Multi-word Bitmask Library

This Bitmask library supports multi-word bitmask operations for applications programmed in 'C'. It works in conjunction with recent Linux kernel support for processor and memory placement on multiprocessor SMP and NUMA systems. The `cpuset` library, being developed in parallel, depends on this bitmask library.

Author: Paul Jackson

Address: pj@sgi.com

Date: 23 Sept 2005

This document is written using the outline processor Leo, and version controlled using CSSC. It is rendered using Python Docutils on reStructuredText extracted from Leo, directly into both html and LaTeX. The LaTeX is converted into pdf using the pdflatex utility. The html is converted into plain text using the lynx utility.

Table of Contents

1 What are bitmasks?

Bitmasks provide multi-word bit masks and operations thereon to do such things as set and clear bits, intersect and union masks, query bits, and display and parse masks.

The initial intended use for these bitmasks is to represent sets of CPUs and Memory Nodes, when configuring large SMP and NUMA systems. However there is little in the semantics of bitmasks that is specific to this particular use, and bitmasks should be usable for other purposes that had similar design requirements.

These bitmasks share the same underlying layout as the bitmasks used by the Linux kernel to represent sets of CPUs and Memory Nodes. Unlike the kernel bitmasks, these bitmasks use dynamically allocated memory and are manipulated via a pointer. This enables a program to work correctly on systems with various numbers of CPUs and Nodes, without recompilation.

There is a related cpuset library which uses the bitmask type provided here to represent sets of CPUs and Memory Nodes. The internal representation (as an array of unsigned longs, in little endian order) is directly compatible with the `sched_setaffinity(2)` and `sched_getaffinity(2)` system calls (added in Linux 2.6).

2 Ascii string representations

There are two ascii representations of these multi-word bitmasks, and this library provides display and parsing routines to convert both representations to and from the internal binary representation of bitmasks.

The hex mask representation of a bitmask of size 64, with bits 1,5,6,11-13,17-19 set looks like:

```
00000000,000E3862
```

and the decimal list representation for this same value looks like:

```
1,5,6,11-13,17-19
```

2.1 Hex mask

The hex mask representation of multi-word bit masks displays each 32-bit word in hex (zero filled), and for masks longer than one word, uses a comma separator between words. Words are displayed in big-endian order most significant first. And hex digits within a word are also in big-endian order.

The number of 32-bit words displayed is the minimum number needed to display all bits of the bitmask, based on the size of the bitmask.

Examples of the hex word bitmask display format:

```
A mask with just bit 0 set displays as "00000001".

A mask with just bit 127 set dis-
plays as "80000000,00000000,00000000,00000000".

A mask with just bit 64 set dis-
plays as "00000001,00000000,00000000".

A mask with bits 0, 1, 2, 4, 8, 16, 32 and 64 set dis-
plays as
"00000001,00000001,00010117".  The first "1" is for bit 64, the sec-
ond
for bit 32, the third for bit 16, and so forth, to the "7", which is for
bits 2, 1 and 0.

A mask with bits 32 through 39 set dis-
plays as "000000ff,00000000".

A 64 bit bitmask with bits 1, 5, 6, 11-13, and 17-
19 set displays as
"00000000,000E3862".
```

2.2 Decimal list

The decimal list representation of bitmasks represents them as a list of numbers and ranges of numbers.

This format supports a space separated list of one or more comma separated sequences of ascii decimal bit numbers and ranges, optionally modified by a *stride operator*.

Example of the decimal list bitmask display format:

```
0-4,9           # set bits 0, 1, 2, 3, 4, and 9
```

The *stride operator* is used to designate every N-th bit in a range It is written as a colon **":"** followed by the number N, with no spaces on either side of the colon.

Examples of the *stride operator*:

```
0-31:2          # the 16 even bits 0, 2, 4, ... 30

1-31:2          # the 16 odd bits 1, 3, 5, ... 31

0-31            # all 32 bits 0, 1, 2, ... 31
```

3 Calling, return and error conventions

As explained in more detail in the next section, all bitmask operations treat all bits, outside of the originally specified bit range from 0 to size-1, as if they were zero.

Most of the operations change the bitmask referenced by the first argument, and return a pointer to that bitmask, to allow convenient chaining of calls. However, be careful of such usage - it's really easy to code memory leaks this way. Each `struct bitmask *` pointer obtained from a call to bitmask_alloc needs to be free'd with a call (exactly *one* call) to bitmask_free.

Bit positions in bitmasks are zero based (not one based). The bit positions in a bitmask of size n are numbered 0 through n-1.

The Boolean functions return 1 (True) or 0 (False).

All but the first `struct bitmask *` pointers passed to any of the following operations are read-only, declared as: `const struct bitmask *`.

The unary operations, such as bitmask_complement, take two bitmask arguments, for the result and the source, in that order. The same `struct bitmask *` pointer may be passed for both arguments, in order to apply the operation in place.

The binary operations, such as bitmask_and, take three bitmask arguments, for the result and the two sources. Either source may be the same as the result. Indeed, all three arguments may be the same pointer (though it is not clear what purpose that would serve).

The shift operations zero fill, whether left or right shifting.

The range operations follow 'C' conventions in using closed left, open right intervals. That is, the range of bit positions determined by the pair of integer arguments (i, j) includes exactly all positions >= i and < j.

The bitmask_next function returns the bitmask size if all bits are clear above the requested position.

The bitmask_first and bitmask_last functions return the bitmask size if all bits are clear in the bitmask.

Two masks are equal if they have the same set bits, regardless of whether they have the same size.

Most of the operations or functions have no error return cases. They are defined so as to have valid returns for all well formed arguments. Of course, if the arguments are not well formed, then your application will probably exit with a Segmentation Violation. This is 'C' after all.

The bitmask_alloc function returns a zero pointer (NULL) and sets *errno* in the event that `malloc(3)` fails. See the `malloc(3)` man page for possible values of *errno* (**ENOMEM** being the most likely).

The bitmask_displayhex, bitmask_displaylist, bitmask_parsehex, and bitmask_parselist routines have more complex error and return conventions. See their detailed descriptions below.

4 Other bits always zero

All bitmask operations treat all bits, outside of the originally specified bit range from 0 to size-1, as if they were zero. Even bits that might actually be present, due to the use of some multiple of unsigned longs to represent the masks, are always zero, if they are outside the specified number of bits in the mask.

The specified number of bits in a bitmask (its size) is established in the bitmask_alloc call, and **never** changed after that.

Note: In particular, observe that the bitmask_copy function does **not** change the size of the target bitmask. Hence the result of copying a large bitmask to a small one will often not be equal to the original large bitmask - rather it will be shortened (to the smaller target size, with bits above that size zero'd).

For example, if you invoke:

```
struct bitmask *bmp = bitmask_alloc(17);
bitmask_setall(bmp);
bitmask_setbit(bmp, 21);
```

then the calls:

```
{
   int x = bitmask_last(bmp);
   int y = bitmask_isbitset(999);
}
```

will set x to 16 (the 17 bits are numbered 0 to 16), not 21 or some other higher number, and they will set y to 0.

Requests to set bits outside those in the range specified in the initial bitmask_alloc are ignored and do not cause any error.

Requests to display or query bits outside those in the range specified in the initial bitmask_alloc always behave as if those bits were present and zero.

5 Internal binary representation

The 'C' code that uses bitmasks sees only a `struct bitmask *` opaque pointer.

Hidden within the implementation of bitmasks, a `struct bitmask` is simply:

```
struct bitmask {
        unsigned int size;      /* size in bits of bit-
mask */
        unsigned long *maskp;   /* array of un-
signed longs */
};
```

This structure, and the variable length array of unsigned long words to which it points are allocated using `malloc(3)` in the calls to bitmask_alloc, and deallocated using `free(3)` in the calls to bitmask_free.

The `maskp` array of unsigned longs is arranged the same as the bitmask operands to the `sched_setaffinity(2)` and `sched_getaffinity(2)` system calls (added in Linux 2.6). As of this writing this is the same layout as is used by the kernels `cpumask_t` and `nodemask_t` types and the task struct `cpus_allowed` and `mems_allowed` fields.

This representation places multiple unsigned long words in little endian order - low order word first. Within each unsigned long, bits are addressed in 'natural C' order, as 1<<n, for n between 0 and 31 on 32 bit architectures, and between 0 and 63 on 64 bit architectures.

The implementation of this bitmask library reserves the right to extend or change this structure and other details of this internal representation.

6 Comparing these bitmasks with the Linux kernel

This section compares this bitmask library with the implementation of bitmasks in the Linux kernel, as of version 2.6.

Users of this library don't actually need to understand these differences. However users already familiar with kernel bitmasks may find this comparison helpful. And this comparison provides an interesting way to present a few of the design tradeoffs that were made in creating this library.

- This library implementation and API is optimized for ease of porting, ease of use and flexible runtime behaviour.
- The Linux kernel bitmasks are optimized for optimum space and time performance with compiled in fixed sizing of critical cpu and node masks.
- This library provides a larger, more complete and consistent set of bitmask routines than the kernel bitmasks.
- All calls are actual subroutine calls, not gcc inline functions or macros.

6.1 Dynamic Memory

The representation of bitmasks in the Linux kernel, as of this writing, is essentially:

```
struct { unsigned long bitmask[N]; };
```

None of this uses memory allocated dynamically at runtime. Instead, all sizes are known at compile time, and the compiler, along with some inline functions and macros, sizes each bitmask to a hardcoded size, such as NR_CPUS (number of CPUs which that kernel will support).

The current representation of bitmasks in this library, as noted in the previous section, is:

```
struct bitmask {
        unsigned int size;      /* size in bits of bit-
mask */
        unsigned long *maskp;   /* array of un-
signed longs */
};
```

Both this structure, and the array `maskp` it references, are dynamically allocated at runtime.

User programs, unlike a specific compilation of the kernel, usually avoid hardcoding the number of CPUs and Memory Nodes which they support. It is for this simple reason that this library uses dynamic memory allocation and runtime sizing, instead of the static allocation and compile time sizing used by the kernel's bitmask implementation.

6.2 Portable C

This library is implemented in Portable C, and presents an API that can easily be used in any Portable C code. The implementing code is kept simple, portable and easy to develop and maintain. The code is not optimized for critical inner loop performance requirements.

The Linux kernel bitmasks make essential use of gcc extensions in order to provide the compile time sizing and optimum performance that is required for use in critical scheduler and allocator loops.

6.3 Larger API

In order to make it easy to code bitmask operations, and reduce the costs of coding errors in the applications using these routines, this library provides a larger, more complete and consistent set of bitmask routines than the kernel bitmasks.

The kernel has some carefully optimized bitmasks routines for specific architectures, which makes it a bit more difficult to keep their API as straightforward and consistent as this library. And it avoids providing routines that don't have an actual use in existing kernel code.

6.4 No macros or access to bitmask internals

The implementation of this bitmask library uses no gcc inline functions or preprocessor macro functions of `struct bitmask` in the bitmask.h header file, and produces no code in the application binary that knows the internals of `struct bitmask`.

Everything that looks like a function on bitmasks is a real function call into the libmask.so library. The `struct bitmask` structure is declared in `bitmask.h` without its members defined, as simply:

```
struct bitmask;
```

The reason that there are no such macro or inline functions in the bitmask.h header file is that without access to the internals of the bitmask structure, they could not be compiled.

There are two reasons that the internals of the bitmask structure are not accessible in the bitmask.h header.

One reason is to discourage code that looks inside a structure that is intended to be opaque. This reduces the risk that some future change to the implementation internals of this structure will adversely impact existing application binaries using this API.

The other reason is to discourage bitmask structure assignment, which reduces the risk of memory corruption bugs from misuse of this structure.

Code such as the following will not compile, but it if did, would typically result in memory corruption.

```
#include <bitmask.h>

struct bitmask *bmp1 = bitmask_alloc(32);
struct bitmask *bmp2 = bitmask_alloc(32);

*bmp2 = *bmp1;       /* 1. Doesn't compile -
if it did, would be unsafe assignment */
bitmask_free(bmp1);  /* 2. Free it once */
bitmask_free(bmp2);  /* 3. Free it twice: corrupted malloc heap */
```

At step [1], the dynamic memory allocated to bmp2 is lost (memory leak) and the dynamic memory allocated to bmp1 is now referenced twice. At step [2], the memory for bmp1 is free'd once. At step [3], it is free'd again, resulting in a corrupt malloc heap, and likely an obscure crash later in the program execution.

Assigning the pointers, and passing them as arguments, is acceptable, so long as you are careful not to cause a memory leak by assigning to a *struct bitmask* * pointer that is currently referencing some other dynamically allocated bitmask which should first be freed via that pointer.

```
#include <bitmask.h>

struct bitmask *bmp1;
struct bitmask *bmp2 = bitmask_alloc(32);
extern void f(struct bitmask *);
bmp1 = bmp2;         /* ok */
f(bmp1);             /* ok */
```

Both of the above reasons reflect the same basic design tradeoff to prefer robust, portable code over aggressive extraction of performance.

The above choices also enable application binaries to continue working correctly in the face of internal changes to the bitmask library, without requiring the applications to be recompiled. This is quite unlike the Linux kernel, which is routinely recompiled in its entirety, as a single unit, anytime any part of it changes.

7 Bitmask Library Functions

The following inclusion and linkage provides access to the bitmask API from 'C' code:

```
#include <bitmask.h>
/* link with -lbitmask */
```

The following functions are supported in the 'C' bitmask API. In some cases, 'C' equivalent code is shown, as if bitmasks were a single unsigned long, even though they are packaged in a structure, referenced by a pointer, and actually contain an array of perhaps multiple unsigned longs.

None of these operations other than bitmask_alloc allocate new bitmasks, and none of them other than bitmask_free free existing bitmasks.

None of these operations other than bitmask_alloc set or change the size of a bitmask.

The following functions are supported in the 'C' bitmask API:

- Allocate and free `struct bitmask *`
  - bitmask_alloc (n) - Allocate a new struct bitmask with a size of n bits
  - bitmask_free (struct bitmask * bmp) - Free struct bitmask
- Display and parse ascii string representations
  - bitmask_displayhex (buf, len, bmp) - Write hex word representation of bmp to buf
  - bitmask_displaylist (buf, len, bmp) - Write decimal list representation of bmp to buf
  - bitmask_parsehex (buf, bmp) - Parse hex word representation in buf to bmp
  - bitmask_parselist (buf, bmp) - Parse decimal list representation in buf to bmp
- Basic initialization operations
  - bitmask_copy (bmp1, bmp2) - Copy bmp2 to bmp1
  - bitmask_setall (bmp) - Set all bits in bitmask: bmp = ~0
  - bitmask_clearall (bmp) - Clear all bits in bitmask: bmp = 0

- Interface aids for kernel sched_{set,get}affinity system calls
  - bitmask_nbytes (bmp) - Length in bytes of mask - use as second argument to these calls
  - bitmask_mask (bmp) - Direct pointer to bit mask - use as third argument to these calls
- Unary numeric queries
  - bitmask_nbits (bmp) - Size in bits of entire bitmask
  - bitmask_weight (bmp) - Hamming Weight: number of set bits
- Unary Boolean queries
  - bitmask_isbitset (bmp, i) - True if specified bit **i** is set
  - bitmask_isbitclear (bmp, i) - True if specified bit **i** is clear
  - bitmask_isallset (bmp) - True if all bits are set
  - bitmask_isallclear (bmp) - True if all bits are clear
- Single bit operations
  - bitmask_setbit (bmp, i) - Set a single bit **i** in bitmask
  - bitmask_clearbit (bmp, i) - Clear a single bit **i** in bitmask
- Binary Boolean operations: bmp1 op? bmp2
  - bitmask_equal (bmp1, bmp2) - True if two bitmasks are equal
  - bitmask_subset (bmp1, bmp2) - True if first bitmask is subset of second
  - bitmask_disjoint (bmp1, bmp2) - True if two bitmasks don't overlap
  - bitmask_intersects (bmp1, bmp2) - True if two bitmasks do overlap
- Range operations
  - bitmask_setrange (bmp, i, j) - Set bits of bitmask in specified range [i, j)
  - bitmask_clearrange (bmp, i, j) - Clear bits of bitmask in specified range
  - bitmask_keeprange (bmp, i, j) - Clear all but specified range
- Unary operations: bmp1 = op(bmp2)
  - bitmask_complement (bmp1, bmp2) - Complement: bmp1 = ~bmp2
  - bitmask_shiftright (bmp1, bmp2, n) - Right shift: bmp1 = bmp2 >> n
  - bitmask_shiftleft (bmp1, bmp2, n) - Left shift: bmp1 = bmp2 << n
- Binary operations: bmp1 = bmp2 op bmp3

- bitmask_and (bmp1, bmp2, bmp3) - Logical *and* of two bitmasks: bmp1 = bmp2 & bmp3
- bitmask_andnot (bmp1, bmp2, bmp3) - Logical *andnot* of two bitmasks: bmp1 = bmp2 & ~bmp3
- bitmask_or (bmp1, bmp2, bmp3) - Logical *or* of two bitmasks: bmp1 = bmp2 | bmp3
- bitmask_eor (bmp1, bmp2, bmp3) - Logical *eor* of two bitmasks: bmp1 = bmp2 ^ bmp3

- Iteration operators
  - bitmask_first (bmp) - Number of lowest set bit (min)
  - bitmask_next (bmp, i) - Number of next set bit above given bit **i**
  - bitmask_rel_to_abs_pos (bmp, n) - Absolute position of nth set bit
  - bitmask_abs_to_rel_pos (bmp, n) - Relative position amongst set bits of bit n
  - bitmask_last (bmp) - Number of highest set bit (max)

7.1 bitmask_alloc

```
struct bitmask *bitmask_alloc(unsigned int nbits);
```

Allocate a new struct bitmask with a size of `nbits` bits.

This is the only bitmask function that creates bitmasks.

Each `struct bitmask *` pointer obtained from a call to bitmask_alloc needs to be free'd with a call (exactly *one* call) to bitmask_free.

The bitmask_alloc function uses the underlying `malloc(3)` routine to obtain memory. It returns a zero pointer (NULL) and sets *errno* in the event that `malloc(3)` fails. See the `malloc(3)` man page for possible values of *errno* (**ENOMEM** being the most likely).

The size of a bitmask, as specified in the *bitmask_alloc* call that created it, is *never* changed by subsequent bitmask operations. Bits at positions outside the range zero to nbits-1 are always zero. Attempts to modify bits at such positions are always ignored, doing nothing, successfully.

For portable code, when allocating a bitmask to handle the CPUs or Memory Nodes on a system, the number of CPUs or Nodes should not be hardcoded, but obtained dynamically from the system. The routines `cpuset_cpus_nbits()` and `cpuset_mems_nbits()` in the related *libcpuset* library provide the maximum number of CPUs or Memory Nodes that the operating system was compiled to support. Use these values to size CPU and Memory Node bitmasks for calls into the *libcpuset* library.

7.2 bitmask_free

```
void bitmask_free(struct bitmask *bmp);
```

Free a bitmask struct.

This call frees the memory assigned to a bitmask. It is the only function that frees bitmasks. The *struct bitmask *‘* pointer must have been returned by a previous call to bitmask_alloc. The memory is not cleared. If *bmp* is NULL, no operation is performed.

Each `struct bitmask *` pointer obtained from a call to bitmask_alloc needs to be free'd with a call (exactly *one* call) to bitmask_free.

bitmask_free returns no value.

7.3 bitmask_displayhex

```
int bitmask_displayhex(char *buf, int len, const struct bitmask *bmp);
```

Write hex mask representation of bmp to buf.

7.4 bitmask_displaylist

```
int bitmask_displaylist(char *buf, int len, const struct bitmask *bmp);
```

Write decimal list representation of bmp to buf.

7.5 bitmask_parsehex

```
int bitmask_parsehex(const char *buf, struct bitmask *bmp);
```

Parse hex mask representation in buf to bmp.

7.6 bitmask_parselist

```
int bitmask_parselist(const char *buf, struct bitmask *bmp);
```

Parse decimal list representation in buf to bmp.

7.7 bitmask_copy

```
struct bitmask *bitmask_copy(struct bitmask *bmp1, const bitmask *bmp2);
```

Copy the value of bitmask bmp2 to bitmask bmp1. If the size of bmp1 is smaller than bmp2, then bits set in bmp2 beyond what fit in bmp1 are lost in the bmp1 copy. If the size of bmp1 is larger than bmp2,then bits set in bmp1 beyond what bmp2 specifies are cleared. The target bitmask bmp1 is not resized in any case.

Returns the pointer bmp1.

7.8 bitmask_setall

```
struct bitmask *bitmask_setall(struct bitmask *bmp);
```

Sets all bits in bitmask bmp.

Returns the pointer bmp.

7.9 bitmask_clearall

```
struct bitmask *bitmask_clearall(struct bitmask *bmp);
```

Clears all bits in bitmask bmp.

Returns the pointer bmp.

7.10 bitmask_nbytes

```
unsigned int bitmask_nbytes(struct bitmask *bmp);
```

Returns the length in bytes of a bitmask.

This is useful as the second argument to systems calls (new in Linux 2.6) `sched_setaffinity(2)` and `sched_getaffinity(2)`

Example:

```
/* Bind current process to the 3rd CPU (num-
ber 2) of a possible 64 CPUs */

struct bitmask *bmp = bitmask_alloc(64);
```

```
bitmask_setbit(2);
if (sched_setaffinity(0, bitmask_nbytes(bmp), bit-
mask_mask(bmp)) < 0)
        ... handle error ...
bitmask_free(bmp);
```

7.11 bitmask_mask

```
unsigned long *bitmask_mask(struct bitmask *bmp);
```

Returns a direct pointer to the unsigned long mask array of a bitmask.

This is useful as the third argument to systems calls (new in Linux 2.6) `sched_setaffinity(2)` and `sched_getaffinity(2)`

See also the example for bitmask_nbytes, above.

7.12 bitmask_nbits

```
unsigned int bitmask_nbits(const struct bitmask *bmp);
```

Size in bits of entire bitmask.

7.13 bitmask_weight

```
unsigned int bitmask_weight(const struct bitmask *bmp);
```

Hamming Weight: number of set bits.

7.14 bitmask_isbitset

```
int bitmask_isbitset(const struct bitmask *bmp, unsigned int i);
```

True if specified bit i is set. Always false if i >= bitmask_nbits_().

7.15 bitmask_isbitclear

```
int bitmask_isbitclear(const struct bitmask *bmp, unsigned int i);
```

True if specified bit i is clear. Always true if i >= bitmask_nbits_().

7.16 bitmask_isallset

```
int bitmask_isallset(const struct bitmask *bmp);
```

True if all bits from positions 0 to bitmask_nbits_() - 1 are set.

7.17 bitmask_isallclear

```
int bitmask_isallclear(const struct bitmask *bmp);
```

True if all bits from positions 0 to bitmask_nbits_() - 1 are clear.

7.18 bitmask_setbit

```
struct bitmask *bitmask_setbit(struct bitmask *bmp, unsigned int i);
```

Set a single bit i in bitmask. Does nothing successfully if i >= bitmask_nbits_(). Returns the `bmp` pointer passed in.

7.19 bitmask_clearbit

```
struct bitmask *bitmask_clearbit(struct bitmask *bmp, unsigned int i);
```

Clear a single bit i in bitmask. Does nothing successfully if i >= bitmask_nbits_(). Returns the `bmp` pointer passed in.

7.20 bitmask_equal

```
int bitmask_equal(const struct bitmask *bmp1, const bitmask *bmp2);
```

True if two bitmasks are equal.

7.21 bitmask_subset

```
int bitmask_subset(const struct bitmask *bmp1, const bitmask *bmp2);
```

True if first bitmask is subset of second.

7.22 bitmask_disjoint

```
int bitmask_disjoint(const struct bitmask *bmp1, const bitmask *bmp2);
```

True if two bitmasks don't overlap.

7.23 bitmask_intersects

```
int bitmask_intersects(const struct bitmask *bmp1, const bitmask *bmp2);
```

True if two bitmasks do overlap.

7.24 bitmask_setrange

```
struct bitmask *bitmask_setrange(struct bitmask *bmp, unsigned int i, unsigned int
j);
```

Set bits of bitmask in specified range [i, j).

7.25 bitmask_clearrange

```
struct bitmask *bitmask_clearrange(struct bitmask *bmp, unsigned int i, unsigned
int j);
```

Clear bits of bitmask in specified range.

7.26 bitmask_keeprange

```
struct bitmask *bitmask_keeprange(struct bitmask *bmp, unsigned int i, unsigned
int j);
```

Clear all but specified range.

7.27 bitmask_complement

```
struct bitmask *bitmask_complement(struct bitmask *bmp1, const bitmask *bmp2);
```

Complement: bmp1 = ~bmp2.

7.28 bitmask_shiftright

```
struct bitmask *bitmask_shiftright(struct bitmask *bmp1, const bitmask *bmp2, un-
signed int n);
```

Right shift: bmp1 = bmp2 >> n.

7.29 bitmask_shiftleft

```
struct bitmask *bitmask_shiftleft(struct bitmask *bmp1, const bitmask *bmp2, un-
signed int n);
```

Left shift: bmp1 = bmp2 << n.

7.30 bitmask_and

```
struct bitmask *bitmask_and(struct bitmask *bmp1, const bitmask *bmp2, const bit-
mask *bmp3);
```

Logical *and* of two bitmasks: bmp1 = bmp2 & bmp3.

The bits that are set in the result bitmask, bmp1, are the intersection of the bits that are set in the source bitmasks bmp2 and bmp3.

7.31 bitmask_andnot

```
struct bitmask *bitmask_andnot(struct bitmask *bmp1, const bitmask *bmp2, const
bitmask *bmp3);
```

Logical *andnot* of two bitmasks: bmp1 = bmp2 & ~bmp3.

The bits that are set in the result bitmask, bmp1, are the the bits that are set in the source bitmask bmp2 but not in bmp3.

7.32 bitmask_or

```
struct bitmask *bitmask_or(struct bitmask *bmp1, const bitmask *bmp2, const bit-
mask *bmp3);
```

Logical *or* of two bitmasks: bmp1 = bmp2 | bmp3.

The bits that are set in the result bitmask, bmp1, are the union of the bits that are set in either source bitmasks bmp2 or bmp3.

7.33 bitmask_eor

```
struct bitmask *bitmask_eor(struct bitmask *bmp1, const bitmask *bmp2, const bit-
mask *bmp3);
```

Logical *eor* of two bitmasks: bmp1 = bmp2 ^ bmp3.

The bits that are set in the result bitmask, bmp1, are the symmetric difference (in one or the other but not both) of the bits that are set in the source bitmasks bmp2 or bmp3.

7.34 bitmask_first

```
int bitmask_first(const struct bitmask *bmp);
```

Number of lowest set bit (min).

7.35 bitmask_next

```
unsigned int bitmask_next(const struct bitmask *bmp, unsigned int i);
```

Number of next set bit above given bit i.

7.36 bitmask_rel_to_abs_pos

```
unsigned int bitmask_rel_to_abs_pos(const struct bitmask *bmp, unsigned int n);
```

Return the number of the nth set bit. Calling bitmask_rel_to_abs_pos(bmp, 0) is equivalent to calling bitmask_first(bmp). Calling bitmask_rel_to_abs_pos(bmp, bitmask_weight(bmp) - 1) is equivalent to calling bitmask_last(bmp).

7.37 bitmask_abs_to_rel_pos

```
unsigned int bitmask_abs_to_rel_pos(const struct bitmask *bmp, unsigned int n);
```

Return the relative bit position, amongst just the set bits, of the nth bit, if the nth bit is set. If the nth bit is not set, return bitmask_nbytes. For the bit positions that are set, bitmask_abs_to_rel_pos is the inverse of bitmask_rel_to_abs_pos.

7.38 bitmask_last

```
unsigned int bitmask_last(const struct bitmask *bmp);
```

Number of highest set bit (max).