The Linux Kernel API

Table of Contents

clk_get_rate — obtain the current clock rate (in Hz) for a clock source. This is only valid once the clock source has been enabled.
clk_put — "free" the clock source
clk_round_rate — adjust a rate to the exact rate a clock can provide
clk_set_rate — set the clock rate for a clock source
clk_set_parent — set the parent clock source for this clock
clk_get_parent — get the parent clock source for this clock
clk_get_sys — get a clock based upon the device name
clk_add_alias — add a new clock alias

Chapter 1. Data Types

Table of Contents

Doubly Linked Lists

Name

list_add — add a new entry

Synopsis

```
void list_add (new,
               head);

struct list_head *  new;
struct list_head *  head;
```

Arguments

new

new entry to be added

head

list head to add it after

Description

Insert a new entry after the specified head. This is good for implementing stacks.

Name

list_add_tail — add a new entry

Synopsis

```
void list_add_tail (new,
                    head);

struct list_head *  new;
struct list_head *  head;
```

Arguments

new

new entry to be added

head

list head to add it before

Description

Insert a new entry before the specified head. This is useful for implementing queues.

Name

list_del — deletes entry from list.

Synopsis

```
void list_del (entry);

struct list_head *  entry;
```

Arguments

entry

the element to delete from the list.

Note

`list_empty` on entry does not return true after this, the entry is in an undefined state.

Name

list_replace — replace old entry by new one

Synopsis

```
void list_replace (old,
                   new);

struct list_head *  old;
struct list_head *  new;
```

Arguments

old

the element to be replaced

new

the new element to insert

Description

If *old* was empty, it will be overwritten.

Name

list_del_init — deletes entry from list and reinitialize it.

Synopsis

```
void list_del_init (entry);

struct list_head *  entry;
```

Arguments

entry

the element to delete from the list.

Name

list_move — delete from one list and add as another's head

Synopsis

```
void list_move (list,
                head);

struct list_head *  list;
struct list_head *  head;
```

Arguments

list

the entry to move

head

the head that will precede our entry

Name

list_move_tail — delete from one list and add as another's tail

Synopsis

```
void list_move_tail (list,
                     head);

struct list_head *  list;
struct list_head *  head;
```

Arguments

list

the entry to move

head

the head that will follow our entry

Name

list_is_last — tests whether *list* is the last entry in list *head*

Synopsis

```
int list_is_last (list,
                  head);

const struct list_head *  list;
const struct list_head *  head;
```

Arguments

list

the entry to test

head

the head of the list

Name

list_empty — tests whether a list is empty

Synopsis

```
int list_empty (head);

const struct list_head *  head;
```

Arguments

head

the list to test.

Name

list_empty_careful — tests whether a list is empty and not being modified

Synopsis

```
int list_empty_careful (head);

const struct list_head *  head;
```

Arguments

head

the list to test

Description

tests whether a list is empty _and_ checks that no other CPU might be in the process of modifying either member (next or prev)

NOTE

using `list_empty_careful` without synchronization can only be safe if the only activity that can happen to the list entry is `list_del_init`. Eg. it cannot be used if another CPU could re-`list_add` it.

Name

list_is_singular — tests whether a list has just one entry.

Synopsis

```
int list_is_singular (head);

const struct list_head *  head;
```

Arguments

head

the list to test.

Name

list_cut_position — cut a list into two

Synopsis

```
void list_cut_position (list,
                        head,
                        entry);

struct list_head *  list;
struct list_head *  head;
struct list_head *  entry;
```

Arguments

list

a new list to add all removed entries

head

a list with entries

entry

an entry within head, could be the head itself and if so we won't cut the list

Description

This helper moves the initial part of *head*, up to and including *entry*, from *head* to *list*. You should pass on *entry* an element you know is on *head*. *list* should be an empty list or a list you do not care about losing its data.

Name

list_splice — join two lists, this is designed for stacks

Synopsis

```
void list_splice (list,
                  head);

const struct list_head *  list;
struct list_head *        head;
```

Arguments

list

the new list to add.

head

the place to add it in the first list.

Name

list_splice_tail — join two lists, each list being a queue

Synopsis

```
void list_splice_tail (list,
                       head);

struct list_head * list;
struct list_head * head;
```

Arguments

list

the new list to add.

head

the place to add it in the first list.

Name

list_splice_init — join two lists and reinitialise the emptied list.

Synopsis

```
void list_splice_init (list,
                       head);

struct list_head * list;
struct list_head * head;
```

Arguments

list

the new list to add.

head

the place to add it in the first list.

Description

The list at *list* is reinitialised

Name

list_splice_tail_init — join two lists and reinitialise the emptied list

Synopsis

```
void list_splice_tail_init (list,
                            head);

struct list_head *  list;
struct list_head *  head;
```

Arguments

list

the new list to add.

head

the place to add it in the first list.

Description

Each of the lists is a queue. The list at *list* is reinitialised

---

Name

list_entry — get the struct for this entry

Synopsis

```
list_entry (ptr,
            type,
            member);

 ptr;
 type;
 member;
```

Arguments

ptr

the struct list_head pointer.

type

the type of the struct this is embedded in.

member

the name of the list_struct within the struct.

Name

list_first_entry — get the first element from a list

Synopsis

```
list_first_entry (ptr,
                  type,
                  member);

ptr;
type;
member;
```

Arguments

ptr

the list head to take the element from.

type

the type of the struct this is embedded in.

member

the name of the list_struct within the struct.

Description

Note, that list is expected to be not empty.

Name

list_for_each — iterate over a list

Synopsis

```
list_for_each (pos,
               head);

pos;
```

head;

Arguments

pos

the struct list_head to use as a loop cursor.

head

the head for your list.

Name

__list_for_each — iterate over a list

Synopsis

__list_for_each (*pos*,
head);

pos;
head;

Arguments

pos

the struct list_head to use as a loop cursor.

head

the head for your list.

Description

This variant differs from `list_for_each` in that it's the simplest possible list iteration code, no prefetching is done. Use this for code that knows the list to be very short (empty or 1 entry) most of the time.

Name

list_for_each_prev — iterate over a list backwards

Synopsis

```
list_for_each_prev (pos,
                    head);

pos;
head;
```

Arguments

pos

the struct list_head to use as a loop cursor.

head

the head for your list.

Name

list_for_each_safe — iterate over a list safe against removal of list entry

Synopsis

```
list_for_each_safe (pos,
                    n,
                    head);

pos;
n;
head;
```

Arguments

pos

the struct list_head to use as a loop cursor.

n

another struct list_head to use as temporary storage

head

the head for your list.

Name

list_for_each_prev_safe — iterate over a list backwards safe against removal of list entry

Synopsis

```
list_for_each_prev_safe (pos,
                         n,
                         head);

pos;
n;
head;
```

Arguments

pos

the struct list_head to use as a loop cursor.

n

another struct list_head to use as temporary storage

head

the head for your list.

Name

list_for_each_entry — iterate over list of given type

Synopsis

```
list_for_each_entry (pos,
                     head,
                     member);

pos;
head;
member;
```

Arguments

pos

the type * to use as a loop cursor.

head

the head for your list.

member

the name of the list_struct within the struct.

Name

list_for_each_entry_reverse — iterate backwards over list of given type.

Synopsis

```
list_for_each_entry_reverse (pos,
                             head,
                             member);

pos;
head;
member;
```

Arguments

pos

the type * to use as a loop cursor.

head

the head for your list.

member

the name of the list_struct within the struct.

Name

list_prepare_entry — prepare a pos entry for use in `list_for_each_entry_continue`

Synopsis

```
list_prepare_entry (pos,
                    head,
                    member);

pos;
head;
member;
```

Arguments

pos

the type * to use as a start point

head

the head of the list

member

the name of the list_struct within the struct.

Description

Prepares a pos entry for use as a start point in `list_for_each_entry_continue`.

Name

list_for_each_entry_continue — continue iteration over list of given type

Synopsis

```
list_for_each_entry_continue (pos,
                              head,
                              member);

pos;
head;
member;
```

Arguments

pos

the type * to use as a loop cursor.

head

the head for your list.

member

the name of the list_struct within the struct.

Description

Continue to iterate over list of given type, continuing after the current position.

Name

list_for_each_entry_continue_reverse — iterate backwards from the given point

Synopsis

```
list_for_each_entry_continue_reverse (pos,
                                      head,
                                      member);

pos;
head;
member;
```

Arguments

pos

the type * to use as a loop cursor.

head

the head for your list.

member

the name of the list_struct within the struct.

Description

Start to iterate over list of given type backwards, continuing after the current position.

Name

list_for_each_entry_from — iterate over list of given type from the current point

Synopsis

```
list_for_each_entry_from (pos,
                          head,
                          member);

pos;
head;
```

member;

Arguments

pos

the type * to use as a loop cursor.

head

the head for your list.

member

the name of the list_struct within the struct.

Description

Iterate over list of given type, continuing from current position.

Name

list_for_each_entry_safe — iterate over list of given type safe against removal of list entry

Synopsis

```
list_for_each_entry_safe (pos,
                          n,
                          head,
                          member);
```

pos;
n;
head;
member;

Arguments

pos

the type * to use as a loop cursor.

n

another type * to use as temporary storage

head

the head for your list.

member

the name of the list_struct within the struct.

Name

list_for_each_entry_safe_continue —

Synopsis

```
list_for_each_entry_safe_continue (pos,
                                   n,
                                   head,
                                   member);

pos;
n;
head;
member;
```

Arguments

pos

the type * to use as a loop cursor.

n

another type * to use as temporary storage

head

the head for your list.

member

the name of the list_struct within the struct.

Description

Iterate over list of given type, continuing after current point, safe against removal of list entry.

Name

list_for_each_entry_safe_from —

Synopsis

```
list_for_each_entry_safe_from (pos,
                               n,
                               head,
                               member);

pos;
n;
head;
member;
```

Arguments

pos

the type * to use as a loop cursor.

n

another type * to use as temporary storage

head

the head for your list.

member

the name of the list_struct within the struct.

Description

Iterate over list of given type from current point, safe against removal of list entry.

Name

list_for_each_entry_safe_reverse —

Synopsis

```
list_for_each_entry_safe_reverse (pos,
                                  n,
                                  head,
                                  member);

pos;
n;
head;
```

member;

Arguments

pos

the type * to use as a loop cursor.

n

another type * to use as temporary storage

head

the head for your list.

member

the name of the list_struct within the struct.

Description

Iterate backwards over list of given type, safe against removal of list entry.

Name

hlist_for_each_entry — iterate over list of given type

Synopsis

```
hlist_for_each_entry (tpos,
                      pos,
                      head,
                      member);
```

tpos;
pos;
head;
member;

Arguments

tpos

the type * to use as a loop cursor.

pos

the struct hlist_node to use as a loop cursor.

head

the head for your list.

member

the name of the hlist_node within the struct.

Name

hlist_for_each_entry_continue — iterate over a hlist continuing after current point

Synopsis

```
hlist_for_each_entry_continue (tpos,
                               pos,
                               member);

tpos;
pos;
member;
```

Arguments

tpos

the type * to use as a loop cursor.

pos

the struct hlist_node to use as a loop cursor.

member

the name of the hlist_node within the struct.

Name

hlist_for_each_entry_from — iterate over a hlist continuing from current point

Synopsis

```
hlist_for_each_entry_from (tpos,
                           pos,
                           member);
```

tpos;
pos;
member;

Arguments

tpos

the type * to use as a loop cursor.

pos

the struct hlist_node to use as a loop cursor.

member

the name of the hlist_node within the struct.

Name

hlist_for_each_entry_safe — iterate over list of given type safe against removal of list entry

Synopsis

```
hlist_for_each_entry_safe (tpos,
                           pos,
                           n,
                           head,
                           member);
```

tpos;
pos;
n;
head;
member;

Arguments

tpos

the type * to use as a loop cursor.

pos

the struct hlist_node to use as a loop cursor.

n

another struct hlist_node to use as temporary storage

head

the head for your list.

member

the name of the hlist_node within the struct.

Chapter 2. Basic C Library Functions

Table of Contents

When writing drivers, you cannot in general use routines which are from the C Library. Some of the functions have been found generally useful and they are listed below. The behaviour of these functions may vary slightly from those defined by ANSI, and these deviations are noted in the text.

String Conversions

Name

simple_strtoll — convert a string to a signed long long

Synopsis

```
long long simple_strtoll (cp,
                          endp,
                          base);

const char *  cp;
char **       endp;
unsigned int  base;
```

Arguments

cp

The start of the string

endp

A pointer to the end of the parsed string will be placed here

base

The number base to use

Name

simple_strtoul — convert a string to an unsigned long

Synopsis

```
unsigned long simple_strtoul (cp,
                              endp,
                              base);

const char *  cp;
char **       endp;
unsigned int  base;
```

Arguments

cp

The start of the string

endp

A pointer to the end of the parsed string will be placed here

base

The number base to use

Name

simple_strtol — convert a string to a signed long

Synopsis

```
long simple_strtol (cp,
                    endp,
                    base);

const char *  cp;
char **       endp;
unsigned int  base;
```

Arguments

cp

The start of the string

endp

A pointer to the end of the parsed string will be placed here

base

The number base to use

Name

simple_strtoull — convert a string to an unsigned long long

Synopsis

```
unsigned long long simple_strtoull (cp,
                                    endp,
                                    base);

const char *  cp;
char **       endp;
unsigned int  base;
```

Arguments

cp

The start of the string

endp

A pointer to the end of the parsed string will be placed here

base

The number base to use

Name

strict_strtoul — convert a string to an unsigned long strictly

Synopsis

```
int strict_strtoul (cp,
                    base,
                    res);
```

```
const char *     cp;
unsigned int     base;
unsigned long *  res;
```

Arguments

cp

The string to be converted

base

The number base to use

res

The converted result value

Description

strict_strtoul converts a string to an unsigned long only if the string is really an unsigned long string, any string containing any invalid char at the tail will be rejected and -EINVAL is returned, only a newline char at the tail is acceptible because people generally

change a module parameter in the following way

echo 1024 > /sys/module/e1000/parameters/copybreak

echo will append a newline to the tail.

It returns 0 if conversion is successful and *res is set to the converted value, otherwise it returns -EINVAL and *res is set to 0.

simple_strtoul just ignores the successive invalid characters and return the converted value of prefix part of the string.

Name

strict_strtol — convert a string to a long strictly

Synopsis

```
int strict_strtol (cp,
                   base,
                   res);

const char *  cp;
unsigned int  base;
```

```
long *        res;
```

Arguments

cp

The string to be converted

base

The number base to use

res

The converted result value

Description

strict_strtol is similiar to strict_strtoul, but it allows the first character of a string is '-'.

It returns 0 if conversion is successful and *res is set to the converted value, otherwise it returns -EINVAL and *res is set to 0.

Name

strict_strtoull — convert a string to an unsigned long long strictly

Synopsis

```
int strict_strtoull (cp,
                     base,
                     res);

const char *          cp;
unsigned int          base;
unsigned long long *  res;
```

Arguments

cp

The string to be converted

base

The number base to use

res

The converted result value

Description

strict_strtoull converts a string to an unsigned long long only if the string is really an unsigned long long string, any string containing any invalid char at the tail will be rejected and -EINVAL is returned, only a newline char at the tail is acceptible because people generally

change a module parameter in the following way

echo 1024 > /sys/module/e1000/parameters/copybreak

echo will append a newline to the tail of the string.

It returns 0 if conversion is successful and *res is set to the converted value, otherwise it returns -EINVAL and *res is set to 0.

simple_strtoull just ignores the successive invalid characters and return the converted value of prefix part of the string.

Name

strict_strtoll — convert a string to a long long strictly

Synopsis

```
int strict_strtoll (cp,
                    base,
                    res);

const char *  cp;
unsigned int  base;
long long *   res;
```

Arguments

cp

The string to be converted

base

The number base to use

res

The converted result value

Description

strict_strtoll is similiar to strict_strtoull, but it allows the first character of a string is '-'.

It returns 0 if conversion is successful and *res is set to the converted value, otherwise it returns -EINVAL and *res is set to 0.

Name

vsnprintf — Format a string and place it in a buffer

Synopsis

```
int vsnprintf (buf,
               size,
               fmt,
               args);

char *        buf;
size_t        size;
const char *  fmt;
va_list       args;
```

Arguments

buf

The buffer to place the result into

size

The size of the buffer, including the trailing null space

fmt

The format string to use

args

Arguments for the format string

Description

This function follows C99 vsnprintf, but has some extensions: `pS` output the name of a text symbol with offset `ps` output the name of a text symbol without offset `pF` output the name of a function pointer with its offset `pf` output the name of a function pointer without its offset `pR` output the address range in a struct resource `n` is ignored

The return value is the number of characters which would be generated for the given input, excluding the trailing '\0', as per ISO C99. If you want to have the exact number of characters written into *buf* as return value (not including the trailing '\0'), use vscnprintf. If the return is greater than or equal to *size*, the resulting string is truncated.

Call this function if you are already dealing with a va_list. You probably want snprintf instead.

Name

vscnprintf — Format a string and place it in a buffer

Synopsis

```
int vscnprintf (buf,
                size,
                fmt,
                args);

char *        buf;
size_t        size;
const char *  fmt;
va_list       args;
```

Arguments

buf

The buffer to place the result into

size

The size of the buffer, including the trailing null space

fmt

The format string to use

args

Arguments for the format string

Description

The return value is the number of characters which have been written into the *buf* not including the trailing '\0'. If *size* is <= 0 the function returns 0.

Call this function if you are already dealing with a va_list. You probably want scnprintf instead.

See the vsnprintf documentation for format string extensions over C99.

Name

snprintf — Format a string and place it in a buffer

Synopsis

```
int snprintf (buf,
              size,
              fmt,
              ...);

char *        buf;
size_t        size;
const char *  fmt;
              ...;
```

Arguments

buf

The buffer to place the result into

size

The size of the buffer, including the trailing null space

fmt

The format string to use @...: Arguments for the format string

...

variable arguments

Description

The return value is the number of characters which would be generated for the given input, excluding the trailing null, as per ISO C99. If the return is greater than or equal to *size*, the resulting string is truncated.

See the `vsnprintf` documentation for format string extensions over C99.

Name

scnprintf — Format a string and place it in a buffer

Synopsis

```
int scnprintf (buf,
               size,
               fmt,
               ...);

char *        buf;
size_t        size;
const char *  fmt;
              ...;
```

Arguments

buf

The buffer to place the result into

size

The size of the buffer, including the trailing null space

fmt

The format string to use @...: Arguments for the format string

...

variable arguments

Description

The return value is the number of characters written into *buf* not including the trailing '\0'. If *size* is <= 0 the function returns 0.

Name

vsprintf — Format a string and place it in a buffer

Synopsis

```
int vsprintf (buf,
              fmt,
              args);

char *        buf;
const char *  fmt;
va_list       args;
```

Arguments

buf

The buffer to place the result into

fmt

The format string to use

args

Arguments for the format string

Description

The function returns the number of characters written into *buf*. Use vsnprintf or vscnprintf in order to avoid buffer overflows.

Call this function if you are already dealing with a va_list. You probably want sprintf instead.

See the vsnprintf documentation for format string extensions over C99.

Name

sprintf — Format a string and place it in a buffer

Synopsis

```
int sprintf (buf,
             fmt,
             ...);

char *        buf;
const char *  fmt;
              ...;
```

Arguments

buf

The buffer to place the result into

fmt

The format string to use @...: Arguments for the format string

...

variable arguments

Description

The function returns the number of characters written into *buf*. Use snprintf or scnprintf in order to avoid buffer overflows.

See the vsnprintf documentation for format string extensions over C99.

Name

vbin_printf — Parse a format string and place args' binary value in a buffer

Synopsis

```
int vbin_printf (bin_buf,
                 size,
                 fmt,
                 args);

u32 *         bin_buf;
size_t        size;
const char *  fmt;
va_list       args;
```

Arguments

bin_buf

The buffer to place args' binary value

size

The size of the buffer(by words(32bits), not characters)

fmt

The format string to use

args

Arguments for the format string

Description

The format follows C99 vsnprintf, except n is ignored, and its argument is skiped.

The return value is the number of words(32bits) which would be generated for the given input.

NOTE

If the return value is greater than *size*, the resulting bin_buf is NOT valid for bstr_printf.

Name

bstr_printf — Format a string from binary arguments and place it in a buffer

Synopsis

```
int bstr_printf (buf,
                 size,
                 fmt,
                 bin_buf);

char *        buf;
size_t        size;
const char *  fmt;
const u32 *   bin_buf;
```

Arguments

buf

The buffer to place the result into

size

The size of the buffer, including the trailing null space

fmt

The format string to use

bin_buf

Binary arguments for the format string

Description

This function like C99 vsnprintf, but the difference is that vsnprintf gets arguments from stack, and bstr_printf gets arguments from *bin_buf* which is a binary buffer that generated by vbin_printf.

The format follows C99 vsnprintf, but has some extensions: see vsnprintf comment for details.

The return value is the number of characters which would be generated for the given input, excluding the trailing '\0', as per ISO C99. If you want to have the exact number of characters written into *buf* as return

value (not including the trailing '\0'), use `vscnprintf`. If the return is greater than or equal to *`size`*, the resulting string is truncated.

Name

bprintf — Parse a format string and place args' binary value in a buffer

Synopsis

```
int bprintf (bin_buf,
             size,
             fmt,
             ...);

u32 *         bin_buf;
size_t        size;
const char *  fmt;
              ...;
```

Arguments

`bin_buf`

The buffer to place args' binary value

`size`

The size of the buffer(by words(32bits), not characters)

`fmt`

The format string to use @...: Arguments for the format string

`...`

variable arguments

Description

The function returns the number of words(u32) written into *`bin_buf`*.

Name

vsscanf — Unformat a buffer into a list of arguments

Synopsis

```
int vsscanf (buf,
             fmt,
             args);

const char *  buf;
const char *  fmt;
va_list       args;
```

Arguments

buf

input buffer

fmt

format of buffer

args

arguments

Name

sscanf — Unformat a buffer into a list of arguments

Synopsis

```
int sscanf (buf,
            fmt,
            ...);

const char *  buf;
const char *  fmt;
              ...;
```

Arguments

buf

input buffer

fmt

formatting of buffer @...: resulting arguments

...

variable arguments

String Manipulation

Name

strnicmp — Case insensitive, length-limited string comparison

Synopsis

```
int strnicmp (s1,
              s2,
              len);

const char * s1;
const char * s2;
size_t       len;
```

Arguments

s1

One string

s2

The other string

len

the maximum number of characters to compare

Name

strcpy — Copy a NUL terminated string

Synopsis

```
char * strcpy (dest,
               src);

char *       dest;
const char * src;
```

Arguments

dest

Where to copy the string to

src

Where to copy the string from

Name

strncpy — Copy a length-limited, NUL-terminated string

Synopsis

```
char * strncpy (dest,
                src,
                count);

char *        dest;
const char *  src;
size_t        count;
```

Arguments

dest

Where to copy the string to

src

Where to copy the string from

count

The maximum number of bytes to copy

Description

The result is not NUL-terminated if the source exceeds *count* bytes.

In the case where the length of *src* is less than that of count, the remainder of *dest* will be padded with NUL.

Name

strlcpy — Copy a NUL terminated string into a sized buffer

Synopsis

```
size_t strlcpy (dest,
                src,
                size);

char *        dest;
const char *  src;
size_t        size;
```

Arguments

dest

Where to copy the string to

src

Where to copy the string from

size

size of destination buffer

BSD

the result is always a valid NUL-terminated string that fits in the buffer (unless, of course, the buffer size is zero). It does not pad out the result like `strncpy` does.

Name

strcat — Append one `NUL-terminated` string to another

Synopsis

```
char * strcat (dest,
               src);

char *        dest;
const char *  src;
```

Arguments

dest

The string to be appended to

src

The string to append to it

Name

strncat — Append a length-limited, NUL-terminated string to another

Synopsis

```
char * strncat (dest,
                src,
                count);

char *        dest;
const char *  src;
size_t        count;
```

Arguments

dest

The string to be appended to

src

The string to append to it

count

The maximum numbers of bytes to copy

Description

Note that in contrast to strncpy, strncat ensures the result is terminated.

Name

strlcat — Append a length-limited, NUL-terminated string to another

Synopsis

```
size_t strlcat (dest,
                src,
                count);

char *        dest;
const char *  src;
size_t        count;
```

Arguments

dest

The string to be appended to

src

The string to append to it

count

The size of the destination buffer.

Name

strcmp — Compare two strings

Synopsis

```
int strcmp (cs,
            ct);

const char *  cs;
const char *  ct;
```

Arguments

cs

One string

ct

Another string

Name

strncmp — Compare two length-limited strings

Synopsis

```
int strncmp (cs,
             ct,
             count);
```

```
const char *  cs;
const char *  ct;
size_t        count;
```

Arguments

cs

One string

ct

Another string

count

The maximum number of bytes to compare

Name

strchr — Find the first occurrence of a character in a string

Synopsis

```
char * strchr (s,
               c);

const char *  s;
int           c;
```

Arguments

s

The string to be searched

c

The character to search for

Name

strrchr — Find the last occurrence of a character in a string

Synopsis

```
char * strrchr (s,
                c);

const char *  s;
int           c;
```

Arguments

s

The string to be searched

c

The character to search for

Name

strnchr — Find a character in a length limited string

Synopsis

```
char * strnchr (s,
                count,
                c);

const char *  s;
size_t        count;
int           c;
```

Arguments

s

The string to be searched

count

The number of characters to be searched

c

The character to search for

Name

strstrip — Removes leading and trailing whitespace from *s*.

Synopsis

```
char * strstrip (s);

char *  s;
```

Arguments

s

The string to be stripped.

Description

Note that the first trailing whitespace is replaced with a `NUL-terminator` in the given string *s*. Returns a pointer to the first non-whitespace character in *s*.

Name

strlen — Find the length of a string

Synopsis

```
size_t strlen (s);

const char *  s;
```

Arguments

s

The string to be sized

Name