Clib

The Clib module is a set of shared libraries providing the implementations of most of the standard C routines in Appendix B of Kernighan & Ritchie. It excludes floating-point support, which is provided by the relevant machine-dependent library. Routines which expect a filing system (for example *rename*) are implemented in the *FSlib* module.

Module Options

CLIB_CONSOLE_PROCESS	The name (rome URL) of the process to which the standard I/O streams should be connected for processes that use *stdin, stdout* and *stderr*. If this option is not set, the default value of "console" will be used.
CLIB_NDEBUG	Disables the *assert* macro.
CLIB_SYSTEM_PROCESS	The name of a process capable of receiving and responding to commands passed through the *system* routine call.
CPU_HAS_FLOAT	If this option is enabled in the CPU plugin module, support for floating-point conversions is compiled in to the I/O routines.
NVRAM_RTC_ENABLED	Enables the use of the *rtc_read_time* routine to get the current date and time.

Data Definitions

The header files in the module contain the following type definitions:

BITMAP	The data structure representing a resource bitmap.
struct tm	The structure for time-of-day routines.

Shared Library Macros and Routines

Full details of the behaviour of most of these routines is exactly as described in Appendix B of K&R. However K&R leaves some implementation details open (such as whether a library routine is a procedure or a macro). These decisions for ROME are documented here, as are any deviations from strict adherence to the K&R specifications.

Many of the I/O routines make messaging calls into the ROME kernel. The specification for the routines usually requires that these be pended calls (i.e. that the process must block until the reply is received). This limits the use of such routines to places where blocking can occur. Specifically this excludes the process' initialisation and queue-handling routines and any registered interrupt handlers (plus routines that they may call). The restriction to 'process context' in the following descriptions indicates a routine which may cause a context switch.

In order to use the *stdio* routines (like *printf* or *fprintf* with a *stdout* argument) the standard files must be initialised. Unless they are to be directed to a non-standard location, this usually means marking the process with the *use-stdio* flag during the build. In the following list, the restriction of a routine to the 'stdio context' means a main process or its direct descendant routine following initialisation of the standard files. The 'stdio context' implies the 'process context' restriction, but some of the 'process context' routines (for example *fopen*) are not restricted to the 'stdio context'.

The standard library specifications describe different styles of I/O without giving clear indications of how they might be mixed. In particular the action of various routines following a call to *ungetc*, or mixing character-based calls like *fgetc* with record-oriented I/O using *fread* are not specified.

These interactions mostly affect only legacy programs. Applications written specifically for ROME, or those written to handle networked multimedia data are expected to use the 'mblk' interface to avoid data copying and provide efficient protocol transport. The standard library routines should, in general, be limited to operations on *stdout*, *stdin* and *stderr*. The use of *fread* and *fwrite* is not recommended (though the routines are implemented).

_main

void *_main*(
  void (**proc***)(void)**,
  char **process*)

The *_main* routine is the entrypoint for processes which use the C standard I/O library calls such as *printf* and which expect *stdin* etc. to be present. The routine opens the standard streams and the calls the main process *proc* for the process named *process*. The routine also calls the *exit* routine should the process terminate, which further executes any *atexit* routines specified.

abs

int *abs*(
  int *n*)

The *abs* routine returns the absolute value of *n*.

allocb

mblk_t **allocb*(
  int *size,*
  int *pri*)

The *allocb* routine allocates an mblk capable of holding at least *size* bytes, and initialises the internal pointers for the data buffer. The *pri* field is ignored.

asctime

char **asctime*(
  const struct tm **tp*)

The *asctime* routine converts the *tp* structure into a printable string in standard format. Note that as the returned pointer is allocated in a static buffer, the data must be copied if they are to be stored.

assert

> **(void)** *assert*(
> **int** *_cond*)

The *assert* macro prints a message on *stdout* and calls *exit* if the supplied condition *_cond* is false. This macro may only be used in the stdio context. If the *CLIB_NDEBUG* option is set, the *assert* macro does nothing.

atexit

> **int** *atexit*(
> **void** (******funcptr*)(**void**))

The *atexit* routine adds a function *funcptr* to the list of functions to be called when a process exits. This call has an effect only when the process is started through the stdio initialisation preamble or explicitly calls the *exit* routine.

bcopy

> **void** *bcopy*(
> **const void** **from*,
> **void** **to*,
> **int** *len*)

The *bcopy* routines copies *len* bytes from *from* to *to*. Note that the arguments are in a different order from *memcpy*.

bitmap_claim

> **uint** *bitmap_claim*(
> **BITMAP** **map*)

The *bitmap_claim* routine returns a resources index current marked as free in the supplied *map*. If no resources are available the the routine returns *BITMAP_NO_RESOURCE*. It is up to the calling program to map the index into a resource value.

bitmap_free

> **void** *bitmap_free*(
> **BITMAP** **map*,
> **uint** *index*)

The *bitmap_free* routine marks the resource at *index* in the supplied *map* as available (free).

bitmap_new

> **BITMAP** **bitmap_*new(
> **uint** *low*,
> **uint** *high*)

The *bitmap_new* routine creates a new bitmap representing a set of indexable resources. The resource index runs from *low* to *high* inclusive (both must be greater than zero). That is, a call to *bitmap_claim* will return an unsigned integer in $low..high$.

bitmap_reset

void *bitmap_reset*(
    BITMAP **map*)

The *bitmap_reset* routine marks all the resources in the supplied *map* as available (free).

bzero

void *bzero*(
    void **to*,
    int *len*)

The *bzero* routine sets *len* bytes in *to* to zero.

calloc

void **calloc*(
    size_t *nobj*,
    size_t *size*)

The *calloc* routine returns an area $nobj \times size$ bytes long, cleared to zeros, using the underlying *rome_alloc* routine. The memory is allocated from the local RAM pool shared by all process, and must be freed as a single unit.

canput

(int) *canput*(
    queue_t **_q*)

The *canput* macro notionally returns *TRUE* if there is space in the queue *_q* for another message block. In ROME, flow control is determined by the availability of message blocks to be used as replies, not by per-stream options. This macro always returns TRUE.

clearerr

(void) *clearerr*(
    FILE **_stream*)

The *clearerr* macro clears the error and eof flags for the FILE *_stream*.

clock

clock_t *clock*(**void**)

The *clock* routine returns -1, since CPU time used by programs is not maintained by ROME.

ctime

char **ctime*(
    const time_t **tp*)

The *ctime* routine converts the calendar time *tp* into a printable string in standard format. Note that the returned pointer is allocated in a static buffer, so the data must be copied if they are to be stored.

difftime

> **double** *difftime*(
>   **time_t** *time2*,
>   **time_t** *time1*)

The *difftime* routine, which returns the floating-point time difference between two times, is implemented only if the *CPU_HAS_FLOAT* option is set in the system.

div

> **div_t** *div*(
>   **int** *num*,
>   **int** *denom*)

The *div* routine return the quotient and remainder of *num* devided by *denom*. This routine does not take any special actions, and is equivalent to using the '/' and '%' operators directly.

exit

> **void** *exit*(
>   **int** *status*)

The *exit* routine terminates a process by closing all open files, calling the *atexit* routines and waiting forever on a non-existant message. Note that this does not prevent other processes from sending message to that process. In general, *exit* should not be used for ROME processes running in an embedded system. The routine is only available in the process context. To terminate ROME from another context, the *rome_fatal* routine may, when all else fails, be invoked.

fclose

> **FILE** **fclose*(
>   **FILE** **stream*)

The *fclose* routine sends a ROME *CLOSE* message on the *stream* and waits for the reply. It also frees the associated file structure and file-slot in the process. The routine is only available in the 'process context'.

feof

> **(int)** *feof*(
>   **FILE** **_stream*)

The *feof* macro returns *TRUE* if the End-of-File indication is set for the FILE *_stream*.

ferror

> **(int)** *ferror*(
>   **FILE** **_stream*)

The *ferror* macro returns *TRUE* if the Error indication is set for the FILE *_stream*.

fflush

FILE *fflush*(
    FILE **stream*)

The *fflush* routine sends a ROME *FLUSH* message on the specified *stream* and waits for the reply. The routine is only available in the ‘process context’.

fgetc

int *fgetc*(
    FILE **stream*)

The *fgetc* routine returns a singel character from *stream.* This is either the character most recently passed to *ungetc* or the next character in the stream obtained by a ROME *FETMBLK* message. The routine is only available in the ‘process context’.

fgets

int *fgets*(
    char **s*,
    int *n*,
    FILE **stream*)

The *fgets* routine returns up to *n* characters from *stream*, including any character pushed back onto the stream with *ungetc*. The characters are fetched from the stream with a ROME *FETMBLK* message. The routine is only available in the ‘stdio context’.

fopen

FILE **fopen*(
    const char **filename*,
    const char **mode*)

The *fopen* routine initialises a FILE structure for subsequent messaging operations, using pended ROME calls. The *filename* must be (ultimately) the string for of a ROME-URL identifying a process within the system. The routine will add the current root and working directory to strings that do not contain a ‘:’ scheme identifier. The *mode* strings are as defined in B1.1 of K&R. The routine is only available in the ‘process context’.

fprintf

int *fprintf*(
    FILE **stream*,
    const char **format*,
    . . .)

The *fprintf* routine formats the arguments following the fixed parameters according to the supplied *format*, and sends the resulting buffer to the file *stream* using pended ROME. The *printf* description (below) lists the supported formatting options. It is only available in the ‘process context’.

fputc

> **int** *fputc*(
>   **int** *c*,
>   **FILE** **stream*)

The *fputc* routine sends the character *c* for output on *stream* using a ROME *PUTMBLK* pended message. The routine is only available in the ‘process context’.

fputs

> **int** *fputs*(
>   **const char** *s*,
>   **FILE** **stream*)

The *fputs* routine sends the string *s* for output on *stream* using a pended ROME *OUTMBLK* message (using the string as the output buffer). The routine is only available in the ‘process context’.

fread

> **size_t** *fread*(
>   **void** **array*,
>   **size_t** *size*,
>   **size_t** *nobj*,
>   **FILE** **stream*)

The *fread* routine returns at most $size \times nobj$ bytes from *stream* into *array* in a single pended ROME *FETMBLK* call. The routine is only available in the ‘process context’.

free

> **void** *free*(
>   **void** **ptr*)

The *free* routine returns the area of memory pointed to by *ptr* to the free pool using the *rome_free* system routine. Note that this memory must have been previously allocated by a call to *calloc*, *malloc* or the underlying system routine *rome_alloc*.

freeb

> **void** *freeb*(
>   **mblk_t** **_m*)

The *freeb* macro returns an mblk *_m* to the free pool. It expands to a direct call of *rome_free*. It should not be used when mblk areas are allocated in special memory, for example by devices.

freemsg

> **void** *freemsg*(
>   **mblk_t** **_m*)

The *freemsg* routine returns an mblk chain to the free pool. It expands to a direct call of *rome_free* for each linked mblk. It should not be used when mblk areas are allocated in special memory, for example by devices.

freopen

FILE **freopen*(
  const char **filename*,
  const char **mode*,
  FILE **stream*)

The *freopen*} routine re-initialises the file *stream* to be connected to *filename* with mode *mode.* The *fopen* routine above describes the requirements on these two strings. The routine is only available in the ‘process context’.

fscanf

int *fscanf*(
  FILE **stream*,
  const char **format*,
  . . .)

The *fscanf* converts the characters from the file *stream* according to supplied *format*, placing the results in the following arguments. The supported format effectors are listed in *scanf* below. It is only available in the ‘process context’.

fwrite

size_t *fwrite*(
  const void **array*,
  size_t *size*,
  size_t *nobj*,
  FILE **stream*)

The *fwrite* routine outputs $size \times nobj$ bytes from *array* onto *stream* in a single pended ROME *OUTMBLK* message. The routine is only available in the ‘process context’.

getc

int *getc*(
  FILE **_stream*)

The *getc* macro expands directly to a *fgetc*(*_stream*) call, and is only available in the ‘process context’.

getchar

int *getchar*(**void**)

The *getchar* macro expands directly to a *fgetc*(*stdin*) call and is only available in the ‘stdio context’.

getenv

char **getenv*(
  const char **name*)

ROME does not support arbitrary ‘C’ environment variables. This routine returns *NULL* for all input strings. Some specific environment information may be available through particular non-volatile RAM implementations, and the *rome_getcwd* and *rome_getroot* routines..

gets

*char *gets*(
char **s*)

The *gets* routine reads a line from *stdin* using a sequence of *fgetc* calls. It discards the trailing newline. The routine is only available in the ‘stdio context’.

gmtime

struct tm **gmtime*(
const time_t **tp*)

The *gmtime* routine returns *NULL* for all input.

isalnum

(int) *isalnum*(
char *_c*)

The *isalnum* macro returns *TRUE* if the argument *_c* is an alphanumeric character. *_c* is evaluated only once.

isalpha

(int) *isalpha*(
char *_c*)

The *isalpha* macro returns *TRUE* if the argument *_c* is an alphabetic character. *_c* is evaluated only once.

iscntrl

(int) *iscntrl*(
char *_c*)

The *iscntrl* macro returns *TRUE* if the argument *_c* is a control character. *_c* is evaluated only once.

isdigit

(int) *isdigit*(
char *_c*)

The *isdigit* macro returns *TRUE* if the argument *_c* is a decimal digit character. *_c* is evaluated only once.

isgraph

(int) *isgraph*(
char *_c*)

The isgraph macro returns TRUE if the argument *_c* is a graphic format effector character. *_c* is evaluated only once.

islower

(int) *islower*(
char *_c*)

The *islower* macro returns *TRUE* if the argument *_c* is a lower-case alphabetic character. *_c* is evaluated only once.

isprint

(int) *isprint*(
char *_c*)

The *isprint* macro returns *TRUE* if the argument *_c* is a printable character. *_c* is evaluated only once.

ispunct

(int) *ispunct*(
char *_c*)

The *ispunct* macro returns *TRUE*} if the argument *_c* is a punctuation character. *_c* is evaluated only once.

isspace

(int) *isspace*(
char *_c*)

The *isspace* macro returns *TRUE* if the argument *_c* is an white-space character. *_c* is evaluated only once.

isupper

(int) *isupper*(
char *_c*)

The *isupper* macro returns *TRUE* if the argument *_c* is an upper-case alphabetic character. *_c* is evaluated only once.

isxdigit

(int) *isxdigit*(
char *_c*)

The *isxdigit* macro returns *TRUE* if the argument *_c* is a hexadecimal digit character.*_c* is evaluated only once.

labs

long *labs*(
long *n*)

The *labs* routine returns the absolute value of *n*.

ldiv

ldiv_t *ldiv*(
  long *num*,
  long *denom*)

The *ldiv* routine return the quotient and remainder of *num* divided by *denom*. The routine does no special processing and is equivalent to the use of the '/' and '%' operators.

linkb

void *linkb*(
  mblk_t **base*,
  mblk_t **add*)

The *linkb* routine concatenates *add* to the end of the mblk chain rooted at *base*.

localtime

struct tm **localtime*(
  const time_t *tp*)

The *localtime* routine converts the calendar time in *tp* (seconds since 1970) into the returned tm structure. Note that the returned pointer is allocated on the local stack, the data must be copied if they are to be stored.

longjump

void *longjump*(
  jmp_buf *env*,
  int *val*)

The *longjmp* routine (actually the *cpu_longjump* routine) executes a long jump to the environment saved in the *env* variables with return code *val*. This implementation of this routine is machine dependent.

malloc

void **malloc*(
  size_t *size*)

The *malloc* routine returns an area *size* bytes long, with uninitialized values, from the common memory pool shared by all processes. Note that this routine never returns *NULL* as a failure indication, since the underlying *rome_alloc* routine will call *rome_fatal* if the system runs out of memory.

memchr

void **memchr*(
  const void **src*,
  uchar *c*,
  int *len*)

The *memchr* routine returns a pointer to the first occurence of *c* within *len* bytes of *src*.

memcmp

int *memcmp*(
  const void **one*,
  const void **two*,
  int *l*)

The *memcmp* routine compares at most *l* bytes between *one* and *two*.

memcpy

void **memcpy*(
  void **to*,
  const void **from*,
  int *len*)

The *memcpy* routine copies *len* bytes from *from* to *to*.

memmove

void **memmove*(
  void **to*,
  const void **from*,
  int *len*)

The *memmove* routine copies *len* bytes from *from* to *to*, preserving data ordering if the areas overlap.

memset

void **memset*(
  void **dst*,
  uchar *val*,
  int *len*)

The *memset* routine sets the first *len* bytes of *dst* to the byte *val*.

mktime

time_t *mktime*(
  struct tm **tp*)

The *mktime* routine converts the local time in the *tp* structure into calendar time (seconds since 1970), or -1 if the structure is invalid.

noenable

(void) *noenable*(
  queue_t **_q*)

The *noenable* macro notionally disables the service procedure for a STREAMS queue *_q*. This operation is not supported in ROME, and the macro.

perror

size_t *perror*(
    const char **s*)

The *perror* routine prints the error message for the current process onto *stderr.* The routine is only available in the ‘stdio context’.

printf

int *printf*(
    const char **format,*
    . . .)

The *printf* routine formats the arguments following the supplied parameters according to *format* and send the resulting buffer to the *stdout* file using pended ROME. The full range of integer and string operations are supported according to table B-1 of K&R. In addition, the ‘I’ format effector prints a 32bit unsigned integer argument as a ‘dotted quad’ Internet Protocol address. The floating-point ‘e’, ‘f’ and ‘g’ format effectors are only available if the *CPU_HAS_FLOAT* option is enabled for the system and a compatible floating-point plugin is linked into the image to supply the *cpu_internal_pfloat* routine. The *printf* routine is only available in the ‘stdio context’.

putc

int *putc*(
    int *_c*,
    FILE **_stream*)

The *putc* macro expands directly to a *fputc*(*_c*, *_stream*) call, and is only available in the ‘process context’.

putchar

int *putchar*(
    int *_c*)

The *putchar* macro expands directly to a *fputc*(*_c*, *stdout*) call. It is available only in the ‘stdio context’.

putnext

void *putnext*(
    queue_t **_q*,
    mblk_t **_m*)

The *putnext* macro notionally adds the message *_m* to the next queue on a STREAM. In ROME, this function is split between the *rome_pass_downwards* routine for intermediate modules, and *rome_reply* for device drivers. The *putnext* macro implements the device driver requirements and expands to *rome_reply*(*_m*).

putq

void *putq*(
queue_t **rq*,
mblk_t **mp*)

The *putq* should not be used in ROME programs as it destroys the local context chain. Instead the *rome_pass_downstream* and *rome_pass_upstream* routines are provided for modules, and the *putnext* routine for drivers.
The *putq* routine calls *rome_fatal* if it is used.

puts

char **puts*(
const char **s*)

The *puts* routine writes the supplied string *s* onto *stdout* using *fputs*, followed by a newlinr character. The routine is only available in the 'stdio context'.

QNAME

char **QNAME*(
queue_t **_q*)

The *QNAME* macro notionally returns the name of the STREAMS queue represented by the *_q* parameter. In ROME it returns the name of the destination process to which the queue points.

qprocson

(void) *qprocson*(
queue_t **_q*)

The *qprocson* macro notionally enables the service procedures for a STREAMS queue *_q*. In ROME, the 'service procedure' is always enabled, and this macro has no effect.

raise

int *raise*(
int *sig*)

The *raise* routine explicitly calls (and so resets) the signal handler for *sig* in the current process. The rouine is only available in the 'process context'.

rand

int *rand*(**void**)

The *rand* routine returns a pseudo-random integer.

realloc

void **realloc*(
void **ptr*,
size_t *size*)

The *realloc* routine returns a block of memory *size* bytes long containing the same values as the memory pointed to by *ptr.* The original *ptr*, which is freed, must reference memory allocated by a call to *calloc*, *malloc* or the underyling system routine *rome_alloc*.

scanf

int *scanf*(
  const char **format*,
  . . .)

The *scanf* routine converts characters from the file *stdin* according to the supplied *format* and places the results in the following arguments. The routine supports all the integer and string format effectors listed in table B-2 of K&R. The floating-point ‘e’, ‘f’ and ‘g’ format effectors are only available if the *CPU_HAS_FLOAT* option is enabled for the system and a compatible floating-point plugin is linked into the image to supply the *strtod* routine. The *scanf* routine is only available in the ‘stdio context’.

setjmp

int *setjmp*(
  jmp_buf *env*)

The *setjmp* routine (actually the *cpu_setjmp* routine) establishes an environment in *env* for a subsequent call to *longjump,* and returns 0. The parameter passed is a pointer to a **struct _jmp_buf** structure defined by the CPU plugin. The implementation of this routine is machine dependent.

signal

void (*signal*(
  int *sig,*
  void (*handler*)(**int**)))(**int**)

The *signal* routine installs a handler for the signal *sig* for the current process. This routine is only available in the ‘process context’.

sprintf

int *sprintf*(
  char **s*,
  const char **format*,
  . . .)

The *sprintf* routine formats the suppied arguments following the fixed paramters according to the *format* into the buffer *s.* It is the responsibility of the caller to ensure that the supplied buffer is large enough to hold the resulting output string. The *printf* description (above) lists the supported formatting options.

srand

void *srand*(
  uint *seed*)

The *srand* routine sets the seed variable for the *rand* routine.

sscanf

> **int** *sscanf*(
>   **char** **buffer*,
>   **const char** **format*,
>   . . .)

The *sscanf* converts the characters in *buffer* according to supplied *format*, placing the results in the following arguments. The supported format effectors are listed in *scanf* above.

strcat

> **char** **strcat*(
>   **char** **to*,
>   **const char** **from*)

The *strcat* routine does byte-by-byte concatenation of the two strings in a machine-independent, though possible not optimal implementation.

strchr

> **char** **strchr*(
>   **const char** **src*,
>   **char** *c*)

The *strchr* routine searches for *c* in *src*.

strcmp

> **char** **strcmp*(
>   **char** **to*,
>   **const char** **from*)

The *strcmp* routine compares (byte-by-byte) two strings.

strcpy

> **char** **strcpy*(
>   **char** **to*,
>   **const char** **from*)

The *strcpy* routine does byte-by-byte copying of the two strings in a machine-independent, though possible not optimal implementation.

strcspn

> **size_t** *strcspn*(
>   **const char** **one*,
>   **const char** **two*)

The *strcspn* routine returns the length of the leading substring of *one* containing characters not in *two*.

strerror

> **char** **strerror*(
>   **int** *n*)

The *strerror* routine returns a pointer to the text string corresponding to the error code *n* for the error codes defined in the ROME system. The returned string is read-only, and may be dynamically generated, so the pointer should not be saved.

strftime

> **size_t** *strftime*(
>   **char** **s*,
>   **size_t** *smax*,
>   **const char** **fmt*,
>   **const struct tm** **tp*)

The *strftime* routine format the time supplied in *tp* into the buffer *s* according to the format *fmt*. The format specifications supported are those listed in K&R.

strlen

> **size_t** *strlen*(
>   **const char** **src*)

The *strlen* routine returns the length of the string *src*.

strncat

> **char** **strncat*(
>   **char** **to*,
>   **const char** **from*,
>   **int** *n*)

The *strncat* routine concatenates (byte-by-byte) at most *n* characters.

strncmp

> **char** **strncmp*(
>   **char** **to*,
>   **const char** **from*,
>   **int** *n*)

The *strncmp* routine compares (byte-by-byte) at most *n* characters.

strncpy

> **char** **strncpy*(
>   **char** **to*,
>   **const char** **from*,
>   **int** *n*)

The *strncpy* routine copies (byte-by-byte) at most *n* characters.

strpbrk

char **strpbrk*(
  const char **one*,
  const char **two*)

The *strpbrk* routine returns the residual string of *one* after stripping off all characters in *two*.

strrchr

char **strrchr*(
  const char **src*,
  char *c*)

The *strrchr* routine searches backwards for *c* in *src*.

strspn

size_t *strspn*(
  const char **one*,
  const char **two*)

The *strspn* routine returns the length of the leading substring of *one* containing characters in *two*.

strstr

char **strstr*(
  const char **one*,
  const char **two*)

The *strstr* routine returns the first occurence of *two* in *one*.

strtok

char **strtok*(
  char **s*,
  const char **ct*)

The *strtok* routine splits the input string *s* into tokens delimited by characters in *ct*. This routine uses a field in the process control block to save context between multiple calls and can only be called in 'process context'.

system

int *system*(
  const char **string*)

The *system* routine sends *string* as a pended ROME *COMMAND* message to the process defined by the *CLIB_SYSTEM_PROCESS* option. If the option is not set, the routine ignores the command and returns zero.

time

> **time_t** *time*(
>   **time_t** **tp*)

The *time* routine returns the current calendar time for systems on which the *NVRAM_RTC_ENABLED* option is set, and so the *rtc_read_time* routine exists, otherwise the routine returns -1.

tolower

> **int** *tolower*(
>   **int** *c*)

The *tolower* routine returns the lower-case equivalent of *c* if *c* is a upper-case letter, or *c* otherwise. Note that as *tolower* is a routine, it may be called with an auto-incrementing argument.

toupper

> **int** *toupper*(
>   **int** *c*)

The *toupper* routine returns the upper-case equivalent of *c* if *c* is a lower-case letter, or *c* otherwise. Note that as *toupper* is a routine, it may be called with an auto-incrementing argument.

ungetc

> **int** *ungetc*(
>   **int** *_c*,
>   **FILE** **_stream*)

The *ungetc* macro pushes the character *_c* back onto *_stream*. Only one character may be pushed per stream. Note that *ungetc* will only work correctly with calls to *fgetc* or a macro equivalent.

unlinkb

> **mblk_t** **unlinkb*(
>   **mblk_t** *base)

The *unlinkb* routine removes the head mblk from the chain rooted at *base* and returns the rest of the chain.

vfprintf

> **int** *vfprintf*(
>   **FILE** **stream*,
>   **const char** **format*,
>   **va_list** *args*)

The *vfprintf* routine formats the supplied variable-arguments in *args* according to the specified *format* and sends the resulting string to the output file *stream* using pended ROME. The *printf* description (above) lists the supported formatting options. It is only available in the ‘process context’.

vprintf

> **int** *vprintf*(
>   **const char** **format,*
>   **va_list** *args*)

The *vprintf* routine formats the supplied variable-arguments in *args* according to the specified *format* and sends the resulting string to the *stdio* file using pended ROME. The *printf* description (above) lists the supported formatting options. It is only available in the ‘stdio context’.

vsprintf

> **int** *vsprintf*(
>   **char** **buffer,*
>   **const char** **format,*
>   **va_list** *args*)

The *vsprintf* routine formats the supplied variable-arguments in *args* according to the specified *format* into the supplied *buffer,* which is assumed to be large enough to contain the resulting string. The *printf* description (above) lists the supported formatting options.

WR

> **queue_t** **WR*(
>   **queue_t** *_*q*)

The *WR* macro notionally returns the ‘writer’ queue of a pair of STREAMS queues. In ROME, the two queues are represented by the same structure, and the macro returns its argument, _*q*.