libHX 3.23
Documentation

August 28, 2018

Contents

1 Introduction

libHX collects many useful day-to-day functions, intended to reduce the amount of otherwise repeatedly open-coded instructions.

2 Overview

- Maps (key-value pairs) (section 13)
  Originally created to provide a data structure like Perl's associative arrays. Different map types and characteristics are available, such as hash-based or the traditional rbtree.

- Deques (section 14)
  Double-ended queues, implemented as a doubly-linked list with sentinels, are suitable for both providing stack and queue functionality.

- Inline doubly-linked list, uncounted and counted (sections 15 and 16)
  Light-weight linked lists as used in the Linux kernel.

- Common string operations (section 17)
  basename, chomp, dirname, getl(ine), split, strlcat/strlcpy, strlower/-upper, str*trim, strsep, etc.

- Memory containers, auto-sizing string operations (section 18)
  Scripting-like invocation for string handling — automatically doing (re)allocations as needed.

- String formatter (section 19)
  HXfmt is a small template system for by-name variable expansion. It can be used to substitute placeholders in format strings supplied by the user by appropriate expanded values defined by the program.

- Directory creation, traversal, removal, and file copying (sections 21, 22 and 23)

- Option parsing (section 24)
  Table-/callback-based option parser that works similar to Perl's `Getopt::Long` — no open-coding but a single "atomic" invocation.

- Shell-style config parser (section 25)
  Configuration file reader for Shell-style "configuration" files with key-value pairs, as usually foudn in `/etc/sysconfig`.

- Random number gathering (section 26)
  Convenient wrapper that uses kernel-provided RNG devices when available.

- External process invocation (section 27)
  Setting up pipes for the standard file descriptors for sending/capturing data to/from a program.

- *a bit more beyond that ... Miscellaneous*

3 Resources

As of this writing, the repository is located at

- `git://libhx.git.sf.net/gitroot/libhx/libhx` — clone URL
- `http://libhx.git.sf.net/` — gitweb interface
- `http://libhx.sf.net/` — home page (and link to tarballs)
- `http://freecode.com/projects/libhx/` — Freecode page (useful for automatic notification of new releases)

4 Installation

libHX uses GNU autotools as a build environment, which means that all you have to run as a end-user is the `configure` with any options that you need, plus the usual `make` and `make install` as desired.

Pay attention to multi-lib Linux distributions where you most likely need to specify a different libdir instead of using the default "lib". In case of the Debian-style multi-arch/multi-lib proposal (`http://wiki.debian.org/Multiarch`):

```
$ ./configure --libdir='${prefix}/lib/x86_64-linux-gnu'
```

and the classic-style 32-64 2-lib distributions:

```
$ ./configure --libdir='${prefix}/lib64'
```

4.1 Requirements

- GNU C Compiler 3.3.5 or newer. Other compilers (non-GCC) have not been tested in months — use at your own risk.
- approximately 80–160 KB of disk space on Linux for the shared library (depends on platform) and header files.

A C++ compiler is only needed if you want to build the C++ test programs that come with libHX. By default, if there is no C++ compiler present, these will not be built.

- No external libraries are needed for compilation of libHX. Helper files, like `libxml_helper.h`, may reference their include files, but they are not used during compilation.

5 Portability notice

libHX runs on contemporary versions of Linux, Solaris and the three BSD distributions. It might even work on Microsoft Windows, but this is not tested very often, if at all. Overly old systems, especially Unices, are not within focus. While AIX 5.3 might still classify as contemporary, strangelets like "Ultrix" or "Dynix" you can find in the autotools-related file `config.guess` are some that are definitely not.

Furthermore, a compiler that understands the C99 or GNU89 standard is required. The integer type "int" should at best have 32 bits at least. There is no ultra-portable version as of this writing, but feel free to start one akin to the "p" variants of OpenBSD software such as OpenSSH.

6 History

The origins of libHX trace back, even crossing a language boundary, to when the author started on using Perl in 1999. Some tasks were just too damn useful to be open-coded every time. Two such examples are what is these days known as `HX_basename` and `HX_mkdir`. The name does not relate to anyone's initials; it is a result of a truncation of the author's nick used years ago.

Around the beginning of 2003, the author also started on the C programming language and soon the small library was converted from Perl to C. The libHX library as of today is the result of working with C ever since, and naturally grew from there to support whatever the author was in need of.

The "correct" name for libHX is with an uppercase "H" and uppercase "X", and the same is used for filenames, such as "libHX.so"[1].

[1]Software projects may choose to entirely lowercase the project name for use in filenames, such as the Linux kernel which is released as `linux-${`*version*`}.tar.bz2`, or the project may choose to keep the name for filenames, like Mesa and SDL do. libHX is of the latter.

Part I
General

Many functions are prefixed with "`HX_`" or "`HXsubsys_`", as are structures (sometimes without underscores, be sure to check the syntax and names), to avoid name clashes with possibly existing files. Functions that are not tied to a specific data structure such as most of the string functions (see chapter 17) use the subsystem-less prefix, "`HX_`". Functions from a clearly-defined subsystem, such as map or deque, augment the base prefix by a suffix, forming e. g. "`HXmap_`".

7 Initialization

```
#include <libHX/init.h>

int HX_init(void);
void HX_exit(void);
```

Before using the library's functions, `HX_init` must be called. This function will initialize any needed state libHX needs for itself, if any. It is designed to be invoked multiple times, such as for example, from different libraries linking to libHX itself, and will refcount. On success, >0 is returned. If there was an error, it will return a negative error code or zero. `HX_exit` is the logical counterpart of notifying that the library is no longer used.

8 Type-checking casts

The C++ language provides so-called "new-style casts", referring to the four template-looking invocations `static_cast<>`, `const_cast<>`, `reinterpret_cast<>` and `dynamic_cast<>`. No such blessing was given to the C language, but still, even using macros that expand to the olde cast make it much easier to find casts in source code and annotate why something was casted, which is already an improvement. — Actually, it *is* possible to do a some type checking, using some GCC extensions, which augments these macros from their documentary nature to an actual safety measure.

8.1 `reinterpret_cast`

`reinterpret_cast()` maps directly to the old-style typecast, `(type)(expr)`, and causes the bit pattern for the expr rvalue to be "reinterpreted" as a new type. You will notice that "reinterpret" is the longest of all the `*_cast` names, and can easily cause your line to grow to 80 columns (the good maximum in many style guides). As a side effect, it is a good indicator that something potentially dangerous might be going on, for example converting intergers from/to pointer.

```
#include <libHX/defs.h>

int i;
/* Tree with numeric keys */
tree = HXhashmap_init(0);
for (i = 0; i < 6; ++i)
        HXmap_add(tree, reinterpret_cast(void *,
                  static_cast(long, i)), my_data);
```

From \ To	c*section	sc*	uc*	Cc*	Csc*	Cuc*
`char *`	✓	✓	✓	✓	✓	✓
`signed char *`	✓	✓	✓	✓	✓	✓
`unsigned char *`	✓	✓	✓	✓	✓	✓
`const char *`	–	–	–	✓	✓	✓
`const signed char *`	–	–	–	✓	✓	✓
`const unsigned char *`	–	–	–	✓	✓	✓

Table 1: Accepted conversions for `signed_cast()`

8.2 `signed_cast`

This tag is for annotating that the cast was solely done to change the signedness of pointers to char — and only those. No integers etc. The intention is to facilitate working with libraries that use `unsigned char *` pointers, such as libcrypto and libssl (from the OpenSSL project) or libxml2, for example. See table 1 for the allowed conversions. C++ does *not* actually have a `signed_cast<>`, and one would have to use `reinterpret_cast<>` to do the conversion, because `static_cast<>` does not allow conversion from `const char *` to `const unsigned char *`, for example. (libHX's `static_cast()` would also throw at least a compiler warning about the different signedness.) libHX does provide a `signed_cast<>` for C++ though. This is where `signed_cast` comes in.

8.3 `static_cast`

Just like C++'s `static_cast<>`, libHX's `static_cast()` verifies that `expr` can be implicitly converted to the new type (by a simple `b = a`). Such is mainly useful for forcing a specific type, as is needed in varargs functions such as `printf`, and where the conversion actually incurs other side effects, such as truncation or promotion:

```
/* Convert to a type printf knows about */
uint64_t x = something;
printf("%llu\n", static_cast(unsigned long long, x));
```

Because there is no format specifier for `uint64_t` for `printf`, a conversion to an accepted type is necessary to not cause undefined behavior. The author has seen code that did, for example, `printf("%u")` on a "long", which only works on architectures where `sizeof(unsigned int)` happens to equal `sizeof(unsigned long)`, such as x86_32. On x86_64, an `unsigned long` is usually twice as big as an `unsigned int`, so that 8 bytes are pushed onto the stack, but `printf` only unshifts 4 bytes because the developer used "`%u`", leading to misreading the next variable on the stack.

```
/* Force promotion */
double a_quarter = static_cast(double, 1) / 4;
```

Were "1" not promoted to double, the result in `q` would be zero because 1/4 is just an integer division, yielding zero. By making one of the operands a floating-point quantity, the compiler will instruct the FPU to compute the result. Of course, one could have also written "`1.0`" instead of `static_cast(double, 1)`, but this is left for the programmer to decide which style s/he prefers.

```
/* Force truncation before invoking second sqrt */
double f = sqrt(static_cast(int, 10 * sqrt(3.0 / 4)));
```

And here, the conversion from `double` to `int` incurs a (wanted) truncation of the decimal fraction, that is, rounding down for positive numbers, and rounding up for negative numbers.

8.3.1 Allowed conversions

- **Numbers**
  Conversion between numeric types, such as `char`, `short`, `int`, `long`, `long long`, `intN_t`, both their signed and unsigned variants, `float` and `double`.
- **Generic Pointer**
  Conversion from `type *` to and from `void *`. (Where `type` may very well be a type with further indirection.)
- **Generic Pointer (const)**
  Conversion from `const type *` to and from `const void *`.

8.4 `const_cast`

`const_cast` allows to add or remove "const" qualifiers from the type a pointer is pointing to. Due to technical limitations, it could not be implemented to support arbitrary indirection. Instead, `const_cast` comes in three variants, to be used for indirection levels of 1 to 3:

- `const_cast1(type *, expr)` with `typeof(expr) = type *`. (Similarly for any combinations of const.)
- `const_cast2(type **, expr)` with `typeof(expr) = type **` (and all combinations of const in all possible locations).
- `const_cast3(type ***, expr)` with `typeof(expr) = type ***` (and all combinations...).

As indirection levels above 3 are really unlikely[2], having only these three type-checking cast macros was deemed sufficient. The only place where libHX even uses a level-3 indirection is in the option parser.

`int **`	`int *const *`
`const int **`	`const int *const *`

Table 2: Accepted expr/target types for `const_cast2`; example for the "int" type
Conversion is permitted when expression and target type are from the table.

It is currently not possible to use `const_cast`1/2/3 on pointers to structures whose member structure is unknown.

[2]See "Three Star Programmer"

9 Macros

All macros in this section are available through `#include <libHX/defs.h>`.

9.1 Preprocessor

```
#define HX_STRINGIFY(s)
```

Transforms the expansion of the argument `s` into a C string.

9.2 Sizes

```
#define HXSIZEOF_Z16
#define HXSIZEOF_Z32
#define HXSIZEOF_Z64
```

Expands to the size needed for a buffer (including '\0') to hold the base-10 string representation of a 16-, 32- or 64-bit integer.

9.3 Locators

```
long offsetof(type, member);
output_type *containerof(input_type *ptr, output_type, member);
size_t FIELD_SIZEOF(struct_type, member);
output_type HXtypeof_member(struct_type, member);
```

In case `offsetof` and `containerof` have not already defined by inclusion of another header file, libHX's defs.h will define these accessors. `offsetof` is defined in `stddef.h` (for C) or `cstddef` (C++), but inclusion of these is not necessary if you have included `defs.h`. `defs.h` will use GCC's `__builtin_offsetof` if available, which does some extra sanity checks in C++ mode.

`offsetof` calculates the offset of the specified member in the type, which needs to be a struct or union.

`containerof` will return a pointer to the struct in which `ptr` is contained as the given member.

```
struct foo {
        int bar;
        int baz;
};

static void test(int *ptr)
{
        struct foo *self = containerof(baz, struct foo, baz);
}
```

`FIELD_SIZEOF` (formerly `HXsizeof_member`) and `HXtypeof_member` are convenient shortcuts to get the size or type of a named member in a given struct:

```
char padding[FIELD_SIZEOF(struct foo, baz)];
```

9.4 Array size

```
size_t ARRAY_SIZE(type array[]); /* implemented as a macro */
```

Returns the number of elements in `array`. This only works with true arrays (`type[]`), and will not output a meaningful value when used with a pointer-to-element (`type *`), which is often used for array access too.

9.5 Compile-time build checks

```
int BUILD_BUG_ON_EXPR(bool condition); /* implemented as a macro */
void BUILD_BUG_ON(bool condition); /* implemented as a macro */
```

Causes the compiler to fail when `condition` evaluates to true. If not implemented for a compiler, it will be a no-op. `BUILD_BUG_ON` is meant to be used as a standalone statement, while `BUILD_BUG_ON_EXPR` is for when a check is to occur within an expression, that latter of which is useful for within macros when one cannot, or does not want to use multiple statements.

```
type DEMOTE_TO_PTR(type expr); /* macro */
```

Changes the type of expr to pointer type: If `expr` of array type class, changes it to a pointer to the first element. If `expr` of function type class, changes it to a pointer to the function.

```
int main(void);
int (*fp)(void);
char a[123];
DEMOTE_TO_PTR(main); /* yields int (*)(void); */
DEMOTE_TO_PTR(fp);   /* also yields int (*)(void); */
DEMOTE_TO_PTR(a);    /* yields char * */
```

9.6 UNIX file modes

```
#define S_IRUGO   (S_IRUSR | S_IRGRP | S_IROTH)
#define S_IWUGO   (S_IWUSR | S_IWGRP | S_IWOTH)
#define S_IXUGO   (S_IXUSR | S_IXGRP | S_IXOTH)
#define S_IRWXUGO (S_IRUGO | S_IWUGO | S_IXUGO)
```

The defines make it vastly easier to specify permissions for large group of users. For example, if one wanted to create a file with the permissions `rw-r--r--` (ignoring the umask in this description), `S_IRUSR | S_IWUSR` can now be used instead of the longer `S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH`.

9.7 VC runtime format specifiers

The Microsoft Visual C runtime (a weak libc) uses non-standard format specifiers for certain types. Whereas C99 specifies "z" for `size_t` and "ll" for long long, MSVCRT users must use "I" and "I64" (forming `%Id` instead of `%zd` for `ssize_t`, for example). libHX provides two convenience macros for this:

```
#define HX_SIZET_FMT    "z" or "I"
#define HX_LONGLONG_FMT "ll" or "I64"
```

These may be used together with printf or scanf:

```
printf("struct timespec is of size %" HX_SIZET_FMT "u\n",
       sizeof(struct timespec));
```

10 Miscellaneous functions

```
#include <libHX/misc.h>

int HX_ffs(unsigned long z);
int HX_fls(unsigned long z);
void HX_hexdump(FILE *fp, const void *ptr, unsigned int len);
void HX_zvecfree(char **);
unsigned int HX_zveclen(const char *const *);
```

HX_ffs Finds the first (lowest-significant) bit in a value and returns its position, or -1 to indicate failure.

HX_fls Finds the last (most-significant) bit in a value and returns its position, or -1 to indicate failure.

HX_hexdump Outputs a nice pretty-printed hex and ASCII dump to the filedescriptor `fp`. `ptr` is the memory area, of which `len` bytes will be dumped.

HX_zvecfree Frees the supplied Z-vector array. (Frees all array elements from the first element to (excluding) the first `NULL` element.)

HX_zveclen Counts the number of array elements until the first `NULL` array element is seen, and returns this number.

11 Time functions

Time in POSIX systems is represented in `struct timespec`. This structure is composed of two members: one integer for the number of full seconds in the time value, and one integer for the number of nanoseconds that remain when subtracting the full seconds from the time value. POSIX leaves it unspecified how negative time is to be represented with this structure, so I have devised an algebra for use with the same struct that gives negative time support.

Since integers often cannot store negative zero (due to e. g. use of 2s complements in the language implementation), we will store the minus sign in the nanosecond member if the integral second part is zero. This gives us the property that we can test for negative time by looking for whether at least one member of the structure is negative. Also, we want to avoid storing the minus in both members to somewhat aid the pretty-printing construct often seen,

```
printf("%ld.%09ld\n", (long)ts.tv_sec, ts.tv_nsec);
```

The number of combinations of a (non-zero) negative number, zero and a (non-zero) positive number is small, so we can actually just exhaustively list them all.

Representation	Time value	R	T	R	T
$\{-1,-1\}$	illegal	$\{0,-1\}$	-0.1 s	$\{1,-1\}$	illegal
$\{-1,0\}$	-1.0 s	$\{0,0\}$	0.0 s	$\{1,0\}$	1.0 s
$\{-1,1\}$	-1.1 s	$\{0,1\}$	0.1 s	$\{1,1\}$	1.1 s

The set of so-extended valid timespecs is therefore:

$$K = \left\{(i,f) : i,f \in \mathbb{Z} \wedge i \neq 0 \wedge 0 \leq f < 10^9\right\} \cup \left\{(i,f) : i = 0 \wedge f \in \mathbb{Z} \wedge -10^9 < f < 10^9\right\}$$

11.1 Function list

```
#include <libHX/misc.h>

bool HX_timespec_isneg(const struct timespec *p);
struct timespec *HX_timespec_neg(struct timespec *result,
    const struct timespec *p);
struct timespec *HX_timespec_add(struct timespec *result,
    const struct timespec *p, const struct timespec *q);
struct timespec *HX_timespec_sub(struct timespec *delta,
    const struct timespec *p, const struct timespec *q);
struct timespec *HX_timespec_mul(struct timespec *delta,
    const struct timespec *p, int f);
struct timespec *HX_timespec_mulf(struct timespec *delta,
    const struct timespec *p, double f);
struct timeval *HX_timeval_sub(struct timeval *delta,
    const struct timeval *p, const struct timeval *q);
int HX_time_compare(const struct stat *a, const struct stat *b, int mode);
```

HX_timespec_isneg Determines whether a timespec structure represents (non-zero) negative time.

HX_timespec_neg Computes the negation of the time specified by `p`. `result` and `p` may point to the same structure.

HX_timespec_add Calculates the sum of the two times specified by `p` and `q`, which are of type `struct timespec`. Any of `result`, `p` and `q` may point to the same structure.

HX_timespec_sub Calculates the difference between the two timepoints p and q, which are of type `struct timespec` (nanosecond granularity).

HX_timespec_mul Multiplies the time quantum in `p` by `f`.

HX_timespec_mulf Multiplies the time quantum in `p` by `f`.

HX_timeval_sub Calculates the difference between the two timepoints p and q, which are of type `struct timeval` (microsecnod granularity).

HX_time_compare Compares the timestamps from two `struct stat`s. `mode` indicates which field is compared, which can either be `'a'` for the access time, `'c'` for the inode change time, `'m'` for the modification time, or `'o'` for the creation time (where available). Returns a negative number if the time in `a` is less than `b`, zero when they are equal, or a positive number greater than zero if `a` is greater than `b`.

The macros `HX_TIMESPEC_FMT` and `HX_TIMESPEC_EXP` can be used for passing and printing a `struct timespec` using the *`printf` function family:

```
struct timespec p;
clock_gettime(CLOCK_MONOTONIC, &p);
printf("Now: " HX_TIMESPEC_FMT, HX_TIMESPEC_EXP(&p));
```

Similarly, `HX_TIMEVAL_FMT` and `HX_TIMEVAL_EXP` exist for the older `struct timeval`.

12 Bitmaps

```
#include <libHX/misc.h>

size_t HXbitmap_size(type array, unsigned int bits);
void HXbitmap_set(type *bmap, unsigned int bit);
void HXbitmap_clear(type *bmap, unsigned int bit);
bool HXbitmap_test(type *bmap, unsigned int bit);
```

All of these four are implemented as macros, so they can be used with any integer type that is desired to be used.

HXbitmap_size Returns the amount of "type"-based integers that would be needed to hold an array of the requested amount of bits.

HXbitmap_set Set the specific bit in the bitmap.

HXbitmap_clear Clear the specific bit in this bitmap.

HXbitmap_test Test for the specific bit and returns `true` if it is set, otherwise `false`.

12.0.1 Example

```
#include <stdlib.h>
#include <string.h>
#include <libHX/misc.h>

int main(void)
{
        unsigned long bitmap[HXbitmap_size(unsigned long, 128)];

        memset(bitmap, 0, sizeof(bitmap));
        HXbitmap_set(bitmap, 49);
        return HXbitmap_get(bitmap, HX_irand(0, 128)) ?
               EXIT_SUCCESS : EXIT_FAILURE;
}
```

Part II
Data structures

13 Maps

A map is a collection of key-value pairs. (Some languages, such as Perl, also call them "associative array" or just "hash", however, the underlying storage mechanism may not be an array or a hash, however.) Each key is unique and has an associated value. Keys can be any data desired; HXmap allows to specify your own key and data handling functions so they can be strings, raw pointers, or complex structures.

To access any map-related functions, `#include <libHX/map.h>`.

13.1 Structural definition

The `HXmap` structure is a near-opaque type. Unlike the predecessor map implementation `struct HXbtree` from libHX 2.x, the 3.x API exposes much less fields.

```
struct HXmap {
        unsigned int items, flags;
};
```

items The number of items in the tree. This field tracks the number of items in the map and is used to report the number of elements to the user, and is updated whenever an element is inserted or removed from the map. The field must not be changed by user.

flags The current behavior flags for the map. While implementation-private bits are exposed, only `HXMAP_NOREPLACE` is currently allowed to be (un)set by the developer while a map exists.

For retrieving elements from a tree, some functions work with `struct HXmap_node`, which is defined as follows:

```
struct HXmap_node {
        union {
                void *key;
                const char *const skey;
        };
        union {
                void *data;
                char *sdata;
        };
};
```

key The so-called primary key, which uniquely identifies an element (a key-value pair) in the map. The memory portions that make up the key must not be modified. (If the key changes, so does its hash value and/or position index, and without taking that into account, writing to the key directly is going to end up with an inconsistent state. To change the key, you will need to delete the element and reinsert it with its new key.)

skey A convenience type field for when the map's keys are C strings. It is useful for use with e. g. `printf` or other varargs function, which would otherwise require casting of the `void *key` member to `const char *` first.

data The data associated with the key.

sdata Convenience type field.

13.2 Map initialization

During initialization, you specify the underlying storage type by selecting a given constructor function. All further operations are done through the unified HXmap API which uses a form of virtual calls internally.

Currently, there are two distinct map types in libHX. There are a handful of selectable symbols, though. Abstract types are:

HXMAPT_DEFAULT No further preferences or guarantees; selects any map type that the libHX maintainer deemed fast.

HXMAPT_ORDERED The map shall use a data structure that provides ordered traversal.

Specific types include:

HXMAPT_HASH Hash-based map – Amortized $\mathcal{O}(1)$ insertion, lookup and deletion; unordered.

HXMAPT_RBTREE Red-black binary tree – $\mathcal{O}(\log(n))$ insertion, lookup and deletion; ordered.

These can then be used with the initialization functions:

```
struct HXmap *HXmap_init(unsigned int type, unsigned int flags);
struct HXmap *HXmap_init5(unsigned int type, unsigned int flags,
    const struct HXmap_ops *ops, size_t key_size, size_t data_size);
```

Both the `*_init` and `*_init5` variant creates a new map; the latter function allows to specify the operations in detail as well as key and data size, which may become necessary when using data sets which have their own way of being managed. The `flags` parameter can contain any of the following:

HXMAP_NONE This is just a mnemonic for the value 0, indicating no flags.

HXMAP_NOREPLACE If a key already exists and another add operation is attempted, the key's associated value will be replaced by the new value. If this flag is absent, `-EEXIST` is returned. This flag is allowed to be subsequently changed by the developer if so desired, using bit logic such as `map->flags &= ~HXMAP_NOREPLACE;`.

HXMAP_SKEY Notifies the constructor that keys will be C-style strings. The flag presets the `k_compare` operation to use `strcmp`. In the flag's absence, direct value comparison will be used if the key size is specified as zero (e. g. with the `HXhashmap_init4` function call), or `memcmp` if the key size is non-zero.

HXMAP_CKEY Instructs the map to make copies of keys when they are added to the map. This is required when the buffer holding the key changes or goes out of scope. The flag presets the `k_clone` and `k_free` operations to `HX_memdup` and `free`, and as such, the `key_size` parameter must not be zero. If however, `HXMAP_SKEY` is also specified, `HX_strdup` and `free` will be used and `key_size` must be zero.

HXMAP_SDATA Notifies the constructor that data will be C-style strings. This sets up the `d_clone` and `d_free` operations.

HXMAP_CDATA Instructs the map to make copies of the data when new entries are added to the map. This is required when the buffer holding the data either goes out of scope, or you want to keep the original contents instead of just a pointer.

HXMAP_SCKEY Mnemonic for the combination of HXMAP_SKEY OR'ed with HXMAP_CKEY.

HXMAP_SCDATA Mnemonic for the combination of HXMAP_SDATA OR'ed with HXMAP_SDATA.

HXMAP_SINGULAR Specifies that the "map" is only used as a set, i. e. it does not store any values, only keys. Henceforth, the value argument to HXmap_add must always be NULL.

13.3 Flag combinations

This subsection highlights the way HXMAP_SKEY interacts with HXMAP_CKEY and the key size. The copy semantics are the same for HXMAP_SDATA and HXMAP_CDATA.

HXMAP_SKEY is unset, HXMAP_CKEY is unset

The key_size parameter at the time of map construction is ignored. The pointer value of the key parameter for the HXmap_add call is directly stored in the tree, and this is the key that uniquely identifies the map entry and which is used for comparisons. This may be used if you intend to directly map pointer values.

```
static struct something *x = ..., *y = ...;
HXmap_add(map, &x[0], "foo");
HXmap_add(map, &x[1], "bar");
```

HXMAP_SKEY is set, HXMAP_CKEY is unset

The key_size parameter at the time of map construction is ignored. The pointer value of the key parameter for the HXmap_add call is directly stored in the tree, but it is the C string *pointed to* by the key parameter that serves as the key.

HXMAP_SKEY is set, HXMAP_CKEY is set

The key_size parameter at the time of map construction is ignored. The string pointed to by the key parameter will be duplicated, and the resulting pointer will be stored in the tree. Again, it is the pointed-to string that is the key.

HXMAP_SKEY is unset, HXMAP_CKEY is set

The memory block pointed to by the key parameter will be duplicated. The key_size parameter must be non-zero for this to successfully work.

With separate ops

However, when a custom struct HXmap_ops is provided in the call to HXmap_init5, any of these semantics can be overridden. Particularly, since your own ops can practically ignore key_size, it could be set to any value.

13.4 Key-data operations

The HXMAP_SKEY/CKEY/SDATA/CDATA flags are generally sufficient to set up common maps where keys and/or data are C strings or simple binary data where memdup/memcmp is enough. Where the provided mechanisms are not cutting it, an extra HXmap_ops structure with functions specialized in handling the keys and/or data has to be used as an argument to the initialization function call.

```
struct HXmap_ops {
        int (*k_compare)(const void *, const void *, size_t);
        void *(*k_clone)(const void *, size_t);
        void (*k_free)(void *);
        void *(*d_clone)(const void *, size_t);
        void (*d_free)(void *);
        unsigned long (*k_hash)(const void *, size_t);
};
```

k_compare Function to compare two keys. The return value is the same as that of memcmp or strcmp: negative values indicate that the first key is "less than" the second, zero indicates that both keys are equal, and positive values indicate that the first key is "greater than" the second. The size argument in third position is provided so that memcmp, which wants a size parameter, can directly be used without having to write an own function.

k_clone Function that will clone (duplicate) a key. This is used for keys that will be added to the tree, and potentially also for state-keeping during traversal of the map. It is valid that this clone function simply returns the value of the pointer it was actually passed; this is used by default for maps without HXMAP_CKEY for example.

k_free Function to free a key. In most cases it defaults to free(3), but in case you are using complex structs, more cleanup may be needed.

d_clone Same as k_clone, but for data.

d_free Same as k_free, but for data.

k_hash Specifies an alternate hash function. Only to be used with hash-based maps. Hashmaps default to using the DJB2 string hash function when HXMAP_SKEY is given, or otherwise the Jenkins' lookup3 hash function.

libHX exports two hash functions that you can select for struct HXmap_ops's k_hash if the default for a given flag combination is not to your liking.

HXhash_jlookup3 Bob Jenkins's lookup3 hash.

HXhash_djb2 DJB2 string hash.

13.5 Map operations

```
int HXmap_add(struct HXmap *, const void *key, const void *value);
const struct HXmap_node *HXmap_find(const struct HXmap *, const void *key);
void *HXmap_get(const struct HXmap *, const void *key);
void *HXmap_del(struct HXmap *, const void *key);
void HXmap_free(struct HXmap *);
struct HXmap_node *HXmap_keysvalues(const struct HXmap *);
```

HXmap_add Adds a new node to the tree using the given key and data. When an element is in the map, the key may not be modified, as doing so could possibly invalidate the internal location of the element, or its ordering with respect to other elements. If you need to change the key, you will have to delete the element from the tree and re-insert it. On error, `-errno` will be returned.
When `HXMAP_SINGULAR` is in effect, `value` must be `NULL`, or `-EINVAL` is returned.

HXmap_find Finds the node for the given key. The key can be read from the node using `node->key` or `node->skey` (convenience alias for `key`, but with a type of `const char *`), and the data by using `node->data` or `node->sdata`. (see section 13.1).

HXmap_get Get is a find operation directly returning `node->data` instead of the node itself. Since `HXmap_get` may legitimately return `NULL` if `NULL` was stored in the tree as the data for a given key, only `errno` will really tell whether the node was found or not; in the latter case, `errno` is set to `ENOENT`.

HXmap_del Removes an element from the map and returns the data value that was associated with it. When an error occurred, or the element was not found, `NULL` is returned. Because `NULL` can be a valid data value, `errno` can be checked for non-zero. `errno` will be `-ENOENT` if the element was not found, or zero when everything was ok.

HXmap_free The function will delete all elements in the map and free memory it holds.

HXmap_keysvalues Returns all key-value-pairs in an array of the size as many items were in the map (`map->items`) at the time it was called. The memory must be freed using `free`(3) when it is no longer needed. The order elements in the array follows the traverser notes (see below), unless otherwise specified.

13.6 Map traversal

```
struct HXmap_trav *HXmap_travinit(const struct HXmap *);
const struct HXmap_node *HXmap_traverse(struct HXmap_trav *iterator);
void HXmap_travfree(struct HXmap_trav *iterator);
void HXmap_qfe(const struct HXmap *, bool (*fn)(const struct HXmap_node *,
     void *arg), void *arg);
```

HXmap_travinit Initializes a traverser (a. k. a. iterator) for the map, and returns a pointer to it. `NULL` will be returned in case of an error, such as memory allocation failure. Traversers are returned even if the map has zero elements.

HXmap_traverse Returns a pointer to a `struct HXmap_node` for the next element / key-value pair from the map, or `NULL` if there are no more entries.

HXmap_travfree Release the memory associated with a traverser.

HXmap_qfe The "quick foreach". Iterates over all map elements in the fastest possible manner, but has the restriction that no modifications to the map are allowed. Furthermore, a separate function to handle each visited node, is required. (Hence this is also called "closed traversal", because one cannot access the stack frame of the original function which called `HXmap_qfe`.) The user-defined function returns a bool which indicates whether traversal shall continue or not.

Flags for `HXmap_travinit`:

`HXMAP_NOFLAGS` A mnemonic for no flags, and is defined to be `0`.

`HXMAP_DTRAV` Enable support for deletion during traversal. As it can make traversal slower, it needs to be explicitly specified for cases where it is needed, to not penalize cases where it is not.

WARNING: Modifying the map while a traverser is active is implementation-specific behavior! libHX generally ensures that there will be no undefined behavior (e. g. crashes), but there is no guarantee that elements will be returned exactly once. There are fundamental cases that one should be aware of:

- An element is inserted before where the traverser is currently positioned at. The element may not be returned in subsequent calls to `HXmap_traverse` on an already-active traverser.
- Insertion or deletion may cause internal data structure to re-layout.
  - Traversers of ordered data structures may choose to rebuild their state.
  - Traversers of unordered data structures would run risk to return more than once, or not at all.

Descriptions for different map types follow.

Hashmaps On `HXmap_add`, an element may be inserted in a position that is before where the traverser is currently positioned. Such elements will not be returned in the remaining calls to `HXmap_traverse`. The insertion or deletion of an element may cause the internal data structure to re-layout itself. When this happens, the traverser will stop, so as to not return entries twice.

Binary trees Elements may be added before the traverser's position. These elements will not be returned in subsequent traversion calls. If the data structure changes as a result of an addition or deletion, the traverser will rebuild its state and continue traversal transparently. Because elements in a binary tree are ordered, that is, element positions may not change with respect to another when the tree is rebalanced, there is no risk of returning entries more than once. Nor will elements that are sorted after the current traverser's position not be returned (= they will be returned, because they cannot get reordered to before the traverser like in a hash map). The HX rbtree implementation also has proper handling for when the node which is currently visiting is deleted.

13.7 RB-tree Limitations

The implementation has a theoretical minimum on the maximum number of nodes, 2^{24} = 16,777,216. A worst-case tree with this many elements already has a height of 48 (`RBT_MAXDEP`), which is the maximum height currently supported. The larger the height is that HXrbtree is supposed to handle, the more memory (linear increase) it needs. All functions that build or keep a path reserve memory for `RBT_MAXDEP` nodes; on x86_64 this is 9 bytes per ⟨node, direction⟩ pair, amounting to 432 bytes for path tracking alone. It may not sound like a lot to many, but given that kernel people can limit their stack usage to 4096 bytes is impressive alone [3].

[3] Not always of course. Linux kernels are often configured to use an 8K stack because some components still use a lot of stack space, but even 8K is still damn good.

13.8 Examples

13.8.1 Case-insensitive ordering

The correct way:

```
static int my_strcasecmp(const void *a, const void *b, size_t z)
{
        return strcasecmp(a, b);
}

static const struct HXmap_ops icase = {
        .k_compare = my_strcasecmp,
};
HXmap_init5(HXMAPT_RBTREE, HXMAP_SKEY, &icase, 0, dsize);
```

A hackish way (which wholly depends on the C implementation and use of extra safeguards is a must):

```
static const struct HXmap_ops icase = {
         .k_compare = (void *)strcasecmp,
};
BUILD_BUG_ON(sizeof(DEMOTE_TO_PTR(strcasecmp)) > sizeof(void *));
BUILD_BUG_ON(sizeof(DEMOTE_TO_PTR(strcasecmp)) > sizeof(icase.k_compare));
HXmap_init5(HXMAPT_RBTREE, HXMAP_SKEY, &icase, 0, dsize);
```

13.8.2 Reverse sorting order

Any function that behaves like `strcmp` can be used. It merely has to return negative when $a < b$, zero on $a = b$, and positive non-zero when $a > b$.

```
static int strcmp_rev(const void *a, const void *b, size_t z)
{
        /* z is provided for cases when things are raw memory blocks. */
        return strcmp(b, a);
}

static const struct HXmap_ops rev = {
        .k_compare = strcmp_rev,
};
HXmap_init5(HXMAPT_RBTREE, HXMAP_SKEY, &rev, 0, dsize);
```

13.8.3 Keys with non-unique data

Keys can actually store non-unique data, as long as this extra fields does not actually contribute to the logical key — the parts that do uniquely identify it. In the following example, the `notes` member may be part of struct package, which is the key as far as HXmap is concerned, but still, only the name and versions are used to identify it.

```
struct package {
        char *name;
        unsigned int major_version;
        unsigned int minor_version;
```

```
        char notes[64];
};

static int package_cmp(const void *a, const void *b)
{
        const struct package *p = a, *q = b;
        int ret;
        ret = strcmp(p->name, q->name);
        if (ret != 0)
                return ret;
        ret = p->major_version - q->major_version;
        if (ret != 0)
                return ret;
        ret = p->minor_version - q->minor_version;
        if (ret != 0)
                return ret;
        return 0;
}

static const struct HXmap_ops package_ops = {
        .k_compare = package_cmp,
};

HXmap_init5(HXMAPT_RBTREE, flags, &package_ops,
            sizeof(struct package), dsize);
```

14 Doubly-linked list

HXdeque is a data structure for a doubly-linked non-circular NULL-sentineled list. Despite being named a deque, which is short for double-ended queue, and which may be implemented using an array, HXdeque is in fact using a linked list to provide its deque functionality. Furthermore, a dedicated root structure and decidated node structures with indirect data referencing are used.

14.1 Structural definition

```
#include <libHX/deque.h>

struct HXdeque {
        struct HXdeque_node *first, *last;
        unsigned int items;
        void *ptr;
};

struct HXdeque_node {
        struct HXdeque_node *next, *prev;
        struct HXdeque *parent;
        void *ptr;
};
```

The `ptr` member in `struct HXdeque` provides room for an arbitrary custom user-supplied pointer. `items` will reflect the number of elements in the list, and must not be modified. `first` and `last` provide entrypoints to the list's ends.

`ptr` within `struct HXdeque_node` is the pointer to the user's data. It may be modified and used at will by the user. See example section .

14.2 Constructor, destructors

```
struct HXdeque *HXdeque_init(void);
void HXdeque_free(struct HXdeque *dq);
void HXdeque_genocide(struct HXdeque *dq);
void HXdeque_genocide2(struct HXdeque *dq, void (*xfree)(void *));
void **HXdeque_to_vec(struct HXdeque *dq, unsigned int *num);
```

To allocate a new empty list, use `HXdeque_init`. `HXdeque_free` will free the list (including all nodes owned by the list), but not the data pointers.

`HXdeque_genocide` is a variant that will not only destroy the list, but also calls a freeing function `free()` on all stored data pointers. This puts a number of restrictions on the characteristics of the list: all data pointers must have been obtained with `malloc`, `calloc` or `realloc` before, and no data pointer must exist twice in the list. The function is more efficient than an open-coded loop over all nodes calling `HXdeque_del`.

A generic variant is available with `HXdeque_genocide2`, which takes a pointer to an appropriate freeing function. `HXdeque_genocide` is thus equivalent to `HXdeque_genocide2(dq, free)`.

To convert a linked list to a `NULL`-terminated array, `HXdeque_to_vec` can be used. If `num` is not `NULL`, the number of elements excluding the `NULL` sentinel, is stored in `*num`.

14.3 Addition and removal

```
struct HXdeque_node *HXdeque_push(struct HXdeque *dq, void *ptr);
struct HXdeque_node *HXdeque_unshift(struct HXdeque *dq, void *ptr);
void *HXdeque_pop(struct HXdeque *dq);
void *HXdeque_shift(struct HXdeque *dq);
struct HXdeque *HXdeque_move(struct HXdeque_node *target,
                             struct HXdeque_node *node);
void *HXdeque_del(struct HXdeque_node *node);
```

`HXdeque_push` and `HXdeque_unshift` add the data item in a new node at the end ("push") or as the new first element ("unshift" as Perl calls it), respectively. The functions will return the new node on success, or `NULL` on failure and `errno` will be set. The node is owned by the list.

`HXdeque_pop` and `HXdeque_shift` remove the last ("pop") or first ("shift") node, respectively, and return the data pointer that was stored in the data.

`HXdeque_move` will unlink a node from its list, and reinsert it after the given target node, which may be in a different list.

Deleting a node is accomplished by calling `HXdeque_del` on it. The data pointer stored in the node is not freed, but returned.

14.4 Iteration

Iterating over a HXdeque linked list is done manually and without additional overhead of function calls:

```
const struct HXdeque_node *node;
for (node = dq->first; node != NULL; node = node->next)
        do_something(node->ptr);
```

14.5 Searching

```
struct HXdeque_node *HXdeque_find(struct HXdeque *dq, const void *ptr);
void *HXdeque_get(struct HXdeque *dq, void *ptr);
```

`HXdeque_find` searches for the node which contains `ptr`, and does so by beginning at the start of the list. If no node is found, `NULL` is returned. If a pointer is more than once in the list, any node may be returned.

`HXdeque_get` will further return the data pointer stored in the node — however, since that is just what the `ptr` argument is, the function practically only checks for existence of `ptr` in the list.

14.6 Examples

In this example, all usernames are obtained from NSS, and put into a list. `HX_strdup` is used, because `getpwent` will overwrite the buffer it uses to store its results. The list is then converted to an array, and the list is freed (because it is not need it anymore). `HXdeque_genocide` must not be used here, because it would free all the data pointers (strings here) that were just inserted into the list. Finally, the list is sorted using the well-known `qsort` function. Because `strcmp` takes two `const char *` arguments, but `qsort` mandates a function taking two `const void *`, a cast can be used to silence the compiler. This only works because we know that the array consists of a bunch of `char *` pointers, so `strcmp` will work.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libHX/defs.h>
#include <libHX/deque.h>
#include <libHX/string.h>
#include <pwd.h>

int main(void)
{
        struct HXdeque *dq = HXdeque_init();
        struct passwd *pw;
        unsigned int elem;
        char **users;

        setpwent();
        while ((pw = getpwent()) != NULL)
                HXdeque_push(dq, HX_strdup(pw->pw_name));
        endpwent();

        users = reinterpret_cast(char **, HXdeque_to_vec(dq, &elem));
        HXdeque_free(dq);

        qsort(users, elem, sizeof(*users), static_cast(void *, strcmp));
        return 0;
}
```

Figure 1: Example use of HXdeque to store and sort a list

15 Inline doubly-linked list

Classical linked-list implementations, such as HXdeque, either store the actual data within a node, or indirectly through a pointer, but the "inline doubly-linked list" instead does it reverse and has the list head within the data structure.

```
struct package_desc {
        char *package_name;
        int version;
};
struct classic_direct_node {
        struct classic_direct_node *next, *prev;
        struct package_desc direct_data;
};
struct classic_indirect_node {
        struct classic_indirect_node *next, *prev;
        void *indirect_data;
};
```

Figure 2: Classic linked-list implementations with direct/indirect data blocks.

```
struct package_desc {
        struct HXlist_head list;
        char *package_name;
        int version;
};
```

Figure 3: List head (next,prev pointers) inlined into the data block

At first glance, an inline list does not look much different from `struct classic_direct_data`, it is mostly a viewpoint decision which struct is in the foreground.

15.1 Synopsis

```
#include <libHX/list.h>

struct HXlist_head {
        /* All fields considered private */
};

HXLIST_HEAD_INIT(name);
HXLIST_HEAD(name);
void HXlist_init(struct HXlist_head *list);
void HXlist_add(struct HXlist_head *list, struct HXlist_head *elem);
void HXlist_add_tail(struct HXlist_head *list, struct HXlist_head *elem);
void HXlist_del(struct HXlist_head *element);
bool HXlist_empty(const struct HXlist_head *list);
```

HXLIST_HEAD_INIT This macro expands to the static initializer for a list head.

HXLIST_HEAD This macro expands to the definition of a list head (i. e. `struct HXlist_head name = HXLIST_HEAD_INIT;`)

HXlist_init Initializes the list head. This function is generally used when the list head is on the heap where the static initializer cannot be used.

HXlist_add Adds `elem` to the front of the list.

HXlist_add_tail Adds `elem` to the end of the list.

HXlist_del Deletes the given element from the list.

HXlist_empty Tests whether the list is empty. Note: For clists, you could also use `clist->items == 0`.

15.2 Traversal

Traversal is implemented using macros that expand to for() statements which can syntactically be used like them, i. e. curly braces may be omitted if only a single statement is in the body of the loop.

The `head` parameter specifies the list head (`struct HXlist_head`), `pos` specifies an iterator, also of type `struct HXlist_head`. Lists can either be traversed in forward direction, or, using the `_rev` variants, in reverse direction. The `_safe` variants use a temporary `n` to hold the next object in the list, which is needed when pos itself is going to be inaccessible at the end of the block, through, for example, freeing its encompassing object.

```
HXlist_for_each(pos, head)
HXlist_for_each_rev(pos, head)
HXlist_for_each_safe(pos, n, head)
HXlist_for_each_rev_safe(pos, n, head)
```

HXlist_for_each Forward iteration over the list heads.

HXlist_for_each_rev Reverse iteration over the list heads.

HXlist_for_each_safe Forward iteration over the list heads that is safe against freeing `pos`.

HXlist_for_each_rev_safe Reverse iteration over the list heads that is safe against freeing `pos`.

The `_entry` variants use an iterator `pos` of the type of the encompassing object (e. g. `struct item` in below's example), so that the manual `HXlist_entry` invocation is not needed. `member` is the name of the list structure embedded into the item.

```
HXlist_for_each_entry(pos, head, member)
HXlist_for_each_entry_rev(pos, head, member)
HXlist_for_each_entry_safe(pos, n, head, member)
```

HXlist_for_each_entry Forward iteration over the list elements.

HXlist_for_each_entry_rev Reverse iteration over the list elements.

HXlist_for_each_entry_safe Forward iteration over the list elements that is safe against freeing `pos`.

15.3 Examples

```
struct item {
        struct HXlist_head anchor;
        char name[32];
};

struct HXlist_head *e;
struct item *i, *j;
HXLIST_HEAD(list);

i = malloc(sizeof(*i));
HXlist_init(&e->anchor);
strcpy(i->name, "foo");
HXlist_add_tail(&list, &i->anchor);

i = malloc(sizeof(*i));
HXlist_init(&e->anchor);
strcpy(i->name, "bar");
HXlist_add_tail(&list, &i->anchor);

HXlist_for_each(e, &list) {
        i = HXlist_entry(e, typeof(*i), anchor);
        printf("e=%p i=%p name=%s\n", e, i, i->name);
}

HXlist_for_each_entry(i, &list, anchor)
        printf("i=%p name=%s\n", i, i->name);

HXlist_for_each_entry_rev(i, &list, anchor)
        printf("i=%p name=%s\n", i, i->name);

HXlist_for_each_entry_safe(i, j, &list, anchor) {
        printf("i=%p name=%s\n", i, i->name);
        free(i);
}
```

15.4 When to use HXdeque/HXlist

The choice whether to use HXdeque or HXlist/HXclist depends on whether one wants the list head handling on the developer or on the library. Especially for "atomic" and "small" data, it might be easier to just let HXdeque do the management. Compare the following two code examples to store strings:

```
int main(int argc, const char **argv)
{
        struct HXdeque *dq = HXdeque_init();
        while (--argc)
                 HXdeque_push(dq, ++argv);
        return 0;
}
```

Figure 4: Storing strings in a HXdeque

```
struct element {
        struct HXlist_head list;
        char *data;
};

int main(int main, const char **argv)
{
        HXLIST_HEAD(lh);
        while (--argc) {
                struct element *e = malloc(sizeof(*e));
                e->data = *++argv;
                HXlist_init(&e->list);
                HXlist_add_tail(&e->list);
        }
        return 0;
}
```

Figure 5: Storing strings in a HXlist

These examples assume that `argv` is persistent, which, for the sample, is true.

With HXlist, one needs to have a struct with a HXlist_head in it, and if one does not already have such a struct —e. g. by means of wanting to store more than just one value — one will need to create it first, as shown, and this may lead to an expansion of code.

This however does not mean that HXlist is the better solution over HXdeque for data already available in a struct. As each struct has a list_head that is unique to the node, it is not possible to share this data. Trying to add a HXlist_head to another list is not going to end well, while HXdeque has no problem with this as list heads are detached from the actual data in HXdeque.

```
struct point p = {15, 30};
HXdeque_push(dq, &p);
HXdeque_push(dq, &p);
```

Figure 6: Data can be added multiple times in a HXdeque without ill effects

To support this, an extra allocation is needed on the other hand. In a HXlist, to store n elements of compound data (e. g. `struct point`), n allocations are needed, assuming the list head is a stack object, and the points are not. HXdeque will need at least $2n + 1$ allocations, n for the nodes, n for the points and another for the head.

16 Counted inline doubly-linked list

clist is the inline doubly-linked list from chapter 15, extended by a counter to retrieve the number of elements in the list in $\mathcal{O}(1)$ time. This is also why all operations always require the list head. For traversal of clists, use the corresponding HXlist macros.

16.1 Synopsis

```
#include <libHX/list.h>

struct HXclist_head {
        /* public readonly: */
        unsigned int items;
        /* Undocumented fields are considered "private" */
};

HXCLIST_HEAD_INIT(name);
HXCLIST_HEAD(name);
void HXclist_init(struct HXclist_head *head);
void HXclist_unshift(struct HXclist_head *head, struct HXlist_head *new_node);
void HXclist_push(struct HXclist_head *head, struct HXlist_head *new_node);
type HXclist_pop(struct HXclist_head *head, type, member);
type HXclist_shift(struct HXclist_head *head, type, member);
void HXclist_del(struct HXclist_head *head, struct HXlist_chead *node);
```

HXCLIST_HEAD_INIT Macro that expands to the static initializer for a clist.

HXCLIST_HEAD Macro that expands to the definition of a clist head, with initialization.

HXclist_init Initializes a clist. This function is generally used when the head has been allocated from the heap.

HXclist_unshift Adds the node to the front of the list.

HXclist_push Adds the node to the end of the list.

HXclist_pop Removes the last node in the list and returns it.

HXclist_shift Removes the first node in the list and returns it.

HXclist_del Deletes the node from the list.

The list count in the clist head is updated whenever a modification is done on the clist through these functions.

Part III
Strings and memory

17 String operations

Some string functions are merely present in libHX because they are otherwise unportable; some are only in the C libraries of the BSDs, some only in GNU libc.

17.1 Locating chars

```
#include <libHX/string.h>

void *HX_memmem(const void *haystack, size_t hsize,
                const void *needle, size_t nsize);
char *HX_strbchr(const char *start, const char *now, char delimiter);
char *HX_strchr2(const char *s, const char *accept);
size_t HX_strrcspn(const char *s, const char *reject);
```

HX_memmem Analogous to strstr(3), memmem tries to locate the memory block pointed to by needle (which is of length nsize) in the block pointed to by haystack (which is of size hsize). It returns a pointer to the first occurrence in haystack, or NULL when it was not found.

HX_strbchr Searches the character specified by delimiter in the range from now to start. It works like strrchr(3), but begins at now rather than the end of the string.

HX_strchr2 This function searches the string s for any set of bytes that are not specified in the second argument, n. In this regard, the function is the opposite to strpbrk(3).

HX_strrcspn Works like strcspn(3), but processes the string from end to start.

17.2 Extraction

```
#include <libHX/string.h>

char *HX_basename(const char *s);
char *HX_basename_exact(const char *s);
char *HX_dirname(const char *s);
char *HX_strmid(const char *s, long offset, long length);
```

HX_basename Returns a pointer to the basename portion of the supplied path s. The result of this function is never NULL, and must never be freed either. Trailing slashes are not stripped, to avoid having to do an allocation. In other words, basename("/mnt/") will return "mnt/". If you need to have the slashes stripped, use HX_basename_exact. A possible use for this function is, for example, to derive a logging prefix from argv[0].

```
int main(int argc, const char **argv)
{
        if (foo())
                fprintf(stderr, "%s: Special condition occurred.\n",
```

```
                        HX_basename(argv[0]));
        return 0;
}
```

HX_basename_exact The accurate and safe version of `HX_basename` that deals with trailing slashes correctly and produces the same result as `dirname`(3). It returns a pointer to a newly-allocated string that must be freed when done using. `NULL` may be returned in case of an allocation error.

HX_dirname Returns a pointer to a new string that contains the directory name portion (everything except basename). When done using the string, it must be freed to avoid memory leaks.

HX_strmid Extract a substring of `length` characters from `s`, beginning at `offset`. If `offset` is negative, counting beings from the end of the string; -1 is the last character (not the `'\0'` byte). If `length` is negative, it will leave out that many characters off the end. The function returns a pointer to a new string, and the user has to free it.

17.3 In-place transformations

```
#include <libHX/string.h>

char *HX_chomp(char *s);
size_t HX_strltrim(char *s);
char *HX_stpltrim(const char *s);
char *HX_strlower(char *s);
char *HX_strrev(char *s);
size_t HX_strrtrim(char *s);
char *HX_strupper(char *s);
```

HX_chomp Removes the characters `'\r'` and `'\n'` from the right edge of the string. Returns the original argument.

HX_strltrim Trim all whitespace (characters on which `isspace`(3) return true) on the left edge of the string. Returns the number of characters that were stripped.

HX_stpltrim Returns a pointer to the first non-whitespace character in `s`.

HX_strlower Transforms all characters in the string `s` into lowercase using `tolower`(3). Returns the original argument.

HX_strrev Reverse the string in-place. Returns the original argument.

HX_strrtrim Trim all whitespace on the right edge of the string. Returns the number of characters that were stripped.

HX_strupper Transforms all characters in the string `s` into uppercase using `toupper`(3). Returns the original argument.

17.4 Out-of-place quoting transforms

```
#include <libHX/string.h>

char *HX_strquote(const char *s, unsigned int type, char **free_me);
```

HX_strquote will escape metacharacters in a string according to type, and returns the escaped result.

Possible values for type:

HXQUOTE_SQUOTE Escape all single quotes and backslashes in a string with a backslash. ("Ol' \Backslash" → "Ol\' \\Backslash")

HXQUOTE_DQUOTE Escape all double quotes and backslahes in a string with the backslash method. ("Ol" \Backslash" → "Ol\" \\Backslash")

HXQUOTE_HTML Escape '<', '>', '&' and '"' by their respective HTML entities <, >, & and ".

HXQUOTE_LDAPFLT Escape the string using backslash-plus-hexcode notation as described in RFC 4515[4], to make it suitable for use in an LDAP search filter.

HXQUOTE_LDAPRDN Escape the string using backslash-plus-hexcode notation as described in RFC 4514[5], to make it suitable for use in an LDAP Relative Distinguished Name.

HXQUOTE_BASE64 Transform the string to BASE64, as described in RFC 4648[6].

HXQUOTE_URIENC Escape the string so that it becomes a valid part for an URI.

HXQUOTE_SQLSQUOTE Escape all single quotes in the string by double single-quotes, as required for using it in a single-quoted SQL string. No surrounding quotes will be generated to facilitate concatenating of HX_strquote results.

HXQUOTE_SQLBQUOTE Escape all backticks in the string by double backticks, as required for using it in a backtick-quoted SQL string (used for table names and columns). No surrounding ticks will be generated to facilitate concatenation.

Specifying an unrecognized type will result in NULL being returned and errno be set to EINVAL.

If free_me is NULL, the function will always allocate memory, even if the string needs no quoting. The program then has to free the result:

```
char *s = HX_strquote("<head>", HXQUOTE_HTML, NULL);
printf("%s\n", s);
free(s);
```

If free_me is not NULL however, the function will put the pointer to the memory area into *free_me, if the string needed quoting. The program then has to free that after it is done with the quoted result:

```
char *tmp = NULL;
char *s = HX_strquote("head", HXQUOTE_HTML, &tmp);
printf("%s\n", s);
free(tmp);
```

[4] http://tools.ietf.org/html/rfc4515
[5] http://tools.ietf.org/html/rfc4514
[6] http://tools.ietf.org/html/rfc4648

tmp could be NULL, and since free(NULL) is not an error, this is perfectly valid. Furthermore, if *free_me is not NULL by the time HX_strquote is called, the function will free it. This makes it possible to call HX_strquote in succession without frees in between:

```
char *tmp = NULL;
printf("%s\n", HX_strquote("<html>", HXQUOTE_HTML, &tmp));
printf("%s\n", HX_strquote("<head>", HXQUOTE_HTML, &tmp));
free(tmp);
```

17.5 Tokenizing

```
#include <libHX/string.h>

char **HX_split(const char *s, const char *delimiters,
                size_t *fields, int max);
char **HX_split4(char *s, const char *delimiters, int *fields, int max);
int HX_split5(char *s, const char *delimiters, int max, char **stack);
char *HX_strsep(char **sp, const char *delimiters);
char *HX_strsep2(char **sp, const char *dstr);
```

HX_split Split the string s on any characters from the "delimiters" string. Both the substrings and the array holding the pointers to these substrings will be allocated as required; the original string is not modified. If max is larger than zero, produces no more than max fields. If fields is not NULL, the number of elements produced will be stored in *fields. The result is a NULL-terminated array of char *, and the user needs to free it when done with it, using HX_zvecfree or equivalent. An empty string (zero-length string) for s yields a single field.

HX_split4 Split the string s in-place on any characters from the "delimiters" string. The array that will be holding the pointers to the substrings will be allocated and needs to be freed by the user, using free(3). The fields and max arguments work as with HX_split.

HX_split5 Split the string s in-place on any characters from the "delimiters" string. The array for the substring pointers must be provided by the user through the stack argument. max must be the number of elements in the array or less. The array will not be NULL-terminated[7]. The number of fields produced is returned.

HX_strsep Extract tokens from a string.

This implementation of strsep has been added since the function is non-standard (according to the manpage, conforms to BSD4.4 only) and may not be available on every operating system.

This function extracts tokens, separated by one of the characters in delimiters. The string is modified in-place and thus must be writable. The delimiters in the string are then overwritten with '\0', *sp is advanced to the character after the delimiter, and the original pointer is returned. After the final token, strsep will return NULL.

HX_strsep2 Like HX_strsep, but dstr is not an array of delimiting characters, but an entire substring that acts as a delimiter.

[7] An implementation may however decide to put NULL in the unassigned fields, but this is implementation and situation-specific. Do not rely on it.

17.6 Size-bounded string ops

```
#include <libHX/string.h>

char *HX_strlcat(char *dest, const char *src, size_t length);
char *HX_strlcpy(char *dest, const char *src, size_t length);
char *HX_strlncat(char *dest, const char *src, size_t dlen, size_t slen);
size_t HX_strnlen(const char *src, size_t max);
```

HX_strlcat and HX_strlcpy provide implementations of the BSD-originating strlcat(3) and strlcpy(3). strlcat and strlcpy are less error-prone variants for strncat and strncpy as they always take the length of the entire buffer specified by dest, instead of just the length that is to be written. The functions guarantee that the buffer is '\0'-terminated.

HX_strnlen will return the length of the input string or the upper bound given by max, whichever is less. It will not attempt to access more than this many bytes in the input buffer.

17.7 Allocation-related

```
#include <libHX/string.h>

void *HX_memdup(const void *ptr, size_t length);
char *HX_strdup(const char *str);
char *HX_strndup(const char *str, size_t max);
char *HX_strclone(char **pa, const char *pb);

#ifdef __cplusplus
template<typename type> type HX_memdup(const void *ptr, size_t length);
#endif
```

HX_memdup Duplicates length bytes from the memory area pointed to by ptr and returns a pointer to the new memory block. ptr may not be NULL.

HX_strdup Duplicates the string. The function is equivalent to strdup, but the latter may not be available on all platforms. str may be NULL, in which case NULL is also returned.

HX_strndup Duplicates the input string, but copies at most max characters. (The resulting string will be NUL-terminated of course.) str may not be NULL.

HX_strclone Copies the string pointed to by pb into *pa. If *pa was not NULL by the time HX_strclone was called, the string is freed before a new one is allocated. The function returns NULL and sets errno to EINVAL if pb is NULL (this way it can be freed), or, if malloc fails, returns NULL and leaves errno at what malloc set it to.
The use of this function is deprecated, albeit no replacement is proposed.

17.8 Examples

17.8.1 Using HX_split5

HX_split5, where the "5" should be interpreted (with a bit of imagination and the knowledge of leetspeak) as an "S" for stack, as HX_split5 is often used only with on-stack variables and where the field count of interest is fixed, as the example for parsing /etc/passwd shows:

```
#include <stdio.h>
#include <libHX/string.h>

char *field[8];
hxmc_t *line = NULL;

while (HX_getl(&line, fp) != NULL) {
        if (HX_split5(line, ":", ARRAY_SIZE(field), field) < 7) {
                fprintf(stderr, "That does not look like a valid line.\n");
                continue;
        }
        printf("Username: %s\n", field[0]);
}
```

17.8.2 Using HX_split4

Where the number of fields is not previously known and/or estimatable, but the string can be modified in place, one uses `HX_split4` as follows:

```
#include <errno.h>
#include <stdio.h>
#include <libHX/string.h>

while (HX_getl(&line, fp) != NULL) {
        char **field = HX_split4(line, ":", NULL, 0);
        if (field == NULL) {
                fprintf(stderr, "Badness! %s\n", strerror(errno));
                break;
        }
        printf("Username: %s\n", field[0]);
        free(field);
}
```

17.8.3 Using HX_split

Where the string is not modifiable in-place, one has to resort to using the full-fledged `HX_split` that allocates space for each substring.

```
#include <errno.h>
#include <stdio.h>
#include <libHX/string.h>

while (HX_getl(&line, fp) != NULL) {
        char **field = HX_split(line, ":", NULL, 0);
        if (field == NULL) {
                fprintf(stderr, "Badness. %s\n", strerror(errno));
                break;
        }
        printf("Username: %s\n", field[0]);
        /* Suppose "callme" needs the original string */
        callme(line);
```

```
        HX_zvecfree(field);
}
```

17.8.4 Using HX_strsep

`HX_strsep` provides for thread- and reentrant-safe tokenizing a string where strtok from the C standard would otherwise fail.

```
#include <stdio.h>
#include <libHX/string.h>

char line[] = "root:x:0:0:root:/root:/bin/bash";
char *wp, *p;

wp = line;
while ((p = HX_strsep(&wp, ":")) != NULL)
        printf("%s\n", p)
```

18 Memory containers

The HXmc series of functions provide scripting-like semantics for strings, especially automatically resizing the buffer on demand. They can also be used to store a binary block of data together with its length. (Hence the name: mc = memory container.)

The benefit of using the HXmc functions is that one does not have to meticulously watch buffer and string sizes anymore.

```
/* Step 1 */
    char buf[whatever was believed to be long enough] = "helloworld";
    if (strlen(buf) + strlen(".txt") < sizeof(buf))
            strcat(s, ".txt");

/* Step 2 */

    char buf[long_enough] = "helloworld";
    strlcat(s, ".txt", sizeof(buf));

/* Step 3 */

    hxmc_t *buf = HXmc_strinit("helloworld");
    HXmc_strcat(&s, ".txt");
```

Figure 7: Improvement of string safety over time

This makes it quite similar to the string operations (and append seems to be the most commonly used one to me) supported in scripting languages that also do without a size argument. The essential part of such memory containers is that their internal (hidden) metadata structure contains the length of the memory block in the container. For binary data this may be the norm, but for C-style strings, the stored and auto-updated length field serves as an accelerator cache. For more details, see `HXmc_length`.

Of course, the automatic management of memory comes with a bit of overhead as the string expands beyond its preallocated region. Such may be mitigated by doing explicit (re)sizing.

18.1 Structural overview

HXmc functions do not actually return a pointer to the memory container (e. g. `struct`) itself, but a pointer to the data block. Conversely, input parameters to HXmc functions will be the data block pointer. It is of type `hxmc_t *`, which is typedef'ed to `char *` and inherits all properties and privileges of `char *`. Pointer arithmetic is thus supported. It also means you can just pass it to functions that take a `char *` without having to do a member access like `s.c_str`. The drawback is that many functions operating on the memory container need a `hxmc_t **` (a level-two indirection), because not only does the memory block move, but also the memory container itself. This is due to the implementation of the container metadata which immediately and always precedes the writable memory block.

HXmc ensures that the data block is terminated by a NUL (`'\0'`) byte (unless you trash it), so you do not have to, and of course, to be on the safe side. But, the automatic NUL byte is not part of the region allocated by the user. That is, when one uses the classic approach with `malloc(4096)`, the user will have control of 4096 bytes and has to stuff the NUL byte in there somehow on his own; for strings this means the maximum string length is 4095. Requesting

space for a 4096-byte sized HXmc container gives you the possibility to use all 4096 bytes for the string, because HXmc provides a NUL byte.

By the way, `hxmc_t` is the *only* typedef in this entire library, to distinguish it from regular `char *` that does not have a backing memory cointainer.

18.2 Constructors, destructors

```
#include <libHX/string.h>

hxmc_t *HXmc_strinit(const char *s);
hxmc_t *HXmc_meminit(const void *ptr, size_t size);
```

HXmc_strinit Creates a new hxmc_t object from the supplied string and returns it.

HXmc_meminit Creates a new hxmc_t object from the supplied memory buffer of the given size and returns it. `HXmc_meminit(NULL, len)` may be used to obtain an empty container with a preallocated region of `len` bytes (zero is accepted for `len`).

```
void HXmc_free(hxmc_t *s);
void HXmc_zvecfree(hxmc_t **s);
```

HXmc_free Frees the hxmc object.

HXmc_zvecfree Frees all hxmc objects in the NULL-terminated array, and finally frees the array itself, similar to `HX_zvecfree`.

18.3 Data manipulation

18.3.1 Binary-based

```
hxmc_t *HXmc_trunc(hxmc_t **mc, size_t len);
hxmc_t *HXmc_setlen(hxmc_t **mc, size_t len);
hxmc_t *HXmc_memcpy(hxmc_t **mc, const void *ptr, size_t len);
hxmc_t *HXmc_memcat(hxmc_t **mc, const void *ptr, size_t len);
hxmc_t *HXmc_mempcat(hxmc_t **mc, const void *ptr, size_t len);
hxmc_t *HXmc_memins(hxmc_t **mc, size_t pos, const void *ptr, size_t len);
hxmc_t *HXmc_memdel(hxmc_t **mc, size_t pos, size_t len);
```

When `ptr` is `NULL`, each call behaves as if `len` would be zero. Specifically, no undefined behavior will result of the use of `NULL`.

HXmc_trunc Truncates the container's data to `len` size. If `len` is greater than the current data size of the container, the length is in fact *not* updated, but a reallocation may be triggered, which can be used to do explicit allocation.

HXmc_setlen Set the data length, doing a reallocation of the memory container if needed. The newly available bytes are uninitialized. Make use of this function when letting 3rd party functions write to the buffer, but it should not be used with `HXmc_str*`(),

HXmc_memcpy Truncates the container's data and copies `len` bytes from the memory area pointed to by `ptr` to the container.

HXmc_memcat Concatenates (appends) `len` bytes from the memory area pointed to by `ptr` to the container's data.

HXmc_mempcat Prepends `len` bytes from the memory area pointed to by `ptr` to the container's data.

HXmc_memins Prepends `len` bytes from the memory area pointed to by `ptr` to the `pos`'th byte of the container's data.

HXmc_memdel Deletes `len` bytes from the container beginning at position `pos`.

In case of a memory allocation failure, the `HXmc_*` functions will return `NULL`.

18.3.2 String-based

The string-based functions correspond to their binary-based equivalents with a `len` argument of `strlen(s)`.

```
hxmc_t *HXmc_strcpy(hxmc_t **mc, const char *s);
hxmc_t *HXmc_strcat(hxmc_t **mc, const char *s);
hxmc_t *HXmc_strpcat(hxmc_t **mc, const char *s);
hxmc_t *HXmc_strins(hxmc_t **mc, size_t pos, const char *s);
```

HXmc_strcpy Copies the string pointed to by `s` into the memory container given by `mc`. If `mc` is `NULL`, the memory container will be deallocated, that is, `*mc` becomes `NULL`.

18.3.3 From auxiliary sources

```
hxmc_t *HX_getl(hxmc_t **mc, FILE *fp);
```

HX_getl Read the next line from `fp` and store the result in the container. Returns `NULL` on error, or when end of file occurs while no characters have been read.

18.4 Container properties

```
size_t HXmc_length(const hxmc_t **mc);
```

HXmc_length Returns the length of the memory container. This is not always equal to the actual string length. For example, if `HX_chomp` was used on an MC-backed string, `strlen` will return less than `HXmc_length` if newline control characters (`'\r'` and `'\n'`) were removed.

19 Format templates

HXfmt is a small template system for by-name variable expansion. It can be used to substitute placeholders in format strings supplied by the user by appropriate expanded values defined by the program. Such can be used to allow for flexible configuration files that define key-value mappings such as

```
detect_peer = ping6 -c1 %(ADDR)
#detect_peer = nmap -sP %(ADDR) | grep -Eq "appears to be up"
```

Consider for example a monitoring daemon that allows the administrator to specify a program of his choice with which to detect whether a peer is alive or not. The user can choose any program that is desired, but evidently needs to pass the address to be tested to the program. This is where the daemon will do a substitution of the string "`ping -c1 %(ADDR)`" it read from the config file, and put the actual address in it before finally executing the command.

```
printf("%s has %u files\n", user, num);
printf("%2$u files belong to %1$s\n", num, user);
```

"`%s`" (or "`%1$s`" here) specifies how large "user" is — `sizeof(const char *)` in this case. If that is missing, there is no way to know the offset of "`num`" relative to "`user`", making varargs retrieval impossible.

Figure 8: `printf` positional parameters

`printf`, at least from GNU libc, has something vaguely similar: positional parameters. They have inherent drawbacks, though. One is of course the question of portability, but there is a bigger issue. All parameters must be specified, otherwise there is no way to determine the location of all following objects following the missing one on the stack in a varargs-function like `printf`., which makes it unsuitable to be used with templates where omitting some placeholders is allowed.

19.1 Initialization, use and deallocation

```
#include <libHX/option.h>

struct HXformat_map *HXformat_init(void);
void HXformat_free(struct HXformat_map *table);
int HXformat_add(struct HXformat_map *table, const char *key,
                 const void *ptr, unsigned int ptr_type);
```

`HXformat_init` will allocate and set up a simple string-to-string map that is used for the underlying storage, and returns it.

To release the substitution table and memory associated with it, call `HXformat_free`.

`HXformat_add` is used to add substitution entries. One can also specify other types such as numeral types. `ptr_type` describes the type behind `ptr` and are constants from `option.h` (cf. section 24.2) — not all constants can be used, though, and their meaning also differs from what `HX_getopt` or `HX_shconfig` use them for — the two could be seen as "read" operations, while HXformat is a write operation.

19.1.1 Immediate types

"Immediate types" are resolved when `HXformat_add` is called, that is, they are copied and inserted into the tree, and are subsequently independent from any changes to variables in the program. Because the HXopt-originating type name, that is, `HXTYPE_*`, is also used for deferred types, the constant `HXFORMAT_IMMED` needs to be specified on some types to denote an immediate value.

- `HXTYPE_STRING` — `ptr` is a `const char *`.
- `HXTYPE_{U,}{CHAR,SHORT,INT,LONG,LLONG} | HXFORMAT_IMMED` — mapping to the standard types

19.1.2 Deferred types

"Deferred types" are resolved on every invocation of a formatter function (`HXformat_*printf`). The expansions may be changed by modifying the underlying variable pointed to, but the pointer must remain valid and its pointee not go out of scope. Figure 9 shows the difference in a code sample.

- `HXTYPE_STRP` — `ptr` is a `const char *const *`; the pointer resolution is deferred until the formatter is called with one of the `HXformat_*printf` functions. Deferred in the sense it is always resolved anew.
- `HXTYPE_BOOL` — `ptr` is a `const int *`.
- `HXTYPE_{U,}{CHAR,SHORT,INT,LONG,LLONG}` — mapping to the standard types with one indirection (e. g. `int *`)
- `HXTYPE_{FLOAT,DOUBLE}` — mapping to the two floating-point types with one indirection (e. g. `double *`)

19.2 Invoking the formatter

```
int HXformat_aprintf(struct HXformat_map *table, hxmc_t **dest, const char *templat
int HXformat_sprintf(struct HXformat_map *table, char *dest, size_t size, const ch
int HXformat_fprintf(struct HXformat_map *table, FILE *filp, const char *template)
```

`HXformat_aprintf` Substitute placeholders in `template` using the given table. This will produce a string in a HX memory container (`hxmc_t`), and the pointer is put into `*dest`. The caller will be responsible for freeing it later when it is done using the result.

`HXformat_sprintf` Do substitution and store the expanded result in the buffer `dest` which is of size `size`.

`HXformat_fprintf` Do substituion and directly output the expansion to the given stdio stream.

On success, the length of the expanded string is returned, excluding the trailing `'\0'`. While `HXformat_sprintf` will not write more than `size` bytes (including the `'\0'`), the length it would have taken is returned, similar to what `sprintf` does. On error, negative errno is returned.

The HXformat function family recognizes make-style like functions and recursive expansion, described below.

19.3 Functions

To expand a variable, one uses a syntax like "`%(NAME)`" in the format string. Recursive expansion like "`%(%(USER))`" is supported; assuming `%(USER)` would expand to "linux", HXformat would try to resolve "`%(linux)`" next. Besides these variable substitutions, HXformat also provides function calls whose syntax is "`%(nameOfFunction parameters[...])`". Parameters can be any text, including variables. Paramters are separated from another by a delimiter specific to each function. See this list for details:

- `%(env` *variable*`)`
  The `env` function expands to the string that is stored in the environmental variable by the given name.

- `%(exec` *command* [*args...*]`)`
  The `exec` function expands to the standard output of the command. The command is directly run without shell invocation, so no special character expansion (wildcards, etc.) takes place. stdin is set to `/dev/null`. The parameter delimiter is the space character. To be able to use this function — as it is relevant to security — the fmt table needs to have a key called "`/libhx/exec`". See example 10 for details.

- `%(if` *condition*`,`[*then*][`,`[*else*]]`)`
  If the condition parameter expands to a string of non-zero length, the function expands to the "then" block, otherwise the "else" block. The delimiter used is a comma.

- `%(lower` *text*`)`, `%(upper` *text*`)`
  Lowercases or uppercases the supplied argument. As these functions are meant to take only one argument, there is no delimiter defined that would need escaping if multiple arguments were supposed to be passed. `%(lower a,b)` is equivalent to `%(lower "a,b")`.

- `%(shell` *command* [*args...*]`)`
  Similar to `%(exec)`, but invokes the shell inbetween (i. e. '`sh -c '`*command...*`'`') such that special characters, redirection, and so on can be used.

- `%(substr` *text*`,`*offset*[`,`*length*]`)`
  Extracts a substring out of the given text, starting at *offset* and running for the given length. If no length is given, will extract until the end of the string. If *offset* is negative, it specifies the offset from the end of the string. If *length* is negative, that many characters are left off the end.

- `%(snl` *text*`)`
  Strips trailing newlines from text and replaces any other newline by a space. What happens implicity in Makefiles' `$(shell ...)` statements usually is explicitly separate in libHX.

```
struct HXformat_map *table = HXformat_init();
HXformat_add(table, "/libhx/exec", NULL, HXTYPE_IMMED);
HXformat_aprintf(table, &result, "%(exec uname -s)");
```

Figure 10: Using the `%(exec)` function

19.4 Examples

```
const char *b = "Hello World";
char c[] = "Hello World";
struct HXformat_map *table = HXformat_init();
HXformat_add(table, "%(GREETING1)", b, HXTYPE_STRING);
HXformat_add(table, "%(GREETING2)", &c, HXTYPE_STRP);
b = NULL;
snprintf(c, sizeof(c), "Hello Home");
HXformat_aprintf(...);
```

Upon calling `HXformat_*printf`, `%(GREETING1)` will expand to "Hello World" whereas `%(GREETING2)` will expand to "Hello Home".

Figure 9: Immediate and deferred resolution

Part IV
Filesystem operations

20 Dentry operations

20.1 Synopsis

```
#include <libHX/io.h>

int HX_readlink(hxmc_t **buf, const char *path);
int HX_realpath(hxmc_t **buf, const char *path, unsigned int flags);
```

`HX_readlink` calls through to `readlink` to read the target of a symbolic link, and stores the result in the memory container referenced by `*buf` (similar to `HX_getl` semantics). If `*buf` is `NULL`, a new container will be allocated and a pointer to it stored in `*buf`. The container's content is naturally zero-terminated automatically. The return value of the function will be the length of the link target, or negative to indicate the system error value.

`HX_realpath` will normalize the given path by transforming various path components into alternate descriptions. The `flags` parameter controls its actions:

`HX_REALPATH_DEFAULT` A mnemonic for a set of standard flags: `HX_REALPATH_SELF | HX_REALPATH_PARENT`. Note that `HX_REALPATH_ABSOLUTE`, which would also be required to get libc's `realpath`(3) behavior, is not included in the set.

`HX_REALPATH_ABSOLUTE` Requests that the output path shall be absolute. In the absence of this flag, an absolute output path will only be produced if the input path is also absolute.

`HX_REALPATH_SELF` Request resolution of "." path components.

`HX_REALPATH_PARENT` Request resolution of ".." path components.

The result is stored in a memory container whose pointer is returned through `*buf`. The return value of the function will be negative to indicate a possible system error, or be positive non-zero for success.

21 Directory traversal

libHX provides a minimal readdir-style wrapper for cross-platform directory traversal. This is needed because the Win32 platforms does not have readdir, and there is some housekeeping to do on Unixish platforms, since the `dirent` structure needs allocation of a path-specific size.

21.1 Synopsis

```
#include <libHX/io.h>

struct HXdir *HXdir_open(const char *directory);
const char *HXdir_read(struct HXdir *handle);
void HXdir_close(struct HXdir *handle);
```

HXdir_open returns a pointer to its private data area, or NULL upon failure, in which case errno is preserved from the underlying system calls. HXdir_read causes the next entry from the directory to be fetched. The pointer returned by HXdir_read must not be freed, and the data is overwritten in subsequent calls to the same handle. If you want to keep it around, you will have to duplicate it yourself. HXdir_close will close the directory and free the private data it held.

21.2 Example

```
#include <errno.h>
#include <stdio.h>
#include <libHX/io.h>

struct HXdir *dh;
if ((dh = HXdir_open(".")) == NULL) {
        fprintf(stderr, "Could not open directory: %s\n", strerror(errno));
        return;
}
while ((dentry = HXdir_read(dh)) != NULL)
        printf("%s\n", dentry);
HXdir_close(dh);
```

This sample will open the current directory, and print out all entries as it iterates over them.

22 Directory operations

22.1 Synopsis

```
#include <libHX/io.h>

int HX_mkdir(const char *path, unsigned int mode);
int HX_rrmdir(const char *path);
```

HX_mkdir will create the directory given by path and all its parents that do not exist yet using the given mode. It is equivalent to the 'mkdir -p' shell command. It will return >0 for success, or -errno on error.

HX_rrmdir also maps to an operation commonly done on the shell, 'rm -Rf', deleting the directory given by path, including all files within it and its subdirectories. Errors during deletion are ignored, but if there was any, the errno value of the first one is returned negated.

23 File operations

23.1 Synopsis

```
#include <libHX/io.h>

int HX_copy_file(const char *src, const char *dest, unsigned int flags, ...);
int HX_copy_dir(const char *src, const char *dest, unsigned int flags, ...);
```

Possible flags that can be used with the functions:

HXF_KEEP Do not overwrite existing files.

HXF_UID Change the new file's owner to the UID given in the varargs section (...). HXF_UID is processed before HXF_GID.

HXF_GID Change the new file's group owner to the GID given in the varargs section. This is processed after HXF_UID.

Error checking is flakey.

HX_copy_file will return >0 on success, or -errno on failure. Errors can arise from the use of the syscalls open, read and write. The return value of fchmod, which is used to set the UID and GID, is actually ignored, which means verifying that the owner has been set cannot be detected with HX_copy_file alone (historic negligience?).

23.2 Filedescriptor I/O

```
#include <libHX/io.h>

ssize_t HXio_fullread(int fd, void *buf, size_t size, unsigned int flags);
ssize_t HXio_fullwrite(int fd, const void *buf, size_t size, unsigned int flags);
```

Since plain read(2) and write(2) may process only part of the buffer — even more likely so with sockets —, libHX provides two functions that calls these in a loop to retry said operations until the full amount has been processed. Since read and write can also be used with socket file descriptors, so can these.

Part V
Options and Configuration Files

24 Option parsing

libHX uses a table-based approach like libpopt[8]. It provides for both long and short options and the different styles associated with them, such as absence or presence of an equals sign for long options (`--foo=bar` and `--foo bar`), bundling (writing `-abc` for non-argument taking options `-a -b -c`), squashing (writing `-fbar` for an argument-requiring option `-f bar`). The "lone dash" that is often used to indicate standard input or standard output, is correctly handled[9], as in `-f -`.

A table-based approach allows for the parser to run as one atomic block of code (callbacks are, by definition, "special" exceptions), making it more opaque than an open-coded `getopt`(3) loop. You give it your argument vector and the table, snip the finger (call the parser function once), and it is done. In getopt on the other hand, the `getopt` function returns for every argument it parsed and needs to be called repeatedly.

24.1 Synopsis

```
#include <libHX/option.h>

struct HXoption {
        const char *ln;
        char sh;
        unsigned int type;
        void *ptr, *uptr;
        void (*cb)(const struct HXoptcb *);
        int val;
        const char *help, *htyp;
};

int HX_getopt(const struct HXoption *options_table, int *argc,
              const char ***argv, unsigned int flags);
```

The various fields of `struct HXoption` are:

`ln` The long option name, if any. May be `NULL` if none is to be assigned for this entry.

`sh` The short option name/character, if any. May be `'\0'` if none is to be assigned for this entry.

`type` The type of the entry, essentially denoting the type of the target variable.

`val` An integer value to be stored into `*(int *)ptr` when `HXTYPE_IVAL` is used.

`ptr` A pointer to the variable so that the option parser can store the requested data in it. The pointer may be `NULL` in which case no data is stored (but `cb` is still called if defined, with the data).

[8] The alternative would be an iterative, open-coded approach like `getopt`(3) requires.

[9] popt failed to do this for a long time.

uptr A user-supplied pointer. Its value is passed verbatim to the callback, and may be used for any purpose the user wishes. If `type` is `HXTYPE_SVAL`, it is the value in `uptr` that will be used to populate `*(const char **)ptr`. (The original `.sval` field has been removed in libHX 3.12.)

cb If not `NULL`, call out to the referenced function after the option has been parsed (and the results possibly be stored in `ptr`)

help A help string that is shown for the option when the option table is dumped by request (e. g. `yourprgram --help`)

htyp String containing a keyword to aid the user in understanding the available options during dump. See examples.

Due to the amount of fields, it is advised to use C99 named initializers to populate a struct, as they allow to omit unspecified fields, and assume no specific order of the members:

```
struct HXoption e = {.sh = 'f', .help = "Force"};
```

It is a sad fact that C++ has not gotten around to implement these yet. It is advised to put the option parsing code into a separate `.c` file that can then be compiled in C99 rather than C++ mode.

24.2 Type map

HXTYPE_NONE The option does not take any argument, but the presence of the option may be record by setting the `*(int *)ptr` to 1. Other rules apply when `HXOPT_INC` or `HXOPT_DEC` are specified as flags (see section 24.3).

HXTYPE_VAL Use the integer value specified by `ival` and store it in `*(int *)ptr`.

HXTYPE_SVAL Use the memory location specified by `sval` and store it in `*(const char **)ptr`.

HXTYPE_BOOL Interpret the supplied argument as a boolean descriptive (must be "yes", "no", "on", "off", "true", "false", "0" or "1") and store the result in `*(int *)ptr`.

HXTYPE_STRING The argument string is duplicated to a new memory region and the resulting pointer stored into `*(char **)ptr`. This incurs an allocation so that subsequently modifying the original argument string in any way will not falsely propagate.

HXTYPE_STRDQ The argument string is duplicated to a new memory region and the resulting pointer is added to the given HXdeque. Note that you often need to use deferred initialization of the options table to avoid putting `NULL` into the entry. See section 24.6.1.

The following table lists the types that map to the common integral and floating-point types. Signed and unsigned integeral types are processed using `strtol` and `strtoul`, respectively. `strtol` and `strtoul` will be called with automatic base detection. This usually means that a leading "0" indicates the string is given in octal (8) base, a leading "0x" indicates hexadecimal (16) base, and decimal (10) otherwise. `HXTYPE_LLONG`, `HXTYPE_ULLONG`, `HXTYPE_INT64` and `HXTYPE_UINT64` use `strtoll` and/or `strtoull`, which may not be available on all platforms.

type	Type of pointee	type	Type of pointee
HXTYPE_CHAR	char	HXTYPE_INT8	int8_t
HXTYPE_UCHAR	unsigned char	HXTYPE_UINT8	uint8_t
HXTYPE_SHORT	short	HXTYPE_INT16	int16_t
HXTYPE_USHORT	unsigned short	HXTYPE_UINT16	uint16_t
HXTYPE_INT	int	HXTYPE_INT32	int32_t
HXTYPE_UINT	unsigned int	HXTYPE_UINT32	uint32_t
HXTYPE_LONG	long	HXTYPE_INT64	int64_t
HXTYPE_ULONG	unsigned long	HXTYPE_UINT64	uint64_t
HXTYPE_LLONG	long long	HXTYPE_FLOAT	float
HXTYPE_ULLONG	unsigned long long	HXTYPE_DOUBLE	double
HXTYPE_SIZE_T	size_t		

Table 3: Integral and floating-point types for the libHX option parser

HXTYPE_FLOAT and HXTYPE_DOUBLE make use of strtod (strtof is not used). A corresponding type for the "long double" format is not specified, but may be implemented on behalf of the user via a callback (see section 24.8.4).

24.3 Flags

Flags can be combined into the type parameter by OR'ing them. It is valid to not specify any flags at all, but most flags collide with one another.

HXOPT_INC Perform an increment on the memory location specified by the *(int *)ptr pointer. Make sure the referenced variable is initialized before!

HXOPT_DEC Perform a decrement on the pointee.

Only one of HXOPT_INC and HXOPT_DEC may be specified at a time, and they require that the base type is HXTYPE_NONE, or they will have no effect. An example may be found in section 24.8.2.

HXOPT_NOT Binary negation of the argument directly after reading it from the command line into memory. Any of the three following operations are executed with the already-negated value.

HXOPT_OR Binary "OR"s the pointee with the specified/transformed value.

HXOPT_AND Binary "AND"s the pointee with the specified/transformed value.

HXOPT_XOR Binary "XOR"s the pointee with the specified/transformed value.

Only one of (HXOPT_OR, HXOPT_AND, HXOPT_XOR) may be specified at a time, but they can be used with any integral type (HXTYPE_UINT, HXTYPE_ULONG, etc.). An example can be found in section 24.8.3.

HXOPT_OPTIONAL This flag allows for an option to take zero or one argument. Needless to say that this can be confusing to the user. *iptables*'s "-L" option for example is one of this kind (though it does not use the libHX option parser). When this flag is used, "-f -b" is interpreted as -f without an argument, as is "-f --bar" — things that look like an option take precedence over an option with an optional argument. "-f -" of course denotes an option with an argument, as "-" is used to indicate standard input/output.

24.4 Special entries

HXopt provides two special entries via macros:

HXOPT_AUTOHELP Adds entries to recognize "`-?`" and "`--help`" that will display the (long-format) help screen, and "`--usage`" that will display the short option syntax overview. All three options will exit the program afterwards.

HXOPT_TABLEEND This sentinel marks the end of the table and is required on all tables. (See examples for details.)

24.5 Invoking the parser

```
int HX_getopt(const struct HXoption *options_table, int *argc,
              const char ***argv, unsigned int flags);
```

`HX_getopt` is the actual parsing function. It takes the option table, and a pointer to your `argc` and `argv` variables that you get from the `main` function. The parser will, unlike GNU getopt, literally "eat" all options and their arguments, leaving only non-options in `argv`, and `argc` updated, when finished. This is similar to how Perl's "Getopt::Long" module works. Additional flags can control the exact behavior of `HX_getopt`:

HXOPT_PTHRU "Passthrough mode". Any unknown options are not "eaten" and are instead passed back into the resulting `argv` array.

HXOPT_QUIET Do not print any diagnostics when encountering errors in the user's input.

HXOPT_HELPONERR Display the (long-format) help when an error, such as an unknown option or a violation of syntax, is encountered.

HXOPT_USAGEONERR Display the short-format usage syntax when an error is encountered.

HXOPT_RQ_ORDER Specifying this option terminates option processing when the first non-option argument in `argv` is encountered. This behavior is also implicit when the environment variable `POSIXLY_CORRECT` is set.

The return value can be one of the following:

HXOPT_ERR_SUCCESS Parsing was successful.

HXOPT_ERR_UNKN An unknown option was encountered.

HXOPT_ERR_VOID An argument was given for an option which does not allow one. In practice this only happens with "`--foo=bar`" when `--foo` is of type `HXTYPE_NONE`, `HXTYPE_VAL` or `HXTYPE_SVAL`. This does not affect "`--foo bar`", because this can be unambiguously interpreted as "`bar`" being a remaining argument to the program.

HXOPT_ERR_MIS Missing argument for an option that requires one.

HXOPT_ERR_AMBIG An abbreviation of a long option was ambiguous.

negative non-zero Failure on behalf of lower-level calls; errno.

24.6 Pitfalls

24.6.1 Staticness of tables

The following is an example of a possible pitfall regarding `HXTYPE_STRDQ`:

```
static struct HXdeque *dq;

static bool get_options(int *argc, const char ***argv)
{
        static const struct HXoption options_table[] = {
                {.sh = 'N', .type = HXTYPE_STRDQ, .q_strdq = dq,
                 .help = "Add name"},
                HXOPT_TABLEEND,
        };
        return HX_getopt(options_table, argc, argv, HXOPT_USAGEONERR) ==
               HXOPT_ERR_SUCCESS;
}

int main(int argc, const char **argv)
{
        dq = HXdeque_init();
        get_options(&argc, &argv);
        return 0;
}
```

The problem here is that `options_table` is, due to the `static` keyword, initialized at compile-time where `dq` is still `NULL`. To counter this problem and have it doing the right thing, you must remove the `static` qualifier on the options table when used with `HXTYPE_STRDQ`, so that it will be evaluated when it is first executed.

It was not deemed worthwhile to have `HXTYPE_STRDQ` take an indirect HXdeque (`struct HXdeque **`) instead just to bypass this issue. (Live with it.)

24.7 Limitations

The HX option parser has been influenced by both popt and Getopt::Long, but eventually, there are differences:

- Long options with a single dash ("`-foo bar`"). This unsupported syntax clashes very easily with support for option bundling or squashing. In case of bundling, "`-foo`" might actually be "`-f -o -o`", or "`-f oo`" in case of squashing. It also introduces redundant ways to specify options, which is not in the spirit of the author.

- Options using a "`+`" as a prefix, as in "`+foo`". Xterm for example uses it as a way to negate an option. In the author's opinion, using one character to specify options is enough — by GNU standards, a negator is named "`--no-foo`". Even Microsoft stuck to a single option introducing character (that would be "`/`").

- Table nesting like implemented in popt. HXopt has no provision for nested tables, as the need has not come up yet. It does however support chained processing (see section 24.8.5). You cannot do nested tables even with callbacks, as the new `argv` array is only put in place shortly before `HX_getopt` returns.

24.8 Examples

24.8.1 Basic example

The following code snippet should provide an equivalent of the GNU getopt sample[10].

```
#include <stdio.h>
#include <stdilb.h>
#include <libHX/option.h>

int main(int argc, const char **argv)
{
        int aflag = 0;
        int bflag = 0;
        char *cflag = NULL;

        struct HXoption options_table[] = {
                {.sh = 'a', .type = HXTYPE_NONE, .ptr = &aflag},
                {.sh = 'b', .type = HXTYPE_NONE, .ptr = &bflag},
                {.sh = 'c', .type = HXTYPE_STRING, .ptr = &cflag},
                HXOPT_AUTOHELP,
                HXOPT_TABLEEND,
        };

        if (HX_getopt(options_table, &argc, &argv, HXOPT_USAGEONERR) !=
            HXOPT_ERR_SUCCESS)
                return EXIT_FAILURE;

        printf("aflag = %d, bflag = %d, cvalue = %s\n",
               aflag, bflag, cvalue);

        while (*++argv != NULL)
                printf("Non-option argument %s\n", *argv);

        return EXIT_SUCCESS;
}
```

24.8.2 Verbosity levels

```
static int verbosity = 1; /* somewhat silent by default */
static const struct HXoption options_table[] = {
        {.sh = 'q', .type = HXTYPE_NONE | HXOPT_DEC, .q_int = &verbosity,
         .help = "Reduce verbosity"},
        {.sh = 'v', .type = HXTYPE_NONE | HXOPT_INC, .q_int = &verbosity,
         .help = "Increase verbosity"},
        HXOPT_TABLEEND,
};
```

This sample option table makes it possible to turn the verbosity of the program up or down, depending on whether the user specified `-q` or `-v`. By passing multiple `-v` flags, the verbosity

[10] `http://www.gnu.org/software/libtool/manual/libc/Example-of-Getopt.html#Example-of-Getopt`

can be turned up even more. The range depends on the "`int`" data type for your particular platform and compiler; if you want to have the verbosity capped at a specific level, you will need to use an extra callback:

```
static int verbosity = 1;

static void v_check(const struct HXoptcb *cbi)
{
        if (verbosity < 0)
                verbosity = 0;
        else if (verbosity > 4)
                verbosity = 4;
}

static const struct HXoption options_table[] = {
        {.sh = 'q', .type = HXTYPE_NONE | HXOPT_DEC, .q_int = &verbosity,
         .cb = v_check, .help = "Lower verbosity"},
        {.sh = 'v', .type = HXTYPE_NONE | HXOPT_INC, .q_int = &verbosity,
         .cb = v_check, .help = "Raise verbosity"},
        HXOPT_TABLEEND,
};
```

24.8.3 Mask operations

```
/* run on all CPU cores by default */
static unsigned int cpu_mask = ~0U;
/* use no network connections by default */
static unsigned int net_mask = 0;
static struct HXoption options_table[] = {
        {.sh = 'c', .type = HXTYPE_UINT | HXOPT_NOT | HXOPT_AND,
         .q_uint = &cpu_mask,
         .help = "Mask of cores to exclude", .htyp = "cpu_mask"},
        {.sh = 'n', .type = HXTYPE_UINT | HXOPT_OR, .q_uint = &net_mask,
         .help = "Mask of network channels to additionally use",
         .htyp = "channel_mask"},
        HXOPT_TABLEEND,
};
```

What this options table does is `cpu_mask &= ~x` and `net_mask |= y`, the classic operations of clearing and setting bits.

24.8.4 Support for non-standard actions

Supporting additional types or custom storage formats is easy, by simply using `HXTYPE_STRING`, `NULL` as the data pointer (usually by not specifying it at all), the pointer to your data in the user-specified pointer `uptr`, and the callback function in `cb`.

```
struct fixed_point {
        int integral;
        unsigned int fraction;
};
```

```
static struct fixed_point number;

static void fixed_point_parse(const struct HXoptcb *cbi)
{
        char *end;

        number.integral = strtol(cbi->data, &end, 0);
        if (*end == '\0')
                number.fraction = 0;
        else if (*end == '.')
                number.fraction = strtoul(end + 1, NULL, 0);
        else
                fprintf(stderr, "Illegal input.\n");
}

static const struct HXoption options_table[] = {
        {.sh = 'n', .type = HXTYPE_STRING, .cb = fixed_point_parse,
         .uptr = &number, .help = "Do this or that",
        HXOPT_TABLEEND,
};
```

24.8.5 Chained argument processing

On the first run, only `--cake` and `--fruit` is considered, which is then used to select the next set of accepted options. Note that `HXOPT_DESTROY_OLD` is used here, which causes the argv that is produced by the first invocation of `HX_getopt` in the `get_options` function to be freed as it gets replaced by a new argv again by `HX_getopt` in `get_cakes`/`get_fruit`. `HXOPT_DESTROY_OLD` is however *not* specified in the first invocation, because the initial argv resides on the stack and cannot be freed.

```
static bool get_cakes(int *argc, const char ***argv)
{
        struct HXoption option_table[] = {
                ...
        };
        return HX_getopt(cake_table, argc, argv,
               HXOPT_USAGEONERR | HXOPT_DESTROY_OLD) == HXOPT_ERR_SUCCESS;
}

static bool get_fruit(int *argc, const char ***argv)
{
        struct HXoption fruit_table[] = {
                ...
        };
        return HX_getopt(fruit_table, argc, argv,
               HXOPT_USAGEONERR | HXOPT_DESTROY_OLD) == HXOPT_ERR_SUCCESS;
}

static bool get_options(int *argc, const char ***argv)
{
```

```
        int cake = 0, fruit = 0;
        struct HXoption option_table[] = {
                {.ln = "cake",  .type = HXTYPE_NONE, .ptr = &cake},
                {.ln = "fruit", .type = HXTYPE_NONE, .ptr = &fruit},
                HXOPT_TABLEEND,
        };
        if (HX_getopt(option_table, argc, argv, HXOPT_PTHRU) !=
            HXOPT_ERR_SUCCESS)
                return false;
        if (cake)
                return get_cakes(argc, argv);
        else if (fruit)
                return get_fruit(argc, argv);
        return false;
}
```

25 Shell-style configuration file parser

libHX provides functions to read shell-style configuration files. Such files are common, for example, in `/etc/sysconfig` on Linux systems. The format is pretty basic; it only knows about "`key=value`" pairs and does not even have sections like INI files. Not relying on any features however makes them quite interchangable as the syntax is accepted by Unix Shells.

Lines beginning with a hash mark (`#`) are ignored, as are empty lines and unrecognized keys.

```
# Minimum / maximum values for automatic UID selection
UID_MIN=100
UID_MAX=65000

# Home directory base
HOME="/home"
#HOME="/export/home"
```

Any form of variable or parameter substitution or expansion is highly implementation specific, and is not supported in libHX's reader. Even Shell users should not rely on it as you never know in which context the configuration files are evaluated. Still, you will have to escape specific sequences like you would need to in Shell. The use of single quotes is acceptable. That means:

```
AMOUNT="US\$5"
AMOUNT='US$5'
```

25.1 Synopsis

```
#include <libHX/option.h>

int HX_shconfig(const char *file, const struct HXoption *table);
int HX_shconfig_pv(const char **path_vec, const char *file,
                   const struct HXoption *table, unsigned int flags);
struct HXmap *HX_shconfig_map(const char *file);
```

The shconfig parser reuses `struct HXoption` that fits very well in specifying name-pointer associations. `HX_shconfig` will read the given file using the key-to-pointer mappings from the table to store the variable contents. Of `struct HXoption`, described in section 24.1, only the "`ln`", "`type`" and "`ptr`" fields are used. The list of accepted types is described in section 24.2.

To parse a file, call `HX_shconfig` function with the corresponding parameters. If you want to read configuration files from different paths, i. e. to build up on default values, you can use `HX_shconfig_pv`[11], which is a variant for reading a file from multiple locations. Its purpose is to facilitate reading system-wide settings which are then overriden by a file in the users home directory, for example (per-setting-override). It is also possible to do per-file-override, that is, a file in the home directory has higher precedence than a system-wide one in such a way that the system-wide configuration file is not even read. This is accomplished by traversing the paths in the "other" direction (actually you have to turn the array around) and stopping at the first existing file by use of the `SHCONF_ONE` flag.

`HX_shconfig_map` will return all entries from the file in a HXmap, usable for parsing arbitrary keys without having to specify any static key table.

[11] pv = path vector

SHCONF_ONE Parsing files will stop after one file has been successfully parsed. This allows for a "personal overrides system config" style.

The call to HX_shconfig will either return >0 for success, 0 for no success (actually, this is never returned) and -errno for an error.

25.2 Example

25.2.1 Per-setting-override

This example sources key-value pairs from a configuration file in a system location (/etc) first, before overriding specific keys with new values from the file in the home directory.

```
long uid_min, uid_max;
char *passwd_file;
struct HXoption options_table[] = {
        {.ln = "UID_MIN",  .type = HXTYPE_LONG,   .ptr = &uid_min},
        {.ln = "UID_MAX",  .type = HXTYPE_LONG,   .ptr = &uid_max},
        {.ln = "PWD_FILE", .type = HXTYPE_STRING, .ptr = &passwd_file},
        HXOPT_TABLEEND,
};
const char *home = getenv("HOME");
const char *paths[] = {"/etc", home, NULL};
HX_shconfig(paths, "test.cf", options_table, 0);
```

25.2.2 Per-file-override

This particular example reads from the file in the home directory first (if it exists), but stops after it has been successfull, so any subsequent locations listed in the paths variable are not read. This has the effect that the file from the home directory has the highest priority too like in the previous example, but without any keys from the system files. Note the SHCONF_ONE flag.

```
const char *home = getenv("HOME");
const char *paths[] = {home, "/usr/local/etc", "/etc", NULL};
HX_shconfig_pv(paths, "test.cf", options_table, SHCONF_ONE);
```

Part VI
Systems-related components

26 Random numbers

26.1 Function overview

```
#include <libHX/misc.h>

int HX_rand(void);
unsigned int HX_irand(unsigned int min, unsigned int max);
double HX_drand(double min, double max);
```

HX_rand Retrieve the next random number.

HX_irand Retrieve the next random number and fold it so that $min \leq n < max$, where min and max are unsigned integers.

HX_drand Retrieve the next random number and fold it so that $min \leq n < max$, where min and max are double-precision floating point numbers.

26.2 Implementation information

On systems that provide operating system-level random number generators, predominantly Linux and Unix-alikes such as BSD and Solaris, these will be used when they are available and random numbers are requested through `HX_rand` or `HX_irand`.

On Linux, Solaris and the BSDs, this is `/dev/urandom`.

If no random number generating device is available (and libHX configured to use it), it will fall back to using the libc's `rand` function. If libc is selected for random number generation, `srand` will be called on library initialization with what is believed to be good defaults — usually this will be before a program's `main` function with normal linking, but may actually happen later when used with `dlopen`. The initial seed would be the current microtime when `gettimeofday` is available, or just the seconds with `time`. To counter the problem of different programs potentially using the same seed within a time window of a second due to the limited granularity of standard `time`, the seed is augmented by process ID and parent process ID where available.

`/dev/random` is not used on Linux because it may block during read, and `/dev/urandom` is just as good when there is entropy available. If you need definitive PRNG security, perhaps use one from a crypto suite such as OpenSSL.

27 Process management

The process code is experimental at this stage (just moved from the pam_mount codebase). As it also relies on the POSIX functions `fork`, `execv`, `execvp` and `pipe`(2), so it may not be available everywhere. Where this is the case, the functions will return `-ENOSYS`.

27.1 Process metadata structure

```
#include <libHX/proc.h>

struct HXproc {
        const struct HXproc_ops *p_ops;
        void *p_data;
        unsigned int p_flags;

        /* Following members should only be read */
        int p_stdin, p_stdout, p_stderr;
        int p_pid;
        char p_status;
        bool p_exited, p_terminated;
};
```

When creating a new process with the intent of running it asynchronously (using `HXproc_run_async`), the first three fields must be filled in by the user.

p_ops A table of callbacks, generally used for setting and/or restoring signals before/after execution. This member may be `NULL`.

p_data Free pointer for the user to supply. Will be passed to the callback functions when they are invoked.

p_flags Process creation flags, see below.

After the subprocess has been started, `HXproc_run_async` will have filled in some fields:

p_stdin If `HXPROC_STDIN` was specified in `p_flags`, `p_stdin` will be assigned the write side file descriptor of the subprocess's to-be stdin. The subprocess will get the read side file descriptor in this member. This is so that the correct fd is used in when `p_ops->p_postfork` is called.

p_stdout If `HXPROC_STDOUT` is specified in `p_flags`, `p_stdout` will be assigned the read side file descriptor of the subprocess's to-be stdout. The subprocess will get the write side file descriptor in this member.

p_stderr If `HXPROC_STDERR` is specified in `p_flags`, `p_stderr` will be assigned the read side file descriptor of the subprocess's to-be stderr, and the subprocess will get the write side fd.

p_pid The process ID of the spawned process.

Upon calling `HXproc_wait`, further fields will have been filled when the function returns:

p_exited Whether the process exited normally (cf. signalled/terminated).

p_terminated Whether the process was terminated (signalled).

p_status The exit status of the process or the termination signal.

27.1.1 Flags

Possible values for the `p_flags` member are:

HXPROC_STDIN The subprocess's stdin file descriptor shall be connected to the master program, that is, not inherit the stdin of the master. Cannot be used for `HXproc_run_sync` (because there would be no one to provide data in a sync operation).

HXPROC_STDOUT Connect the stdout file descriptor of the subprocess with the master. Cannot be used for `HXproc_run_sync`.

HXPROC_STDERR Connect the stderr file descriptor of the subprocess with the master. Cannot be used for `HXproc_run_sync`.

HXPROC_NULL_STDIN The subprocess's stdin file descriptor shall be connected to `/dev/null`. `HXPROC_STDIN` and `HXPROC_NULL_STDIN` are mutually exclusive.

HXPROC_NULL_STDOUT Connect the stdout file descriptor of the subprocess to `/dev/null`, thereby essentially discarding its output. `HXPROC_STDOUT` and `HXPROC_NULL_STDOUT` are mutuall exclusive.

HXPROC_NULL_STDERR Connect the stderr file descriptor of the subprocess to `/dev/null`, thereby essentially discarding its output. `HXPROC_STDERR` and `HXPROC_NULL_STDERR` are mutually exclusive.

HXPROC_VERBOSE Have the subprocess print an error message to stderr if exec'ing returned an error.

HXPROC_A0 `argv[0]` refers to program file, while `argv[1]` to the program invocation name, with `argv[2]` being the arguments. Without this flag, `argv[0]` will be both the program file and program invocation name, and arguments begin at `argv[1]`.

HXPROC_EXECV Normally, `execvp`(3) will be used which scans `$PATH` for the program. Use this flag to use `execv`(3) instead, which will not do such thing.

27.2 Callbacks

```
#include <libHX/proc.h>

struct HXproc_ops {
        void (*p_prefork)(void *);
        void (*p_postfork)(void *);
        void (*p_complete)(void *);
};
```

`struct HXproc_ops` provides a way to run user-specified functions just before the fork, after, and when the process has been waited for. They can be used to set and/or restore signals as needed, for example. The function pointers can be `NULL`. The `p_data` member is passed as an argument.

p_prefork Run immediately before calling `fork`(2). This is useful, for taking any action regarding signals, like setting `SIGCHLD` to `SIG_DFL`, or `SIGPIPE` to `SIG_IGN`, for example.

p_postfork Run in the subprocess (and only there) after forking. Useful to do a `setuid`(2) or other change in privilege level.

p_complete Run in HXproc_wait when the process has been waited for. Useful to restore the signal handler(s).

27.3 Process control

```
#include <libHX/proc.h>

int HXproc_run_async(const char *const *argv, struct HXproc *proc);
int HXproc_run_sync(const char *const *argv, unsigned int flags);
int HXproc_wait(struct HXproc *proc);
```

HXproc_run_async Start a subprocess according to the parameters in proc. Returns a negative errno code if something went wrong, or positive non-zero on success.

HXproc_run_sync Start a subprocess synchronously, similar to calling system(3), but with the luxury of being able to specify arguments as separate strings (via argv) rather than one big command line that is run through the shell. flags is a value composed of the HXproc flags mentioned above in section 27.1.1. HXPROC_STDIN, HXPROC_STDOUT and HXPROC_STDERR are ignored because there would be no one in a synchronous execution that could supply data to these file descriptors or read from them[12].

HXproc_wait Wait for a subprocess to terminate, if it has not already. It will also retrieve the exit status of the process and store it in the struct HXproc.

Return value will be positive non-zero on success, or negative on error. Underlying system function's errors are returned, plus:

EINVAL Flags were not accepted.

28 Helper headers

28.1 ctype helpers

Functions from the <ctype.h> header, including, but not limited to, isalpha, tolower, and so forth, are defined to take an "int" as first argument. Strings used in C programs are usually "char *", without any "signed" or "unsigned" qualifier. By a high-level view, which also matches daily common sense, characters (a. k. a. letters) have no notion of signedness — there is no "positive" or "negative" "A" in at least the Latin alphabet that is mapped into the ASCII set. In fact, char * could either be signed char * or unsigned char *, depending on the compiler settings. Only when you start interpreting and using characters as a number does such become important.

There come the problems. Characters are in the same class as numbers in C, that is, can be implicitly converted from or to a "number" (in this case, their ASCII code point) without causing a compiler warning. That may be practical in some cases, but is also a bit "unfortunate". Characters, when interpreted as the 8-bit signed numeric quantity they are implicitly convertable to, run from 0 to 127 and -128 to -1. Since the isalpha function and others from ctype.h take a (signed) int as argument means that values fed to isalpha are sign-extended, preserving negative values.

[12]Even for threads, please just use the async model.

```
/* "hyvää yötä", UTF-8 encoded */
const char h[] = {'h', 'y', 'v', 0xc3, 0xa4, 0xc3, 0xa4, ' ',
                  'y', 0xc3, 0xb6, 't', 0xc3, 0xa4};
```

When you now pass `h[3]` to `isalpha` for example (regardless of whether doing so actually produces a meaningful result), the CPU is instructed to copy "0xc3" into a register and sign-extend it (because "char" is often "signed char", see above), producing 0xfffffffc3 (-61). But passing -61 is not what was intended.

libHX's `ctype_helper.h` therefore provides wrappers with a different function signature that uses zero extension (not sign extension) by means of using an `unsigned` quantity. Currently this is `unsigned char`, because `isalpha`'s domain only goes from 0–255. The implication is that you cannot pass `EOF` to `HX_isalpha`.

```
#include <libHX/ctype_helper.h>

bool HX_isalnum(unsigned char c);
bool HX_isalpha(unsigned char c);
bool HX_isdigit(unsigned char c);
bool HX_islower(unsigned char c);
bool HX_isprint(unsigned char c);
bool HX_isspace(unsigned char c);
bool HX_isupper(unsigned char c);
bool HX_isxdigit(unsigned char c);
unsigned char HX_tolower(unsigned char c);
unsigned char HX_toupper(unsigned char c);
```

The `is*` functions also differ from ctype's in that they return `bool` instead of `int`. Not all functions from `ctype.h` are present either; `isascii`, `isblank`, `iscntrl`, `isgraph`, `ispunct` and `isxdigit` have been omitted as the author has never needed them so far.

28.2 libxml2 helpers

libxml2 uses an "`xmlChar`" type as an underlying type for the strings that it reads and outputs. `xmlChar` is typedef'ed to `unsigned char` by libxml2, causing compiler warnings related to differing signedness whenever interacting with strings from the outside world, which are usually just a pointer to `char`. Because casting would be a real chore, `libxml_helper.h` will do it by providing some wrappers with better argument types.

```
#include <libHX/libxml_helper.h>

int xml_strcmp(const xmlChar *a, const char *b);
int xml_strcasecmp(const xmlChar *a, const char *b);
char *xml_getprop(xmlNode *node, const char *attr);
char *xml_getnsprop(xmlNode *node, const char *nsprefix, const char *attr);
xmlAttr *xml_newprop(xmlNode *node, const char *attr);
xmlNode *xml_newnode(xmlNode *parent, const char *name, const char *value);
xmlAttr *xml_setprop(xmlNode *node, const char *name, const char *value);
```

The functions map to `strcmp`(3), `strcasecmp`(3), `xmlGetProp`, `xmlNewProp`, `xmlNewTextNode` and `xmlSetProp`, respectively.

`xml_getnsprop` works similar to `xmlGetNsProp`, but instead of taking a namespace URI, it does a lookup by namespace prefix. The argument order is also different compared to `xmlGetNsProp`.

28.3 wxWidgets

```
#include <libHX/wx_helper.hpp>
```

28.3.1 Shortcut macros

wxACV Expands to `wxALIGN_CENTER_VERTICAL`

wxCDF Expands to a set of common dialog flags for `wxDialog`s, which includes `wxDEFAULT_FRAME_STYLE` and a flag such that the dialog does not create a new window in the task bar (`wxFRAME_NO_TASKBAR`).

wxDPOS Expands to `wxDefaultPosition`.

wxDSIZE Expands to `wxDefaultSize`.

wxDSPAN Expands to `wxDefaultSpan`.

28.3.2 String conversion

```
wxString wxfu8(const char *);
wxString wxfv8(const char *);
const char *wxtu8(const wxString &);
```

wxfu8 Converts an UTF-8 string to a `wxString` object.

wxfv8 Converts an UTF-8 string to an entity usable by `wxPrintf`.

wxtu8 Converts a wxString to a pointer to char usable by `printf`. Note that the validity of the pointer is very limited and usually does not extend the statement in which it is used. Hence storing the pointer in a variable ("`const char *p = wxtu8(s);`") will make `p` pointing to an invalid region as soon as the assignment is done.

Part VII
Appendix