

NAME

mbuf – memory management in the kernel IPC subsystem

SYNOPSIS

```
#include <sys/param.h>
#include <sys/system.h>
#include <sys/mbuf.h>
```

Mbuf allocation macros

```
MGET(struct mbuf *mbuf, int how, short type);
MGETHDR(struct mbuf *mbuf, int how, short type);
MCLGET(struct mbuf *mbuf, int how);
MEXTADD(struct mbuf *mbuf, caddr_t buf, u_int size,
        void (*free)(void *opt_args), void *opt_args, short flags,
        int type);
MEXTFREE(struct mbuf *mbuf);
MEXT_ADD_REF(struct mbuf *mbuf);
MEXT_REM_REF(struct mbuf *mbuf);
MFREE(struct mbuf *mbuf, struct mbuf *successor);
```

Mbuf utility macros

```
void *
mtod(struct mbuf *mbuf, type);

int
MEXT_IS_REF(struct mbuf *mbuf);
M_COPY_PKTHDR(struct mbuf *to, struct mbuf *from);
M_ALIGN(struct mbuf *mbuf, u_int len);
MH_ALIGN(struct mbuf *mbuf, u_int len);

int
M_LEADINGSPACE(struct mbuf *mbuf);

int
M_TRAILINGSPACE(struct mbuf *mbuf);
M_PREPEND(struct mbuf *mbuf, int len, int how);
MCHTYPE(struct mbuf *mbuf, u_int type);

int
M_WRITABLE(struct mbuf *mbuf);
```

Mbuf allocation functions

```
struct mbuf *
m_get(int how, int type);

struct mbuf *
m_getm(struct mbuf *orig, int len, int how, int type);
```

```

struct mbuf *
m_getclr(int how, int type);

struct mbuf *
m_gethdr(int how, int type);

struct mbuf *
m_free(struct mbuf *mbuf);

void
m_freem(struct mbuf *mbuf);

```

Mbuf utility functions

```

void
m_adj(struct mbuf *mbuf, int len);

struct mbuf *
m_prepend(struct mbuf *mbuf, int len, int how);

struct mbuf *
m_pullup(struct mbuf *mbuf, int len);

struct mbuf *
m_copym(struct mbuf *mbuf, int offset, int len, int how);

struct mbuf *
m_copypacket(struct mbuf *mbuf, int how);

struct mbuf *
m_dup(struct mbuf *mbuf, int how);

void
m_copydata(const struct mbuf *mbuf, int offset, int len, caddr_t buf);

void
m_copyback(struct mbuf *mbuf, int offset, int len, caddr_t buf);

struct mbuf *
m_devget(char *buf, int len, int offset, struct ifnet *ifp,
          void (*copy)(char *from, caddr_t to, u_int len));

void
m_cat(struct mbuf *m, struct mbuf *n);

u_int
m_fixhdr(struct mbuf *mbuf);

u_int
m_length(struct mbuf *mbuf, struct mbuf **last);

struct mbuf *
m_split(struct mbuf *mbuf, int len, int how);

```

DESCRIPTION

An mbuf is a basic unit of memory management in the kernel IPC subsystem. Network packets and socket buffers are stored in mbufs. A network packet may span multiple mbufs arranged into a chain (linked list), which allows adding or trimming network headers with little overhead.

While a developer should not bother with mbuf internals without serious reason in order to avoid incompatibilities with future changes, it is useful to understand the mbuf's general structure.

An mbuf consists of a variable-sized header and a small internal buffer for data. The mbuf's total size, `MSIZE`, is a machine-dependent constant defined in `machine/param.h`. The mbuf header includes:

```

    m_next      a pointer to the next buffer in the chain
    m_nextpkt  a pointer to the next chain in the queue
    m_data     a pointer to the data
    m_len      the length of the data
    m_type     the type of data
    m_flags    the mbuf flags

```

The mbuf flag bits are defined as follows:

```

/* mbuf flags */
#define M_EXT      0x0001 /* has associated external storage */
#define M_PKTHDR  0x0002 /* start of record */
#define M_EOR     0x0004 /* end of record */
#define M_RDONLY  0x0008 /* associated data marked read-only */
#define M_PROTO1  0x0010 /* protocol-specific */
#define M_PROTO2  0x0020 /* protocol-specific */
#define M_PROTO3  0x0040 /* protocol-specific */
#define M_PROTO4  0x0080 /* protocol-specific */
#define M_PROTO5  0x0100 /* protocol-specific */

/* mbuf pkthdr flags, also in m_flags */
#define M_BCAST   0x0200 /* send/received as link-level broadcast */
#define M_MCAST   0x0400 /* send/received as link-level multicast */
#define M_FRAG    0x0800 /* packet is fragment of larger packet */
#define M_FIRSTFRAG 0x1000 /* packet is first fragment */
#define M_LASTFRAG 0x2000 /* packet is last fragment */

```

The available mbuf types are defined as follows:

```

/* mbuf types */
#define MT_FREE    0 /* should be on free list */
#define MT_DATA    1 /* dynamic (data) allocation */
#define MT_HEADER  2 /* packet header */
#define MT_SONAME  8 /* socket name */
#define MT_FTABLE  11 /* fragment reassembly header */
#define MT_CONTROL 14 /* extra-data protocol message */
#define MT_OOBDATA 15 /* expedited data */

```

If the `M_PKTHDR` flag is set, a `struct pkthdr m_pkthdr` is added to the mbuf header. It contains a pointer to the interface the packet has been received from (`struct ifnet *rcvif`), and the total packet length (`int len`).

If small enough, data is stored in the mbuf's internal data buffer. If the data is sufficiently large, another mbuf may be added to the chain, or external storage may be associated with the mbuf. `MHLEN` bytes of data can fit into an mbuf with the `M_PKTHDR` flag set, `MLEN` bytes can otherwise.

If external storage is being associated with an mbuf, the `m_ext` header is added at the cost of losing the internal data buffer. It includes a pointer to external storage, the size of the storage, a pointer to a function used for freeing the storage, a pointer to an optional argument that can be passed to the function, and a pointer to a reference counter. An mbuf using external storage has the `M_EXT` flag set.

The system supplies a macro for allocating the desired external storage buffer, `MEXTADD`.

The allocation and management of the reference counter is handled by the subsystem. The developer can check whether the reference count for the given mbuf's external storage is greater than 1 with the `MEXT_IS_REF` macro. Similarly, the developer can directly add and remove references, if absolutely necessary, with the use of the `MEXT_ADD_REF` and `MEXT_REM_REF` macros.

The system also supplies a default type of external storage buffer called an "mbuf cluster". Mbuf clusters can be allocated and configured with the use of the `MCLGET` macro. Each cluster is `MCLBYTES` in size, where `MCLBYTES` is a machine-dependent constant. The system defines an advisory macro `MINCLSIZE`, which is the smallest amount of data to put into a cluster. It's equal to the sum of `MLEN` and `MHLEN`. It is typically preferable to store data into an mbuf's data region, if size permits, as opposed to allocating a separate mbuf cluster to hold the same data.

Macros and Functions

There are numerous predefined macros and functions that provide the developer with common utilities.

mtod(*mbuf*, *type*)

Convert an mbuf pointer to a data pointer. The macro expands to the data pointer cast to the pointer of the specified type. **Note:** It is advisable to ensure that there is enough contiguous data in the mbuf. See `m_pullup()` for details.

MGET(*mbuf*, *how*, *type*)

Allocate an mbuf and initialize it to contain internal data. *mbuf* will point to the allocated mbuf on success, or be set to `NULL` on failure. The *how* argument is to be set to `M_TRYWAIT` or `M_DONTWAIT`. It specifies whether the caller is willing to block if necessary. If *how* is set to `M_TRYWAIT`, a failed allocation will result in the caller being put to sleep for a designated `kern.ipc.mbuf_wait` (`sysctl(8)` tunable) number of ticks. A number of other mbuf-related functions and macros have the same argument because they may at some point need to allocate new mbufs.

Programmers should be careful not to confuse the mbuf allocation flag `M_DONTWAIT` with the `malloc(9)` allocation flag, `M_NOWAIT`. They are not the same.

MGETHDR(*mbuf*, *how*, *type*)

Allocate an mbuf and initialize it to contain a packet header and internal data. See `MGET()` for details.

MCLGET(*mbuf*, *how*)

Allocate and attach an mbuf cluster to an mbuf. If the macro fails, the `M_EXT` flag won't be set in the mbuf.

M_PREPEND(*mbuf*, *len*, *how*)

This macro operates on an mbuf chain. It is an optimized wrapper for `m_prepend()` that can make use of possible empty space before data (e.g. left after trimming of a link-layer header). The new chain pointer or `NULL` is in *mbuf* after the call.

M_WRITABLE(*mbuf*)

This macro will evaluate true if the mbuf is not marked `M_RDONLY` and if either the mbuf does not contain external storage or, if it does, then if the reference count of the storage is not greater than 1. The `M_RDONLY` flag can be set in the mbuf's `m_flags`. This can be achieved during setup of the external storage, by passing the `M_RDONLY` bit as a *flags* argument to the `MEXTADD()` macro, or can be directly set in individual mbufs.

The functions are:

m_get(*how*, *type*)

A function version of **MGET**() for non-critical paths.

m_getm(*orig*, *len*, *how*, *type*)

Allocate *len* bytes worth of mbufs and mbuf clusters if necessary and append the resulting allocated chain to the *orig* mbuf chain, if it is non-NULL. If the allocation fails at any point, free whatever was allocated and return NULL. If *orig* is non-NULL, it will not be freed. It is possible to use **m_getm**() to either append *len* bytes to an existing mbuf or mbuf chain (for example, one which may be sitting in a pre-allocated ring) or to simply perform an all-or-nothing mbuf and mbuf cluster allocation.

m_gethdr(*how*, *type*)

A function version of **MGETHDR**() for non-critical paths.

m_getclr(*how*, *type*)

Allocate an mbuf and zero out the data region.

The functions below operate on mbuf chains.

m_freem(*mbuf*)

Free an entire mbuf chain, including any external storage.

m_adj(*mbuf*, *len*)

Trim *len* bytes from the head of an mbuf chain if *len* is positive, from the tail otherwise.

m_prepend(*mbuf*, *len*, *how*)

Allocate a new mbuf and prepend it to the chain, handle **M_PKTHDR** properly. **Note:** It doesn't allocate any clusters, so *len* must be less than **MLEN** or **MHLEN**, depending on the **M_PKTHDR** flag setting.

m_pullup(*mbuf*, *len*)

Arrange that the first *len* bytes of an mbuf chain are contiguous and lay in the data area of *mbuf*, so they are accessible with **mtod**(*mbuf*, *type*). Return the new chain on success, NULL on failure (the chain is freed in this case). **Note:** It doesn't allocate any clusters, so *len* must be less than **MHLEN**.

m_copym(*mbuf*, *offset*, *len*, *how*)

Make a copy of an mbuf chain starting *offset* bytes from the beginning, continuing for *len* bytes. If *len* is **M_COPYALL**, copy to the end of the mbuf chain. **Note:** The copy is read-only, because clusters are not copied, only their reference counts are incremented.

m_copypacket(*mbuf*, *how*)

Copy an entire packet including header, which must be present. This is an optimized version of the common case **m_copym**(*mbuf*, 0, **M_COPYALL**, *how*). **Note:** the copy is read-only, because clusters are not copied, only their reference counts are incremented.

m_dup(*mbuf*, *how*)

Copy a packet header mbuf chain into a completely new chain, including copying any mbuf clusters. Use this instead of **m_copypacket**() when you need a writable copy of an mbuf chain.

m_copydata(*mbuf*, *offset*, *len*, *buf*)

Copy data from an mbuf chain starting *off* bytes from the beginning, continuing for *len* bytes, into the indicated buffer *buf*.

m_copyback(*mbuf*, *offset*, *len*, *buf*)

Copy *len* bytes from the buffer *buf* back into the indicated mbuf chain, starting at *offset* bytes from the beginning of the chain, extending the mbuf chain if necessary. **Note:** It doesn't allocate any clusters, just adds mbufs to the chain. It's safe to set *offset* beyond the current chain end: zeroed

mbufs will be allocated to fill the space.

m_length(*buf*, *last*)

Return the length of the mbuf chain, and optionally a pointer to the last mbuf.

m_fixhdr(*buf*)

Set the packet-header length to the length of the mbuf chain.

m_devget(*buf*, *len*, *offset*, *ifp*, *copy*)

Copy data from a device local memory pointed to by *buf* to an mbuf chain. The copy is done using a specified copy routine *copy*, or **bcopy**() if *copy* is NULL.

m_cat(*m*, *n*)

Concatenate *n* to *m*. Both chains must be of the same type. *N* is still valid after the function returned.

Note: It does not handle M_PKTHDR and friends.

m_split(*mbuf*, *len*, *how*)

Partition an mbuf chain in two pieces, returning the tail: all but the first *len* bytes. In case of failure, it returns NULL and attempts to restore the chain to its original state.

STRESS TESTING

When running a kernel compiled with the option `MBUF_STRESS_TEST`, the following `sysctl(8)`-controlled options may be used to create various failure/extreme cases for testing of network drivers and other mbuf-reliant parts of the kernel.

net.inet.ip.mbuf_frag_size

Causes **ip_output**() to fragment outgoing mbuf chains into fragments of the specified size. Setting this variable to 1 is an excellent way to test the long mbuf chain handling ability of network drivers.

kern.ipc.m_defragrandomfailures

Causes the function **m_defrag**() to randomly fail, returning NULL. Any piece of code which uses **m_defrag**() should be tested with this feature.

RETURN VALUES

See above.

HISTORY

Mbufs appeared in an early version of BSD. Besides for being used for network packets, they were used to store various dynamic structures, such as routing table entries, interface addresses, protocol control blocks, etc.

AUTHORS

The original **mbuf** man page was written by Yar Tikhyy.