Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'LIST_INSERT_HEAD.3' command

$ man LIST_INSERT_HEAD.3

LIST(3) Linux Programmer's Manual LIST(3)

NAME

LIST_EMPTY, LIST_ENTRY, LIST_FIRST, LIST_FOREACH, LIST_HEAD, LIST_HEAD_INITIALIZER, LIST_INIT, LIST_INSERT_AFTER, LIST_INSERT_BE? FORE, LIST_INSERT_HEAD, LIST_NEXT, LIST_REMOVE - implementation of a doubly linked list

SYNOPSIS

```
#include <sys/queue.h>
int LIST_EMPTY(LIST_HEAD *head);
LIST_ENTRY(TYPE);
struct TYPE *LIST_FIRST(LIST_HEAD *head);
LIST_FOREACH(struct TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME);
LIST_HEAD(HEADNAME, TYPE);
LIST_HEAD LIST_HEAD_INITIALIZER(LIST_HEAD head);
void LIST_INIT(LIST_HEAD *head);
void LIST_INSERT_AFTER(struct TYPE *listelm, struct TYPE *elm,
        LIST_ENTRY NAME);
void LIST_INSERT_BEFORE(struct TYPE *listelm, struct TYPE *elm,
        LIST_ENTRY NAME);
void LIST_INSERT_HEAD(LIST_HEAD *head, struct TYPE *elm,
        LIST_ENTRY NAME);
struct TYPE *LIST_NEXT(struct TYPE *elm, LIST_ENTRY NAME);
void LIST_REMOVE(struct TYPE *elm, LIST_ENTRY NAME);
```

DESCRIPTION

These macros define and operate on doubly linked lists.

In the macro definitions, TYPE is the name of a user-defined structure, that must contain a field of type LIST_ENTRY, named NAME. The argument HEADNAME is the name of a user-defined structure that must be declared using the macro LIST_HEAD().

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 re? moved without traversing the list. New elements can be added to the list after an existing element, before an existing element, or at the head of the list. A LIST_HEAD structure is declared as follows:

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

where struct HEADNAME is the structure to be defined, and struct 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_HEAD_INITIALIZER() evaluates to an initializer for the list head.

The macro LIST_EMPTY() evaluates to true if there are no elements in the list.

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

The macro LIST_FIRST() returns the first element in the list or NULL if the list is empty.

The macro LIST_FOREACH() traverses the list referenced by head in the forward direction, assigning each element in turn to var.

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 el? ement listelm.

The macro LIST_INSERT_BEFORE() inserts the new element elm before the element listelm.

The macro LIST_NEXT() returns the next element in the list, or NULL if this is the last.

The macro LIST_REMOVE() removes the element elm from the list.

RETURN VALUE

LIST_EMPTY() returns nonzero if the list is empty, and zero if the list contains at least one entry.

LIST_FIRST(), and LIST_NEXT() return a pointer to the first or next TYPE structure, respectively.

LIST_HEAD_INITIALIZER() returns an initializer that can be assigned to the list head.

CONFORMING TO

Not in POSIX.1, POSIX.1-2001 or POSIX.1-2008. Present on the BSDs (LIST macros first appeared in 4.4BSD).

BUGS

The macro LIST_FOREACH() doesn't allow var to be removed or freed within the loop, as it would interfere with the traversal. The macro LIST_FOREACH_SAFE(), which is present on the BSDs but is not present in glibc, fixes this limitation by allowing var to safely be removed from the list and freed from within the loop without interfering with the traversal.

EXAMPLES

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/queue.h>
struct entry {
    int data;
    LIST_ENTRY(entry) entries;          /* List. */
};
LIST_HEAD(listhead, entry);
int
```

```
main(void)
{
    struct entry *n1, *n2, *n3, *np;
    struct listhead head;                   /* List head. */
    int i;
    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);
    n3 = malloc(sizeof(struct entry));      /* Insert before. */
    LIST_INSERT_BEFORE(n2, n3, entries);
    i = 0;                                  /* Forward traversal. */
    LIST_FOREACH(np, &head, entries)
        np->data = i++;
    LIST_REMOVE(n2, entries);               /* Deletion. */
    free(n2);
                                            /* Forward traversal. */
    LIST_FOREACH(np, &head, entries)
        printf("%i\n", np->data);
                                            /* List Deletion. */
    n1 = LIST_FIRST(&head);
    while (n1 != NULL) {
        n2 = LIST_NEXT(n1, entries);
        free(n1);
        n1 = n2;
    }
    LIST_INIT(&head);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

insque(3), queue(7)

COLOPHON