Application Messages

Associated with status printing.

```
#include <ep/ep_app.h>

void ep_app_info(const char *fmt,           // printf-style format
```

```
            ...)

void ep_app_warn(const char *fmt,           // printf-style format
            ...)

void ep_app_error(const char *fmt,          // printf-style format
            ...)

void ep_app_fatal(const char *fmt,          // printf-style format
            ...)

void ep_app_abort(const char *fmt,          // printf-style format
            ...)

void ep_app_setflags(uint32_t flags)        // set operational tweaks
```

The first three just print messages; the second two print the message and does not return. `ep_app_abort` generates a core dump on termination. All five use printf formats. `ep_app_setflags` sets flags telling when to also do logging; the flags are `EP_APP_FLAG_LOGABORTS`, `EP_APP_FLAG_LOGFATALS`, `EP_APP_FLAG_LOGERRORS`, `EP_APP_FLAG_LOGWARNINGS`, and `EP_APP_FLAG_LOGINFOS`. The log severity is different for these various functions.

```
const char *
ep_app_getprogname(void)                    // get current program name
```

This is a portability wrapper that returns the name of the current program (essentially, the last component of argv[0]).

9.4 Printing Memory

To print out a block of binary memory, use ep_hexdump.

```
#include <ep/ep_hexdump.h>

void
ep_hexdump(void *bufp,                      // block of memory to print
    size_t buflen,                          // size of that block
    FILE *fp,                               // output file
    int format,                             // see description
    size_t offset);                         // offset
```

This prints a block of memory as a hexadecimal dump, optionally with an ASCII rendition. The offset printed starts at the `offset` parameter (zero to make the printed offsets be relative to `bufp`). The `format` may be `EP_HEXDUMP_HEX` to print only the hexadecimal or `EP_HEXDUMP_ASCII` to also show the bytes interpreted as ASCII (unprintable characters are substituted).

10 DEBUGGING, TRACING, ASSERTIONS

Named flags, each settable from 0 to 127.

When setting flags, wildcards can be used (only "*" supported for now).

```
#include <ep/ep_dbg.h>

void
ep_dbg_init(void)                   // initialize debugging

void
ep_dbg_set(const char *fspec)       // set debug flags (command line)
```

```
void
ep_dbg_setto(const char *fpat,      // flag pattern
             int lev)               // level

EP_DBG  flag EP_DBG_INIT(           // opaque structure for flag
                     name,          // external name of flag
                     desc);         // description (internal use only)

int
ep_dbg_level(EP_DBG *flag)          // return level of given flag

bool
ep_dbg_test(EP_DBG *flag,
              int value)            // true if flag set to >= value

void
ep_dbg_printf(fmt, ...)             // print to EpStStddbg

void
ep_dbg_cprintf(EP_DBG *flag,        // if flag level >= value,
             int value,
             fmt, ...)              // print fmt etc as though printf.

void
ep_dbg_setfile(FILE *fp)            // set debug output to indicated file

void
ep_dbg_getfile(void)                // return current debug output file
```

Assertions are intended to catch "cannot happen" cases. They are not necessarily fatal, depending on configuration controlled by administrative parameters: `libep.assert.maxfailures` specifies the number of assertion failures that will be tolerated before the process aborts; however, every `libep.assert.resetinterval` seconds the failure count is reset. For example, if `libep.assert.maxfailures` is one, all assertion failures are fatal. It defaults to zero (no assertions are fatal); `libep.assert.resetinterval` defines to 60 (one minute).

```
#include <ep/ep_assert.h>

EP_ASSERT(condition)                // fail if condition is false; returns the condition

EP_ASSERT_ELSE(condition, recovery) // print and run recovery code if condition not satis

EP_ASSERT_PRINT(                    // print assertion failure message
        const char *msg,               // message to print
        ...)                           // arguments to message
ep_assert_print(                    // print assertion failure with extra info and messag
        const char *file,              // file name
        int line,                      // line number
        const char *msg,               // message to print
        ...)                           // arguments to message

EP_ASSERT_FAILURE(                  // abort process with message
        const char *msg,               // message to print
        ...)                           // arguments to message
ep_assert_failure(                  // abort process with extra info and message
        const char *file,              // file name
        int line,                      // line number
        const char *msg,               // message to print
        ...)                           // arguments to message
```

```
    void    (*EpAssertInfo)(void)       // if set, call to print additional information
    void    (*EpAssertAbort)(void)      // function to call before aborting
    bool    EpAssertAllAbort            // if set, all assertions are immediately fatal
```

Programs may try to recover from assertion failures by testing the result of `EP_ASSERT`, which will be true if the condition holds. For example, either of these return an error code if a pointer is `NULL`:

```
    if (!EP_ASSERT(p != NULL))
        return EP_STAT_ASSERT_ABORT;

    EP_ASSERT_ELSE(p != NULL, return EP_STAT_ASSERT_ABORT);
```

Processes can force an abort as though they got an assertion failure by calling `ep_assert_failure`. Note that this does not attempt any recovery; `ep_assert_print` does the same thing but does not abort. The macros `EP_ASSERT_FAILURE` and `EP_ASSERT_PRINT` do the same thing, but provide the `file` name and `line` number in the same way as the assertion tests. If the `EpAssertInfo` variable is set, that function will be called after printing the message but before aborting. It can be used to dump process state for debugging. If the `EpAssertAbort` variable is set, that function will be called after the message is printed and immediately before the process aborts. This might do last minute recovery or alternative termination (e.g., terminate just the thread rather than the entire process).

11 THREAD SUPPORT

These are mostly wrappers around the pthreads library, but they will print errors if the `ep.thr` debug flag is set to at least 4.

```
#include <ep/ep_thr.h>

int
ep_thr_mutex_init(EP_THR_MUTEX *mtx, int type);

int
ep_thr_mutex_destroy(EP_THR_MUTEX *mtx);

int
ep_thr_mutex_lock(EP_THR_MUTEX *mtx);

int
ep_thr_mutex_trylock(EP_THR_MUTEX *mtx);

int
ep_thr_mutex_unlock(EP_THR_MUTEX *mtx);

int
ep_thr_mutex_check(EP_THR_MUTEX *mtx);

int
ep_thr_cond_init(EP_THR_COND *cv);

int
ep_thr_cond_destroy(EP_THR_COND *cv);

int
ep_thr_cond_signal(EP_THR_COND *cv);

int
ep_thr_cond_wait(EP_THR_COND *cv, EP_THR_MUTEX *mtx, EP_TIME_SPEC *timeout);
```

```
int
ep_thr_cond_broadcast(EP_THR_COND *cv);

int
ep_thr_rwlock_init(EP_THR_RWLOCK *rwl);

int
ep_thr_rwlock_destroy(EP_THR_RWLOCK *rwl);

int
ep_thr_rwlock_rdlock(EP_THR_RWLOCK *rwl);

int
ep_thr_rwlock_tryrdlock(EP_THR_RWLOCK *rwl);

int
ep_thr_rwlock_wrlock(EP_THR_RWLOCK *rwl);

int
ep_thr_rwlock_tryrwlock(EP_THR_RWLOCK *rwl);

int
ep_thr_rwlock_unlock(EP_THR_RWLOCK *rwl);
```

The ep_thr_*_check routines check the structures for consistency and print an error; this is only for debugging. *This should be expanded to include spawning threads etc.; for the time being just use the pthreads primitives.*

There is also a basic thread pool implementation:

```
#include <ep/ep_thr.h>

void
ep_thr_pool_init(
        int min_threads,
        int max_threads,
        uint32_t flags);

void
ep_thr_pool_run(
        void (*func)(void *),
        void *arg);
```

Thread pools are initially started with `min_threads` workers (which may be zero; defaults to the `libep.thr.pool.min_workers` administrative parameter, or 1 if that is not set). Threads will be spawned as necessary up to `max_threads` total workers (defaults to `libep.thr.pool.max_workers`; if that is not set, defaults to twice the number of cores available).

Threads are run in essentially the same way as spawning a pthreads thread; this is really just a convenience wrapper around that so resources can be better controlled.

12 LOGGING

Messages may be logged together with a status code:

```
#include <ep/ep_log.h>

void
ep_log_init(
        const char *tag,
        int logfac,
```

```
        FILE *logfile);

void
ep_log_addmethod(
        void (*func)(void *ctx, EP_STAT estat, const char *fmt, va_list ap),
        void *ctx,
        int minsev);

void
ep_log(
        EP_STAT estat,
        const char *fmt,
        ...);

void
ep_logv(
        EP_STAT estat,
        const char *fmt,
        va_list va);
```

The `ep_log` and `ep_logv` routines send information to various system logs. The default is to send to `syslog`(3) and to `stderr`. You can disable or change this by calling `ep_log_init` before the first logging call, and extend it by passing another logging method to `ep_log_addmethod`, which causes `func` to be called whenever a message with severity at least the value of the `minsev` parameter (`EP_STAT_SEV_OK`, `EP_STAT_SEV_WARN`, `EP_STAT_SEV_ERROR`, `EP_STAT_SEV_SEVERE`, `EP_STAT_SEV_ABORT`).

The status code is logged together with the printf-style message. The syslog severity is determined from the severity of the status code: `OK` codes log an `LOG_INFO` message, `WARN` codes log a `LOG_WARNING` message, `ERROR` codes log a `LOG_ERR` message, `SEVERE` codes log a `LOG_CRIT` message, and `ABORT` codes log a `LOG_ALERT` message.

13 ARGUMENT CRACKING

Not implemented at this time. To help with parsing command line arguments. A descriptor is declared as follows:

```
    #include <ep/ep_crackargv.h>

    unsigned long       NTests;
    long                Seed;
    static char         *FileName;

    EP_CAV_DESCR        ArgvDescriptor[] =
    {
        { "debug",              EP_CAV_TYPE(debug),     'D',    5,
                "Debug",                "debug-flags",          NULL,
                EP_CAV_FLAG_NOARGS
        },
        { "ntests",             EP_CAV_TYPE(ulong),     'n',    1,
                "Number of tests",      NULL,                   &NTests,
                EP_CAV_FLAG_REQUIRED
        },
        { "seed",               EP_CAV_TYPE(long),      's',    1,
                NULL,                   NULL,                   &Seed,
                EP_CAV_FLAG_NONE
        }
        { NULL,                 EP_CAV_TYPE(string),    '\0',   0,
                NULL,                   NULL,                   &FileName,
                EP_CAV_FLAG_NONE
```

```
        }
        EP_CAV_DESCR_END
    };
```

The ep_crackargv routine is then called with an argument vector and a descriptor:

```
    stat = ep_crackargv(const char **argv, const EP_CAV_DESCR *descr);
```

The argument vector is then matched to the descriptor and appropriate bindings done. Duplicate and missing flags are diagnosed and all conversions are done.

The fields in the descriptor are:

- The long name. On Unix, this is matched against arguments beginning "--". This is case independent.
- The data type. This is always EP_CAV_TYPE(something), which calls the conversion routine named ep_cvt_txt_to_something passing it the value as a text string and a pointer to the output location (see below).
- The short (single character) name. On Unix, this is matched against arguments beginning "-". Flags without values can be combined into one flag -- that is, if "-a -b" sets two boolean flags, "-ab" does the same thing.
- The number of bytes of the long name that must match. This allows abbreviation of names. See below.
- The prompt. If flags are required and a prompt is available, ep_crackargv can prompt for missing parameters. Not yet implemented.
- The usage message to describe this parameter. Defaults to the long name.
- The value pointer. A pointer to the data area in which to store the results. If NULL, this parameter cannot accept a value.
- Flag bits, as described below.

Long flag names can be abbreviated. All characters of the command line must match the descriptor, but only the number indicated in the "must match" field need be present. For example, given a name in the descriptor of "ntests" with a "must match" field of 2 will match "--ntests", "--ntes", "--nt", but not "--ntext", "--n", or "--nteststotry".

Flag bits include:

EP_CAV_FLAG_NONE
: No special processing

EP_CAV_FLAG_NOARGS
: This parameter takes no arguments (e.g., a boolean)

EP_CAV_FLAG_NOMORE
: This consumes all remaining arguments (normally EP_CAV_TYPE(Vector))

EP_CAV_FLAG_MULTVAL
: There can be multiple values for this parameter (only relevant for flags)

EP_CAV_FLAG_REQUIRED
: If this parameter is missing it is an error

Predefined types and the type of the corresponding value pointer are:

bool bool_t * Booleans. Should have EP_CAV_FLAG_NOARGS. string const char ** Strings. long long * Signed long integers. ulong unsigned long * Unsigned long integers. double double * Double point floating point.

vector const char *** Vectors. Must have the EP_CAV_FLAG_MULTVAL flag set. Can only be one, and it must be at the end. debug NULL Sets debug flags

To appear:

int8 int8_t * uint8 uint8_t * int16 int16_t * uint16 uint16_t * int32 int32_t * uint32 uint32_t * int64 int64_t * uint64 uint64_t * admparam const char *

Administrative parameters (see `ep_adm_getintparam` and `ep_adm_getstrparam`). The value pointer is the name of the parameter to set.

New parameter types can be trivially created by defining new routines named `ep_cvt_txt_to_`*type* that take a `const char *` as input and a *type* `*` output pointer. They return `EP_STAT`. Conversion errors should fail.

14 MISCELLANEOUS STUFF

```
    EP_UT_BITSET(uint32 bits,           // return true if any bits...
                uint32_t word)          // ... are set in word

/*
    EP_UT_SETBIT(uint32_t bits,         // set these bits...
                uint32_t word)          // ... in this word

    EP_UT_CLRBIT(uint32 bits,           // clear these bits...
                uint32_t word)          // ... in this word
*/

    EP_UT_BITMAP(               // declare bitmap
        name,                           // name of bitmap to declare
        nbits)                          // number of bits in map

    EP_UT_CLRBITMAP(            // clear bitmap
        name)                           // bitmap to clear

    EP_UT_BITNSET(int bitn,             // true if bit number bitn is set...
                bitmap)                 // ... in this map

    EP_UT_SETBITN(int bitn,             // set bit number bitn...
                bitmap)                 // ... in this map

    EP_UT_CLRBITN(int bitn,             // clear bit number bitn...
                bitmap)                 // ... in this map

    EP_GEN_DEADBEEF                     // a value you can use to trash memory
```

Warning There is no checking for the BITMAP routines (`EP_UT_BITNSET`, `EP_UT_SETBITN`, `EP_UT_CLRBITN`) to ensure that the bit indicated is in range for the size of the bitmap.

15 INTERACTION WITH THE ENVIRONMENT

15.1 Global Administrative Parameters

There are a bunch of parameters that we would prefer to be settable at run time. We'll model this on sysctl(8). Before accessing parameters you must read them using `ep_adm_readparams`. This routine takes a `name` and then looks for a file in a search path. That path may be set using the PARAM_PATH environment variable, and defaults to:

```
.ep_adm_params:~/.ep_adm_params:/usr/local/etc/ep_adm_params:/etc/ep_adm_params
```

For example, searching for a name such as "`defaults`" will first try to read the file `.ep_adm_params/defaults`. If that is found the search stops, otherwise it trys `~/.ep_adm_params/defaults`, and so forth. New values replace old ones, so programs that want to search more than one file should start with the most generic one and continue to the least generic one.

```
#include <ep/ep_adm.h>

void
ep_adm_readparams(
            const char *name)       // basename of the parameter file

int
ep_adm_getintparam(
            const char *name,       // name of the parameter
            int default)            // value if parameter not set

long
ep_adm_getlongparam(
            const char *name,       // name of the parameter
            long default)           // value if parameter not set

bool
ep_adm_getboolparam(
            const char *name,       // name of the parameter
            bool default)           // value if parameter not set

const char *
ep_adm_getstrparam(
            const char *name,       // name of the parameter
            char *default)          // value if parameter not set
```

Names are structured kind of like `sysctl` arguments or X Resource names, e.g., "`libep.stream.hfile.bsize`". You must read one or more parameter files before getting parameters.

15.2 Terminal Video Sequences and Characters

Mostly for debugging use. Right now compiled in for ANSI xterms.

```
#include <ep/ep_string.h>

struct epVidSequences
{
    const char    *vidnorm;       // set video to normal
    const char    *vidbold;       // set video to bold
    const char    *vidfaint;      // set video to faint
    const char    *vidstout;      // set viadeo to "standout"
    const char    *viduline;      // set video to underline
    const char    *vidblink;      // set video to blink
    const char    *vidinv;        // set video to invert
    const char    *vidfgblack;    // set foreground black
    const char    *vidfgred;      // set foreground red
    const char    *vidfggreen;    // set foreground green
    const char    *vidfgyellow;   // set foreground yellow
    const char    *vidfgblue;     // set foreground blue
    const char    *vidfgmagenta;  // set foreground magenta
    const char    *vidfgcyan;     // set foreground cyan
    const char    *vidfgwhite     // set foreground white
    const char    *vidbgblack;    // set background black
    const char    *vidbgred;      // set background red
    const char    *vidbggreen;    // set background green
```

```
        const char    *vidbgyellow;   // set background yellow
        const char    *vidbgblue      // set background blue
        const char    *vidbgmagenta;  // set background magenta
        const char    *vidbgcyan;     // set background cyan
        const char    *vidbtwhite;    // set background white
    } *EpVid;

    struct epCharSequences
    {
        const char    *lquote;        // left quote sequence
        const char    *rquote;        // right quote sequence
        const char    *copyright;     // copyright symbol
        const char    *degree;        // degree symbol
        const char    *micro;         // micro symbol
        const char    *plusminus;     // +/- symbol
        const char    *times;         // mathematical times symbol
        const char    *divide;        // mathematical division symbol
        const char    *null;          // "null" symbol
        const char    *notequal;      // mathematical "not equal" symbol
        const char    *unprintable;   // substitution for unprintable characters
        const char    *paragraph;     // paragraph symbol
        const char    *section;       // section symbol
        const char    *notsign;       // logical not symbol
        const char    *infinity;      // infinity symbol
    } *EpChar;

    EP_STAT   ep_str_vid_set(         // set video style
        const char    *type);         // NULL, "none", or "ansi"

    EP_STAT   ep_str_char_set(        // set special characters
        const char    *type);         // character set (see below)
```

These structures contain character sequences used for printing video controls and special characters respectively. The `ep_str_vid_set` routines allows you to choose the video escape sequences. Passing `NULL` causes an educated guess at the default on the basis of the `TERM` environment variable. Any `TERM` setting beginning with "`xterm`" is the same as specifying "`ansi`" as the type and anything else is the same as specifying "`none`" as the type (which sets all the video strings to null strings). Blessedly, xterm doesn't seem to render blink, nor faint or standout. Bold and blink are both rendered in bold. So, for best results use bold, uline, and inv (and of course norm).

The `ep_str_char_set` allows you to set special character encodings. Its parameter may be `NULL` (which guesses based on the `LANG` environment variable), "`ascii`", "`iso-8859-1`", "`iso-latin-1`", "`utf-8`", or "`utf8`". The mappings are shown in the following table:

Name	ASCII	Other Charset
lquote	`	«
rquote	'	»
copyright	(c)	©
degree	deg	°
micro	u	µ
plusminus	+/-	±
times	*	×
divide	/	÷

null	NULL	∅
notequal	!=	≠
unprintable	?	⍰
paragraph	pp.	¶
section	sec.	§
notsign	(not)	¬
infinity	(inf)	∞

Other values may be added to this table as needed.

Example:

```
    fprintf(ep_dbg_getfile(), "Input was %s%s%s\n",
                        EpVidSeq.lquote, input, EpVidSeq.rquote);
```

15.3 Startup/Shutdown

If running under `systemd`-based versions of Linux, it is possible to signal status changes to the startup environment, notably about system startup and shutdown. This is only relevant for programs that run as system daemons.

```
#include <ep/ep_sd.h>

void
ep_sd_notifyf(
        const char *fmt,
        ...);
```

The `fmt` and any arguments are printed a'la `printf`(3) to a system buffer that is delivered to `systemd`. The format of that buffer is defined by `sd_notify`(3) and is not detailed here. Roughly, each line of the output looks like an environment variable definition. For example:

```
// to inform the system that this service is ready:
ep_sd_notifyf(
        "READY=1");

// to inform the system that this service is shutting down:
ep_sd_notifyf(
        "STOPPING=1\n"
        "STATUS=Exiting on user signal %d\n",
        sig);
```

Important values include `READY=1`, `RELOADING=1`, `WATCHDOG=1`, and `STOPPING=...`.

If the `EP_OSCF_HAS_SD_NOTIFY` flag is set to zero at compilation time, this call is a no-op. This is the case on most systems.

This should really abstract the syntax out more, rather than making it `systemd` specific, so that it can potentially be used on other systems.

16 TRANSLATIONS

Simple string translations for certain external formats, for example as might be used by URLs or Quoted-Printable.

```
#include <ep/ep_xlate.h>

int
ep_xlate_in(
    const void *ext,                // external (encoded) string input
    uchar_t *out,                   // pointer to output buffer
    size_t olen,                    // length of output buffer
    char stopchar,                  // input char to stop at
    uint32_t how)                   // what kind of translations to do
```

Translates an external form (with encodings) into internal form (potentially 8-bit binary). Returns the number of output bytes. The "how" parameter tells what translations to do -- they can be combined:

EP_XLATE "how" Bits

EP_XLATE_PERCENT	Translate "%xx" like ESMTP
EP_XLATE_BSLASH	Translate backslash escapes like C
EP_XLATE_AMPER	Translate "&name;" like HTML
EP_XLATE_PLUS	Translate "+xx" like DNSs
EP_XLATE_EQUAL	Translate "=xx" like quoted-printable
EP_XLATE_8BIT	Translate 8-bit characters (`ep_xlate_out` only)
EP_XLATE_NPRINT	Translate non-printable characters (`ep_xlate_out` only)

```
int
ep_xlate_out(
    const void *in,                 // internal (not encoded) input string
    size_t ilen,                    // length of in
    FILE *osp,                      // encoded output stream pointer
    const char *forbid,             // list of characters to encode
    uint32_t how)                   // how to do output translations
```

Unlike input, it doesn't make sense to list more than one of `EP_XLATE_PERCENT`, `EP_XLATE_BSLASH`, `EP_XLATE_EQUAL`, and `EP_XLATE_PLUS`. If none are listed, `EP_XLATE_PLUS` is assumed. `EP_XLATE_8BIT` can be added to encode all 8-bit characters and `EP_XLATE_NPRINT` translates all unprintable characters (as determined by `isprint(3)`, which generally does understand locales). Returns the number of bytes output to the indicated osp.

Note [[Arguably they should both use streams for both input and output.]]

There are also routines to encode/decode binaries in base64.

```
#include <ep/ep_b64.h>

EP_STAT
ep_b64_encode(
    const void *bin,                // binary data to encode
    size_t bsize,                   // size of bin to encode
    char *txt,                      // text output buffer
    size_t tsize,                   // size of output buffer
    const char *encoding)           // type of encoding (see below)

EP_STAT
ep_b64_decode(
    const char *txt,                // text to decode
    size_t tsize,                   // stop after tsize characters
    void *bin,                      // binary output buffer
```

```
        size_t bsize,                   // size of bin buffer
        const char *encoding)           // type of encoding (see below)

    #define EP_B64_NOWRAP       0x00    // never wrap lines
    #define EP_B64_WRAP64       0x01    // wrap at 64 characters
    #define EP_B64_WRAP76       0x02    // wrap at 76 characters
    #define EP_B64_WRAPMASK     0x03    // bit mask for wrapping
    #define EP_B64_PAD          0x04    // pad with '='
    #define EP_B64_IGNCRUD      0x08    // ignore unrecognized chars

    // encodings for common standards
    #define EP_B64_ENC_MIME     "+/N"   // WRAP76  PAD  IGNCRUD
    #define EP_B64_ENC_PEM      "+/E"   // WRAP64  PAD -IGNCRUD
    #define EP_B64_ENC_URL      "-_@"   // NOWRAP -PAD -IGNCRUD
```

The encoding is a three character string. The first two characters are used to represent the codes for positions 62 and 63 (these are the only two that are not letters or digits). The third is used as flag bits to indicate variations for various encodings. The three most common strings are included as defined constants (for MIME email, Privacy Enhanced Mail, and URLs).

17 XXX TO BE DONE

- Document ep_pprint.
- Document ep_dumpfds (shows open file descriptors (for debugging).
- Document ep_fread_unlocked.