NAME

LIST_ENTRY, LIST_HEAD, LIST_INIT, LIST_INSERT_AFTER, LIST_INSERT_HEAD, LIST_REMOVE, TAILQ_ENTRY, TAILQ_HEAD, TAILQ_INIT, TAILQ_INSERT_AFTER, TAILQ_INSERT_HEAD, TAILQ_INSERT_TAIL, TAILQ_REMOVE, CIRCLEQ_ENTRY, CIRCLEQ_HEAD, CIRCLEQ_INIT, CIRCLEQ_INSERT_AFTER, CIRCLEQ_INSERT_BEFORE, CIRCLEQ_INSERT_HEAD, CIRCLEQ_INSERT_TAIL, CIRCLEQ_REMOVE - implementations of lists, tail queues, and circular queues

SYNOPSIS

#include <sys/queue.h>

LIST_ENTRY(*TYPE***);**
LIST_HEAD(*HEADNAME***, *TYPE***);**
LIST_INIT(LIST_HEAD ***head*);**
LIST_INSERT_AFTER(LIST_ENTRY ***listelm*,**
TYPE ***elm*, LIST_ENTRY *NAME*);**
LIST_INSERT_HEAD(LIST_HEAD ***head*,**
TYPE ***elm*, LIST_ENTRY *NAME*);**
LIST_REMOVE(TYPE ***elm*, LIST_ENTRY *NAME*);**

TAILQ_ENTRY(*TYPE***);**
TAILQ_HEAD(*HEADNAME***, *TYPE***);**
TAILQ_INIT(TAILQ_HEAD ***head*);**
TAILQ_INSERT_AFTER(TAILQ_HEAD ***head*, TYPE ***listelm***,**
TYPE ***elm*, TAILQ_ENTRY *NAME*);**
TAILQ_INSERT_HEAD(TAILQ_HEAD ***head*,**
TYPE ***elm*, TAILQ_ENTRY *NAME*);**
TAILQ_INSERT_TAIL(TAILQ_HEAD ***head*,**
TYPE ***elm*, TAILQ_ENTRY *NAME*);**
TAILQ_REMOVE(TAILQ_HEAD ***head*, TYPE ***elm***, TAILQ_ENTRY *NAME*);**

CIRCLEQ_ENTRY(*TYPE***);**
CIRCLEQ_HEAD(*HEADNAME***, *TYPE***);**
CIRCLEQ_INIT(CIRCLEQ_HEAD ***head*);**
CIRCLEQ_INSERT_AFTER(CIRCLEQ_HEAD ***head*, TYPE ***listelm***,**
TYPE ***elm*, CIRCLEQ_ENTRY *NAME*);**
CIRCLEQ_INSERT_BEFORE(CIRCLEQ_HEAD ***head*, TYPE ***listelm***,**
TYPE ***elm*, CIRCLEQ_ENTRY *NAME*);**
CIRCLEQ_INSERT_HEAD(CIRCLEQ_HEAD ***head*,**
TYPE ***elm*, CIRCLEQ_ENTRY *NAME*);**
CIRCLEQ_INSERT_TAIL(CIRCLEQ_HEAD ***head*,**
TYPE ***elm*, CIRCLEQ_ENTRY *NAME*);**
CIRCLEQ_REMOVE(CIRCLEQ_HEAD ***head*,**
TYPE ***elm*, CIRCLEQ_ENTRY *NAME*);**

DESCRIPTION

These macros define and operate on three types of data structures: lists, tail queues, and circular queues. All three structures support the following functionality:

* Insertion of a new entry at the head of the list.
* Insertion of a new entry after any element in the list.
* Removal of any entry in the list.
* Forward traversal through the list.

Lists are the simplest of the three data structures and support only the above functionality.

Tail queues add the following functionality:

* Entries can be added at the end of a list.

However:

1. All list insertions and removals must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. Code size is about 15% greater and operations run about 20% slower than lists.

Circular queues add the following functionality:

* Entries can be added at the end of a list.
* Entries can be added before another entry.
* They may be traversed backward, from tail to head.

However:

1. All list insertions and removals must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. The termination condition for traversal is more complex.
4. Code size is about 40% greater and operations run about 45% slower than lists.

In the macro definitions, *TYPE* is the name of a user-defined structure, that must contain a field of type **LIST_ENTRY**, **TAILQ_ENTRY**, or **CIRCLEQ_ENTRY**, named *NAME*. The argument *HEADNAME* is the name of a user-defined structure that must be declared using the macros **LIST_HEAD**, **TAILQ_HEAD**, or **CIRCLEQ_HEAD**. See the examples below for further explanation of how these macros are used.

Lists

A list is headed by a structure defined by the **LIST_HEAD** macro. This structure contains a single pointer to the first element on the list. The elements are doubly linked so that an arbitrary element can be removed without traversing the list. New elements can be added to the list after an existing element or at the head of the list. A **LIST_HEAD** structure is declared as follows:

```
LIST_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

The macro **LIST_ENTRY** declares a structure that connects the elements in the list.

The macro **LIST_INIT** initializes the list referenced by *head*.

The macro **LIST_INSERT_HEAD** inserts the new element *elm* at the head of the list.

The macro **LIST_INSERT_AFTER** inserts the new element *elm* after the element *listelm*.

The macro **LIST_REMOVE** removes the element *elm* from the list.

List example

```
LIST_HEAD(listhead, entry) head;
struct listhead *headp; /* List head. */
struct entry {
LIST_ENTRY(entry) entries; /* List. */
} *n1, *n2, *np;

LIST_INIT(&head); /* Initialize the list. */

n1 = malloc(sizeof(struct entry)); /* Insert at the head. */
LIST_INSERT_HEAD(&head, n1, entries);

n2 = malloc(sizeof(struct entry)); /* Insert after. */
LIST_INSERT_AFTER(n1, n2, entries);
```

```
/* Forward traversal. */
for (np = head.lh_first; np != NULL; np = np->entries.le_next)
np-> ...

while (head.lh_first != NULL) /* Delete. */
LIST_REMOVE(head.lh_first, entries);
```

Tail queues

A tail queue is headed by a structure defined by the **TAILQ_HEAD** macro. This structure contains a pair of pointers, one to the first element in the tail queue and the other to the last element in the tail queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the tail queue. New elements can be added to the tail queue after an existing element, at the head of the tail queue, or at the end of the tail queue. A **TAILQ_HEAD** structure is declared as follows:

```
TAILQ_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the tail queue. A pointer to the head of the tail queue can later be declared as:

```
struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

The macro **TAILQ_ENTRY** declares a structure that connects the elements in the tail queue.

The macro **TAILQ_INIT** initializes the tail queue referenced by *head*.

The macro **TAILQ_INSERT_HEAD** inserts the new element *elm* at the head of the tail queue.

The macro **TAILQ_INSERT_TAIL** inserts the new element *elm* at the end of the tail queue.

The macro **TAILQ_INSERT_AFTER** inserts the new element *elm* after the element *listelm*.

The macro **TAILQ_REMOVE** removes the element *elm* from the tail queue.

Tail queue example

```
TAILQ_HEAD(tailhead, entry) head;
struct tailhead *headp; /* Tail queue head. */
struct entry {
TAILQ_ENTRY(entry) entries; /* Tail queue. */
} *n1, *n2, *np;

TAILQ_INIT(&head); /* Initialize the queue. */

n1 = malloc(sizeof(struct entry)); /* Insert at the head. */
TAILQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry)); /* Insert at the tail. */
TAILQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry)); /* Insert after. */
TAILQ_INSERT_AFTER(&head, n1, n2, entries);
/* Forward traversal. */
for (np = head.tqh_first; np != NULL; np = np->entries.tqe_next)
np-> ...
/* Delete. */
while (head.tqh_first != NULL)
TAILQ_REMOVE(&head, head.tqh_first, entries);
```

Circular queues

A circular queue is headed by a structure defined by the **CIRCLEQ_HEAD** macro. This structure contains a pair of pointers, one to the first element in the circular queue and the other to the last element in the circular queue. The elements are doubly linked so that an arbitrary element

can be removed without traversing the queue. New elements can be added to the queue after an existing element, before an existing element, at the head of the queue, or at the end of the queue. A **CIRCLEQ_HEAD** structure is declared as follows:

```
CIRCLEQ_HEAD(HEADNAME, TYPE) head;
```

where *HEADNAME* is the name of the structure to be defined, and *TYPE* is the type of the elements to be linked into the circular queue. A pointer to the head of the circular queue can later be declared as:

```
struct HEADNAME *headp;
```

(The names *head* and *headp* are user selectable.)

The macro **CIRCLEQ_ENTRY** declares a structure that connects the elements in the circular queue.

The macro **CIRCLEQ_INIT** initializes the circular queue referenced by *head*.

The macro **CIRCLEQ_INSERT_HEAD** inserts the new element *elm* at the head of the circular queue.

The macro **CIRCLEQ_INSERT_TAIL** inserts the new element *elm* at the end of the circular queue.

The macro **CIRCLEQ_INSERT_AFTER** inserts the new element *elm* after the element *listelm*.

The macro **CIRCLEQ_INSERT_BEFORE** inserts the new element *elm* before the element *listelm*.

The macro **CIRCLEQ_REMOVE** removes the element *elm* from the circular queue.

Circular queue example

```
CIRCLEQ_HEAD(circleq, entry) head;
struct circleq *headp; /* Circular queue head. */
struct entry {
CIRCLEQ_ENTRY(entry) entries; /* Circular queue. */
} *n1, *n2, *np;

CIRCLEQ_INIT(&head); /* Initialize the circular queue. */

n1 = malloc(sizeof(struct entry)); /* Insert at the head. */
CIRCLEQ_INSERT_HEAD(&head, n1, entries);

n1 = malloc(sizeof(struct entry)); /* Insert at the tail. */
CIRCLEQ_INSERT_TAIL(&head, n1, entries);

n2 = malloc(sizeof(struct entry)); /* Insert after. */
CIRCLEQ_INSERT_AFTER(&head, n1, n2, entries);

n2 = malloc(sizeof(struct entry)); /* Insert before. */
CIRCLEQ_INSERT_BEFORE(&head, n1, n2, entries);
/* Forward traversal. */
for (np = head.cqh_first; np != (void *)&head;
np = np->entries.cqe_next)
np-> ...
/* Reverse traversal. */
for (np = head.cqh_last; np != (void *)&head; np = np->entries.cqe_prev)
np-> ...
/* Delete. */
while (head.cqh_first != (void *)&head)
CIRCLEQ_REMOVE(&head, head.cqh_first, entries);
```

CONFORMING TO

Not in POSIX.1-2001. Present on the BSDs. The queue functions first appeared in 4.4BSD.

COLOPHON

This page is part of release 3.74 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at http://www.kernel.org/doc/man-pages/.