RadiSys.

iRMX®
C Library Reference

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR   97124
(503) 615-1100
FAX: (503) 615-1150
www.radisys.com
07-0698-01
December 1999

Quick Contents

Notational Conventions

Descriptive text in this manual uses these notational conventions:

C library functions and macros appear **like this**, for example **fprintf( )**. C functions are indicated by the **( )** suffix. iRMX system calls appear **like this** and have an **rq_** prefix, for example **rq_exit_io_job**.

Standard C language syntax as used in your program, including constants, keywords, identifiers, and types, appears `like this`. Variable names also appear `like this`, for example `type, member`.

Filenames and book titles appear *like this*, for example *:config:r?env*, *System Call Reference*. In addition, C header filenames are indicated by surrounding < > characters.

If ANSI appears in the heading, this is an ANSI function. If stdio appears, this function requires that the calling task has access to the standard streams. If DOS appears in the function heading, this is a DOS function.

These abbreviations are used:

Abbreviation	**Meaning**
ANSI	American National Standard for Information Systems, C programming language
BIOS	Basic I/O system layer for the iRMX OS
<CR>	Carriage-return character
C task	Process (task) that uses the C library
EIOS	Extended I/O system layer for the iRMX OS
Epoch time	00:00:00, January 1, 1970, GMT
GMT	Greenwich mean time
HI	Human Interface layer for the iRMX OS
ICU	iRMX Interactive Configuration Utility
I/O	Input/output
<LF>	Line-feed character
stdio	Indicates that access to the standard streams: *stdin*, *stdout*, and *stderr* is required

Related Publications

For additional information about the C programming language and library functions

See also: *C: A Reference Manual* by Harbison and Steele,
The Standard C Library by P.J. Plauger

The term ANSI indicates that a function conforms to the 1989 American National Standard for Information Systems - Programming Language C (ANSI X3.159-1989). The C library provides a superset of ANSI functionality, with additional features defined by the IEEE Portable Operating System Interface for Computer Environments (POSIX).

See also: IEEE Std 1003.1-1988, *IEEE Standard Portable Operating System Interface for Computer Environments*, copyright 1988, by The Institute of Electrical and Electronics Engineers, Inc.

This standard provides locale-specific information, such as the alphabetic international currency symbol.

See also: *ISO 4217 Codes for the Representation of Currency and Funds*

Various mathematics reference books provide information about the Bessel functions.

See also: *Handbook of Mathematical Functions* (Abramowitz and Stegun; Washington: U.S. Government Printing Office, 1964)

For further information refer to the manuals provided with your C compiler.

Contents

Tables

Introduction 1

The shared C library includes functions and macros for applications that run in the iRMX® Operating System environment. This manual describes the iRMX shared C library; it is intended for C program developers who are using a compiler that follows ANSI and POSIX C language standards.

This manual assumes general knowledge of the C programming language, standard programming techniques and the iRMX OS.

This chapter provides general information that is helpful in using the *C Library Reference*:

- C library overview
- Supplied C library files
- DOS syntax
- Support for non-Intel development tools
- Overview

Shared C Library Overview

The C library is available as an iRMX OS extension job to your application in one of two ways:

- Run-time loadable job, *clib.job*, loaded using the HI **sysload** command
- Resident first-level job, set up using the Interactive Configuration Utility (ICU) Sub-systems and Shared C Library screens

Instead of each C application having to link to its own copy of the library, this job is a system-wide library that can be shared by all tasks and jobs in the system. This greatly reduces the code size of individual applications as well as decreases the time required to bind (link), load, and execute the applications. For example, if you run five applications that call a certain C function and each application is individually linked to a C library, the code for that function is loaded into memory five different times. With the shared C library, there is only one copy of the function loaded, and it is available to all five applications.

You link each application to a small interface library, which provides access to the shared C library.

Any number of tasks and jobs can share the C library. Each task can have its own data segment; the data segment does not have to be shared. A few functions related to signal handling, such as **abort**, **raise**, and **signal**, are private to each task. These functions are in the interface library linked to the task, not in the shared C library.

The shared C library supports many standard C functions that enable a task to perform common, OS-independent operations without making direct, iRMX OS-dependent system calls. You can mix shared C library calls with direct iRMX system calls in your application.

The shared C library takes care of iRMX OS-dependent operations such as multitasking, time-of-day, signal management, and environment management; this enables you to create portable code using standard ANSI and POSIX programming practices.

The C Library includes floating point functions and macros and links to the standard floating point libraries; there are no separate libraries for floating point applications.

Depending on your system configuration, the C library may not support all of the functions mentioned in this manual.

See also: Supplied C Library Files, in this chapter
C Library, *clib.job, System Configuration and Administration*

Shared C Library Advantages

The C library can be shared concurrently by multiple tasks and jobs running on the system. The advantages of the shared C library are:

Code size	Only one small interface library, which provides access to the shared C Library, is bound to the application.
Bind speed	Only the application and interface library symbolic information need to be processed.
Load speed	The application which utilizes the shared C Library is much smaller.
Execution speed	Because the shared C Library is an iRMX OS extension, the need for localized task and job management is eliminated. In addition, many small functions are performed in the interface library itself, rather than by the shared C library.

Each C job or task can have its own data segment; this segment does not have to be shared with other C jobs or tasks using the library.

Resources Allocated to C Tasks and Jobs

The C library automatically manages common system resources such as I/O interfaces and memory when your code makes calls that use these common resources.

Job Resources

Each C job uses resources which count against the memory and object limits for the job. When a C job is created, the C library allocates one private memory heap from the job's memory pool; every C task is associated with its owner job's heap. The C library allocates additional resources when a task in the job makes the first call to a C library function; these resources consist of a bookkeeping segment for heap management, **exit( )** register, **stat( )** directory cache, and one synchronization semaphore for the heap manager. When the job terminates, these resources are automatically deleted. The **malloc** mutual exclusion semaphore and any **malloc** segments are also deleted when the job is deleted.

Task Resources

When a task makes the first C library call, some task-specific resources are automatically allocated and maintained locally. These include data structures and semaphores that support the task's operation in the multitasking environment.

The standard I/O functions are contained in the *stdio* header file. When the first call is made to an *stdio* function, all of the standard streams are created, open for sharing by all tasks. The stdio connections are cataloged using the existing *:ci:* as *stdin*, *:co:* as *stdout*, and *:term:* as *stderr*. These connections and the memory required for them are added to the resources allocated to the task. They also count against the memory and object limits for the job. Thus, if a task does not make *stdio* calls, it consumes fewer resources. You can minimize the total amount of resources required by an application by having a single task that calls *stdio* functions, or by dynamically creating and deleting tasks that call *stdio* functions.

The streams are opened using the iRMX system calls **rq_s_attach_file** and **rq_s_open**.

Table 1-1 lists functions which are responsible for input and output.

Table 1-1. Input and Output Functions

chmod	chsize	close	creat
_dup	dup2	eof	fclose
fcloseall	fdopen	fgetpos	filelength
fopen	freopen	fstat	ftell
getenv	getuid	isatty	lseek
ltell	mkdir	mktemp	open
putenv	remove	rename	rmdir
stat	tmpfile	tmpnam	tzset
unlink	utime	_tzset_ptr	_dos_close
_dos_creat	_dos_creatnew	_dos_findfirst	_dos_findnext
_dos_getdate	_dos_getftime	_dos_open	_dos_setdate
_dos_setftime	_tzset_ptrs		

The resources associated with a C task are not automatically freed when the task is deleted with **rq_delete_task**. Before you delete a C task using **rq_delete_task**, delete the task's C library-specific resources using the **_cstop( )** function.

Most resource allocations apply to each task; there are also resources allocated to each job containing C library applications. Table 1-2 lists the resources used per task and per job. Each connection object, mailbox, and semaphore consumes from the object limits for the job.

Table 1-2. Resources Used for C Tasks and Jobs

Resources Required For:	Memory	Semaphores	Mailboxes	Connection Objects
Each Job	600 bytes	1		
Each Task	300 bytes	2		
Additional for each *stdio* Task	400 bytes	3	6	3

Supplied C Library Files

The iRMX OS provides header (include) files containing declarations for C library functions and definitions of related macros and data types. The shared C library loadable job is *clib.job*.

32-bit interface libraries are provided, as well as cstart modules which initialize processes and call *main()*.

There are a variety of interface libraries supplied with the OS for the interface to C library functions and iRMX system calls. For different Intel and non-Intel tools you must bind (link) to different libraries.

See also: Cstart, *iC-386 Compiler User's Guide*,
Cstart modules to use with non-Intel compilers, *Programming Techniques*,
Interface libraries, *System Call References*, for a complete list of interface libraries for different compilers,
Header files for a description of the include libraries, in this chapter,
clib.job, System Configuration and Administration

The Cstart Module

Each application must link to the cstart module. This code makes library calls that set up an internal C environment for your application. To make initialization changes in earlier (individually linked) versions of the C library, you would change source code for the cstart module and reassemble it before linking to your code. With the shared C library it is rarely necessary to make initialization changes in cstart. However, there are two configuration changes you can make.

The source code for the cstart module defines values for two literals used in parsing of command lines. Edit and re-assemble a copy of the cstart source code to change these values:

- _ARGV_MAX, the maximum number of command-line parameters (*argv*)
- _COMMAND_MAX, the maximum number of characters in a command line

DOS Syntax

You can use DOS syntax or iRMX syntax in all C library calls that require a pathname argument. DOS backslashes are converted to iRMX forward slashes and DOS device names are converted to iRMX logical names.

Support for Development Tools

You can develop applications with DOS-based development tools by using these provided iRMX elements:

- A set of common C header files, compatible with all supported compilers.
- A custom cstart module for each supported compiler.
- An interface library to the shared clib, for each supported compiler.
- An OMF translator to convert *.exe* and *.exp* files to OMF-386.

See also: *Programming Techniques* for more details on third-party compilers, *System Call Reference*, for information on interface libraries

The following configuration and compiler control header files control program compilation without being compiler-specific.

<*_align.h*>	Starts 2-byte/4-byte structure alignment (16-bit/32-bit compilers); default header file, required to support multiple compilers
<*_noalign.h*>	Ends multiple-byte alignment (refer to <*_align.h*> above); provides compiler-independent 1-byte structure alignment (no alignment)
<*yvals.h*>	Standard C values and support definitions that help make the other header files compiler-independent
<*_restore.h*>	Returns structure alignment to the compiler default

Header Files

The header files described here contain declarations for C library functions and definitions of related macros and data types. For more complete and detailed information, see the header files themselves.

See also: Header files, *System Call Reference*, for a list of iRMX OS-specific header files

⚠ **CAUTION**

For the C functions to work properly, you must use the header files, and you must not change them.

Header File	Contents
<_align.h>	Starts 2-byte/4-byte alignment (16-bit/32-bit compilers); default header file, required to support multiple compilers
<assert.h>	Assert macro (diagnostic tool)
<ctype.h>	Character handling functions and macros
<conio.h>	DOS-specific console I/O functions
<direct.h>	Directory management functions and types
<dos.h>	DOS system call macros
<errno.h>	Error indication macros
<fcntl.h>	File access mode and status flag macros
<float.h>	Floating-point types and constants
<io.h>	File input/output functions
<limits.h>	Ranges of integer and character types
<locale.h>	Locale-specific functions, types, and macros
<math.h>	Floating-point math functions and macros
<_noalign.h>	Ends multiple-byte alignment (refer to *<_align.h>* above); provides compiler-independent 1-byte alignment (no alignment)
<process.h>	Task execution and identification functions and types
<_restore.h>	Returns structure alignment to the compiler default
<rmxtypes.h>	Makes iRMX PL/M data types available to C programmers
<search.h>	Linear search functions
<setjmp.h>	Non-local jump functions and environment structure
<share.h>	Access, sharing and inheritance rights
<signal.h>	Signal handling functions and signals
<stdarg.h>	Variable-argument list macros
<stddef.h>	Common types and macros
<stdio.h>	Stream input/output functions, macros, and types
<stdlib.h>	Utility functions, macros, and types

Header File	Contents
<string.h>	String handling functions

<*sys/stat.h*>	File information functions, macros, manifest constants, and types
<*sys/types.h*>	File information primitive types
<*sys/utime.h*>	**utime** function and type
<*time.h*>	Date/time functions, macros, and types
<*udi_c.h*>	iRMX UDI system calls
<*unistd.h*>	Symbolic constants used by **lseek( )** function
<*yvals.h*>	Standard C values and support definitions that help make the other header files compiler-independent

You must include the appropriate header files in order to use the functions. The description of each function lists the required include statements.

❑❑❑

Functional Groupings 2

This chapter lists all the C functions, grouped to identify the functions that are appropriate for a specific purpose.

Character Processing Functions

These functions classify and convert characters for text manipulation.

isalnum	Test for alphanumeric character.
isalpha	Test for alphabetical character.
isascii	Test if a character-coded integer is an ASCII code (i.e., between 0 and 0x7F inclusive).
iscntrl	Test for control character.
isdigit	Test for decimal digit.
isgraph	Test for printable character (excluding space).
islower	Test for lowercase character.
isprint	Test for printable character (including space).
ispunct	Test for punctuation character.
isspace	Test for white space character.
isupper	Test for uppercase character.
isxdigit	Test for hexadecimal digit.
toascii	Converts character to ASCII.
tolower	Converts uppercase character to lowercase.
_tolower	Converts uppercase character to lowercase if appropriate.
toupper	Converts lowercase character to uppercase.
_toupper	Converts lowercase character to uppercase if appropriate.

Control Functions

These functions control and monitor task execution.

abort	Aborts the current job and returns the error code.
assert	Prints a diagnostic message and aborts the calling task.
atexit	Processes the specified function when the calling task terminates normally.
exit	Terminates the current job after cleanup.
_exit	Terminates the current job immediately.
getenv	Searches the environment-variable table for a specified entry.
getpid	Gets the calling task's connection token (process ID).
getuid	Gets the calling task's user ID.
longjmp	Restores the context previously saved by *setjmp*.
onexit	Registers a function to be called when the task terminates normally.
putenv	Adds new environment variables or modifies the values of existing ones.
raise	Sends a signal to the executing program.
setjmp	Saves the current context of the executing program and stores it in the specified location.
signal	Sets up one of several ways for a task to handle an interrupt signal from the OS.
sleep	Suspends a task for a specified number of seconds.
system	Invokes the system call **rq_c_send_command** to execute an iRMX command line.

Conversion Functions

These functions cover a range of purposes including conversion of various data types to strings and to wide characters.

ecvt	Converts a value to a character string.
fcvt	Converts a floating point value to a string.
ftoa	Converts a double value to a formatted string.
gcvt	Converts a double value to a string of significant digits and places them in a specified location.
itoa	Converts an integer of the specified base to a null-terminated string of characters and stores it.
itoh	Converts an integer into the equivalent null-terminated, hexadecimal string and stores it.
ltoa	Converts a long integer of the specified base to a null-terminated string of characters and stores it.
ltoh	Converts a long integer to a null-terminated hexadecimal string and stores it.
ltos	Converts a long integer to a null-terminated string of characters and stores it; negative base values are acceptable.
mblen	Gets the length and determines the validity of a multibyte character.
mbstowcs	Converts a sequence of multibyte characters to a sequence of wide characters, as determined by the current locale; stores the resulting wide-character string at the specified address.
strtod	Converts a string to double.
strol	Converts a string to long.
strtoul	Converts a string to unsigned long.
ultoa	Converts unsigned long to a null-terminated string and stores it without overflow checking.
utoa	Converts an integer to a null-terminated string and stores it without overflow checking.
wcstombs	Converts a sequence of wide characters to a corresponding sequence of multibyte characters.
wctomb	Converts a wide character to a corresponding multibyte characters.

DOS Console I/O Functions

These functions provide DOS-compatible ways for an application to get input from or provide output to the console.

cgets	Gets a character string from the console and stores it.
cprintf	Formats a string and prints to the console.
cputs	Writes a null-terminated string directly to the console.
cscanf	Reads formatted data from the console into the specified locations.
getch	Reads a single character from the console without echoing.
getche	Reads a single character and echoes the character read.
putch	Writes a character directly (without buffering) to the console.
ungetch	Pushes a character back to the console, causing that character to be the next character read.

DOS Interface Functions

These functions provide a DOS-like interface for DOS program compatibility.

_dos_allocmem	Allocates a block of memory.
_dos_close	Closes a file.
_dos_creat, _dos_creatnew	These functions create and open a new file with the specified access attributes.
_dos_findfirst, _dos_findnext	*_dos_findfirst* finds the first file with the specified name and attributes; *_dos_findnext* finds the next file.
_dos_freemem	Releases a block of memory previously allocated by *_dos_allocmem.*
_dos_getdate	Gets the current system date.
_dos_getftime	Gets the date and time that a file was last written.
_dos_gettime	Gets the current system time.
_dos_open	Opens an existing file.
_dos_read	Reads a specified number of bytes of data from a file.
_dos_setdate	Sets the current system date.
_dos_setftime	Sets the date and time that a file was last written.
_dos_settime	Sets the current system time.
_dos_write	Writes a specified number of bytes from a buffer to a file.

File Management Functions

These functions manage the file system. This includes for making directories and changing file attributes. This also includes functions for obtaining information about a file's length or a descriptor associated with a file.

chmod	Changes the permission mode of a file.
chsize	Extends or truncates the size of a file to the specified length.
closedir	Closes a directory.
filelength	Gets the length of a file in bytes.
fstat	Gets information on the file associated with the specified file descriptor.
isatty	Determines whether a file descriptor is associated with a character device: a terminal, console, printer, or serial port.
mkdir	Creates a new directory with the specified ownership and access rights.
mktemp	Creates a unique temporary filename.
opendir	Opens a directory.
readdir	Reads a directory.
rewinddir	Resets a directory.
rmdir	Deletes a directory.
setmode	Sets binary or text translation mode of a file.
stat	Gets information on a file.
umask	Sets the default file-permission mask of the current process to the specified mode.
unlink	Deletes a file.

Input/Output Functions

These functions provide ways to control the flow of an application.

clearerr	Resets the error and end-of-file indicators for a stream.
fclose	Closes a specified stream.
fcloseall	Closes all open streams.
fdopen	Opens a stream associated with a file descriptor, allowing a file opened for low-level I/O to be buffered and formatted.
feof	Tests for end-of-file on a stream.
ferror	Tests for a read or write error on a stream.

fflush	Flushes a buffered stream (has no effect on an unbuffered stream).
fgetc	Reads a single character from the current position of the specified stream and increments the file pointer to the next character.
fgetchar	Reads from a single character from stdin.
fgetpos	Gets a stream's file pointer position-indicator and stores it. This function does not get the file pointer; use the *ftell* function instead.
fgets	Reads a specified number of characters from a stream and stores them in a string.
fileno	Gets the file descriptor associated with a stream.
flushall	Writes the contents of all buffers associated with open output streams to their associated files.
fopen	Opens a file with the specified open mode.
fprintf	Prints formatted data to a stream.
fputc	Writes a single character to an output stream at the current position.
fputchar	Writes a single character to stdout.
fputs	Writes a string to the stream at the current file pointer.
fread	Reads up to the specified number of items of the specified size from the input stream and stores them in a buffer.
freopen	Closes the file currently associated with a stream and reassigns a new file to the stream.
fscanf	Reads and formats character data from the current position of a stream into the specified locations.
fseek	Moves the file pointer to a specified location in a stream.
fsetpos	Sets a stream's file pointer position-indicator.
ftell	Gets the current position of the file pointer for a stream.
fwrite	Writes a specified number of characters to a stream.
getc, getchar	*Getc* reads a single character from a stream and increments the associated file pointer to point to the next character; *getchar* reads from stdin.
gets	Gets a line from stdin and stores it in the specified location.
getw	Reads the next integer from a stream and increments the associated file pointer (if there is one) to point to the next unread value.

perror	Prints an error message to stderr.
printf	Prints formatted data to stdout.
putc	Writes a character to a specified stream at the current position.
putchar	Writes a character to stdout.
puts	Writes a string to stdout, replacing the string's terminating null character `\0` with a newline character `\n`.
putw	Writes an integer to the current position of a stream.
rename	Renames a file or directory.
rewind	Repositions the file pointer to the beginning of a file and clears the end-of-file indicator.
rmtmp	Removes all the temporary files that were created by *tmpfile* from the current directory.
scanf	Reads from stdin at current position, and formats character data.
setbuf	Allows the user to control buffering for a stream.
setvbuf	Controls stream buffering and buffer size.
sprintf	Prints formatted data to a string.
sscanf	Reads and formats character data from a string.
tmpfile	Creates a temporary file, opens in it binary read/write mode, and returns a stream pointer to it.
tmpnam	Creates a temporary filename, which can open a temporary file without overwriting an existing file.
vfprintf	Formats and sends data to the file specified by `stream.`
vprintf	Sends data to stdout.
vsprintf	Sends data to the memory pointed to by `buffer`.

iRMX-specific Functions

These functions provide C library access to OS-specific functions.

_cstop	Deletes the C resources allocated for a task.
_get_arguments	Sets up the standard C command line parser.
_get_cs	Returns an application’s current code segment.
_get_ds	Returns an application’s current data segment.
_get_info	Obtains specific C library information.

_get_rmx_conn	Translates a file descriptor to a valid iRMX connection token, usable as a parameter in iRMX system calls.
_get_ss	Returns an application’s current stack segment.
_put_rmx_conn	Places an iRMX connection token into the file descriptor table and returns a valid file descriptor, usable as an argument in C library calls.
_set_info	Modifies C library information.

Low-level I/O Functions

These functions provide low-level ways to manage file processing

creat	Creates a new file or opens an existing file for writing and truncates it to length 0, destroying the previous contents.
eof	Checks whether the file's current file pointer is EOF.
lseek	Moves the file pointer to a location specified as an offset from the origin in a file.
ltell	Sets the absolute position of the file pointer for the next I/O operation.
open	Opens a file and prepares it for subsequent reading or writing.
read	Reads the specified number of bytes from a file into a buffer, beginning at the current position of the file pointer.
sopen	Opens a file for shared reading or writing.
write	Writes data from a buffer to a file.

Math Functions

These functions provide such math functions as integer, floating point, trigonometric operations.

abs	Calculates the absolute value of an integer.
acos	Calculates the arccosine of a double value.
asin	Calculates the arcsine of a double value.
atan	Calculates the arctangent of a double value.
atan2	Calculates the arctangent of the quotient of two doubles.
Bessel functions	Compute the Bessel function.
cabs	Calculates the absolute value of a complex number.

ceil	Calculates the *ceiling* (the smallest integer that is greater than or equal to the value) of a double value.
cos	Calculates the cosine.
cosh	Calculates the hyperbolic cosine of an angle.
div	Divides the numerator by the denominator, computing the quotient and the remainder of two integer values.
exp	Calculates the exponential of a double value.
fabs	Calculates the absolute value of a double value.
floor	Calculates the *floor* (largest integer that is less than or equal to a value) of a double value.
fmod	Calculates the floating-point remainder.
frexp	Gets the mantissa and exponent of a double value.
labs	Calculates the absolute value of a long integer.
ldexp	Computes a real number from the mantissa and exponent.
ldiv	Divides numerator by denominator, and computes the quotient and remainder.
log	Calculates the natural logarithm of a value.
log10	Calculates the base-10 logarithm.
matherr	Processes errors generated by the functions of the math library.
modf	Splits a value into fractional and integer parts, retaining the sign.
pow	Computes a value raised to the power of another value.
rand	Generates a pseudo-random number.
sin	Calculates the sine.
sinh	Calculates the hyperbolic sine of an angle.
sqrt	Calculates the square root of a number.
srand	Sets the starting point for generating a series of pseudo-random integers.
square	Calculates the square of a number.
tan	Calculates the tangent.
tanh	Calculates the hyperbolic tangent of the number.

Memory Functions

These functions copy, compare, and set blocks of memory.

memccpy	Copies characters from one buffer to another, halting when the specified character is copied or when the specified number of bytes have been copied.
memcpy	Copies specified number of bytes from a source buffer to a destination buffer.
memchr	Finds the first occurrence of a character in a buffer and stops when it finds the character or when it has checked the specified number of bytes.
memcmp	Compares the specified number of bytes of two buffers and returns a value indicating their relationship.
memicmp	Compares characters in two buffers byte-by-byte (case-insensitive).
memmove	Moves specified number of bytes from a source buffer to a destination buffer.
memset	Sets characters in a buffer to a specified character.
swab	Copies while swapping bytes.

Searching and Sorting Functions

These functions provide efficient search and sort routines.

bsearch	Performs a binary search of a sorted array.
lfind	Performs a linear search for a specified key in an unsorted array.
lsearch	Performs a linear search for a specified value in an unsorted array, appending the value to the array if not found.
qsort	Performs a quick sort of an array, overwriting the input array with the sorted elements.

Storage Allocation Functions

These functions provide storage allocation management.

calloc	Allocates and clears an array in memory; initializes each element to 0.
free	Deallocates a memory block previously allocated by *malloc*.
malloc	Allocates a memory block of the specified size.
realloc	Changes the size of a previously allocated memory block or allocates a new one.
sbrk	Creates iRMX segments of the specified number of bytes.

String Processing Functions

The following functions provide string conversion, parsing, movement and manipulation capabilities.

atof	Converts a character string to a double value.
atoi	Converts to an integer value.
atol	Converts to a long integer value.
cstr	Converts a count-prefixed iRMX-style string to a null-terminated C-style string and stores it.
strcmp, strcmpi, stricmp	Compare two null-terminated strings lexicographically.
strcat	Appends a null-terminated string to another string.
strchr	Searches for a character in a null-terminated string.
strcoll	Compares null-terminated strings using locale-specific collating sequences.
strcpy	Copies a null-terminated string.
strcspn	Finds a null-terminated substring in a string.
strdup	Duplicates null-terminated strings.
strerror	Gets a system error message.
strlen	Gets the length of a null-terminated string.
strlwr	Converts uppercase letters in a null-terminated string to lowercase. Other characters are not affected.
strncat	Appends characters to a string.
strncmp	Compares substrings.

strncpy	Copies the specified number of characters from one string to another.
strnicmp	Compares substrings without regard to case.
strnset	Sets the specified number of characters in a string to a character.
strpbrk	Searches a string for the first occurrence of any character in the specified character set.
strrchr	Searches a string for the last occurrence of a character.
strrev	Reverses the order of the characters in a string.
strset	Sets all characters in a string to a specified character.
strspn	Finds the first character in a string that does not belong to a set of characters in a substring.
strstr	Finds a substring within a string.
strtok	Finds the next token in a string.
strup	Converts any lowercase letters in a null-terminated string to uppercase.
strxfrm	Transforms a string based on locale-specific information and stores the result.
strtod	Converts a string to double.
strol	Converts to long.
strtoul	Converts to an unsigned long.
udistr	Converts a null-terminated C-style string to a count-prefixed iRMX-style string and stores it.

Time and Date Functions

These functions provides ways to control and process the time and date

asctime	Converts a time stored as a structure to a character string.
clock	Measures the time used by the calling task, from when the calling task first began execution to the current time.
ctime	Converts a time stored as a `time_t` value to a character string.
difftime	Finds the difference between two time values.
gmtime	Converts a time value to a structure.
localeconv	Gets detailed information on locale settings.
localtime	Converts a time stored as a `time_t` value and corrects for the local timezone.
mktime	Converts the time/date structure into a fully-defined structure with normalized values and then converts it to calendar time.
setlocale	Sets the task's current entire locale or specified portions of it.
strftime	Formats a time string.
time	Gets the system time.
time macros, *_tzset_ptr*	Accesses *daylight*, *timezone*, and *tzname* environment variables.
tzset	Sets the time environment variables.
utime	Sets the modification time for a file.

Variable Argument Functions

These functions provide a convenient way to access argument lists.

va_arg	Retrieves current argument.
va_end	Resets argument list pointer.
va_start	Sets argument list pointer to first optional argument.

❑❑❑

Functions 3

This chapter presents C library function descriptions in alphabetical order. In these descriptions, *double* means floating-point, double precision value.

You must include the appropriate header files in order to use the functions. The description of each function lists the required include statements. To check the **errno** value, you must include the *<errno.h>* header file.

Each C function (or group of related functions) contains a description with these elements:

- Function heading
- Required `#include` statement(s)
- Function prototype(s)
- Description of argument(s)
- Description of behavior
- Description of successful returns followed by error returns

If ANSI appears in the function heading, this is an ANSI function. If DOS appears in the function heading, this is a DOS function. If stdio appears, this is a stdio function, which requires that the calling task has access to the standard streams: *stdin*, *stdout*, and *stderr*, along with the necessary connections and memory requirements.

abort

Aborts the current task and returns an error code.

Syntax

```
#include <process.h>
      #include <stdlib.h>
      void abort (void);
```

Additional Information

Abort( ) does not flush stream buffers or do **atexit( )**/**onexit( )** processing. It does not return control to the caller.

This function calls **raise (SIGABRT)**; the response to the signal depends on the action defined in a prior call to **signal( )**. The default action is for the calling task to terminate with an **_exit( )** call.

This function is implemented in the C interface library (not in the shared C library) and is private to each application.

See also: _exit( ), raise( ), signal( )

Returns

Exit code 3 (default) to the parent job and terminates the task.

abs

Calculates the absolute value of an integer.

Syntax

```
#include <stdlib.h>
      #include <math.h>
      int abs (int n);
```

Parameter

n	Integer value whose absolute value is calculated.

See also: fabs( ), labs( ), cabs( )

Returns

The absolute value result.

No error return.

acos

Calculates the arccosine of a double value.

Syntax

```
#include <math.h>
      double acos (double x);
```

Parameter

x	Value whose arccosine is calculated. Must be between -1 and 1.

See also: asin( ), atan( ), cos( ), matherr( ), sin( ), tan( )

Returns

The arccosine result in the range 0 to π radians.

0 if x is less than -1 or greater than 1; the function sets **errno** to EDOM and prints a DOMAIN error message to *stderr*.

This function does not return standard ANSI domain or range errors.

asctime

Converts a time stored as a structure to a character string.

Syntax

```
#include <time.h>
      char *asctime (const struct tm *timedate);
```

Parameter

`timedate`
A pointer to a `tm` time/date structure, usually obtained using **gmtime( )** or **localtime( )**.

Additional Information

The converted string contains exactly 26 characters and has this form:

```
Wed Jan 02 02:03:55 1980\n\0
```

All elements have a constant width. The newline character `\n` and the null character `\0` occupy the last two positions of the string.

This function uses a 24-hour clock.

The function uses a single statically allocated buffer to hold the return string. Each call destroys the result of the previous call.

See also: Description of the `tm` structure elements in <*time.h*>, **localtime( )**, **time( )**, **tzset( )**

Returns

A pointer to the character string.

No error return.

asin

Calculates the arcsine of a double value.

Syntax

```
#include <math.h>
      double asin (double x);
```

Parameter

x	Value whose arcsine is calculated. Must be between -1 and 1.

See also: acos( ), atan( ), cos( ), matherr( ), sin( ), tan( )

Returns

The arcsine result in the range $-\pi/2$ to $\pi/2$ radians.

0 if `x` is less than -1 or greater than 1; function sets **errno** to EDOM and prints a DOMAIN error message to *stderr*.

This function does not return standard ANSI domain or range errors.

assert

Prints a diagnostic message and aborts the calling task.

Syntax

```
#include <assert.h>
      #include <stdio.h>
      void assert (int expression);
```

Parameter

`expression`
C expression specifying assertion being tested.

Additional Information

This function calls the **abort( )** function if `expression` is false (0). The diagnostic message has this form:

```
Assertion failed: expression, file filename, line linenumber
```

Where:

`filename`	Name of the source file.
`linenumber`	Line number of the assertion that failed in the source file.

No action is taken if `expression` is true (not 0).

Use the **assert( )** macro in program development to identify program logic errors. Choose `expression` so that it holds true only if the program is operating as intended.

After a program has been debugged, remove **assert( )** calls from the program using the special identifier `NDEBUG`. If `NDEBUG` is defined by any value with a `/D` command-line option or with a `#define` directive, the C preprocessor removes all **assert( )** calls from the program source.

See also: abort( ), raise( ), signal( )

Returns

Nothing.

atan, atan2

Atan( ) calculates the arctangent of a double value; **atan2( )** calculates the arctangent of the quotient of two doubles.

Syntax

```
#include <math.h>
      double atan (double x);
      double atan2 (double x, double y);
```

Parameters

`x, y`	Any number(s) whose arctangent is calculated.

Additional Information

The **atan2( )** function uses the signs of both arguments to determine the quadrant of the return value.

See also: acos( ), asin( ), cos( ), matherr( ), sin( ), tan( )

Returns

Atan( )	Returns the arctangent result in the range -π/2 to π/2 radians.
Atan2( )	Returns the arctangent result in the range −π to π radians. Returns 0 if both arguments are 0, sets **errno** to EDOM and prints a DOMAIN error message to *stderr*.

This function does not return standard ANSI domain or range errors.

atexit

Processes the specified function when the calling task terminates normally.

Syntax

```
#include <stdlib.h>
      int atexit (void (_Pascal * func) (void));
```

Parameter

`func`	Function(s) to be called; the called function(s) cannot take parameters. No more than 32 functions can be registered. **Atexit( )** receives the address of `func` when the task terminates normally, using the **exit( )** function.

Additional Information

Successive calls to **atexit( )** create a register of functions that execute in LIFO (last-in-first-out) order.

See also: exit( ), onexit( )

Returns

Value	**Meaning**
0	Successful
Not 0	Error occurred, such as 32 exit functions already defined

atof, atoi, atol

Atof( ) converts a character string to a double value; **atoi( )** converts to an integer value; **atol( )** converts to a long integer value.

Syntax

```
#include <math.h>
      #include <stdlib.h>
      double atof (const char *string);
      int atoi (const char *string);
      long atol (const char *string);
```

Parameter

`string`	A sequence of characters that represent a numerical value of the specified type. The maximum string size for **atof( )** is 100 characters.

Additional Information

These functions stop reading the input string at the first character not recognizable as part of a number. This character may be the null character `\0` terminating the string.

Atof( ) expects `string` to have this form:

```
[whitespace] [sign] {[digits]|[.digits]} [d | D | e | E[sign]digits]
```

Where:

`whitespace`	Space and/or tab characters, which are ignored.
`sign`	Either plus (+) or minus (-).
`digits`	Decimal digits. If no digits appear before the decimal point, at least one must appear after it. There may be an exponent, which is an introductory letter (d, D, e, or E) and an optionally signed integer.

Atoi( ) and **atol( )** do not recognize decimal points or exponents. The `string` argument for these functions has this form:

```
[whitespace] [sign] [digits]
```

Where `whitespace`, `sign`, and `digits` are as described for **atof( )**.

Results are undefined on overflow.

See also: ecvt( ), fcvt( ), gcvt( )

Returns

The converted value.

0 for **atoi( )**, 0L for **atol( )**, and 0.0 for **atof( )**, if the input cannot be converted to a value of the specified type.

Bessel Functions

Compute the Bessel function.

Syntax

```
#include <math.h>
      double j0 (double x);
      double j1 (double x);
      double jn (int n, double x);
      double y0 (double x);
      double y1 (double x);
      double yn (int n, double x);
```

Parameters

`x`	Value must be positive for **y0( )**, **y1( )**, and **yn( )**.
`n`	Integer order.

Additional Information

These functions are commonly used in the mathematics of electromagnetic wave theory.

See also: Mathematics reference books, such as the *Handbook of Mathematical Functions* (Abramowitz and Stegun; Washington: U.S. Government Printing Office, 1964), **matherr( )**

Returns

J0( ), **j1( )**, and **jn( )** return the result of Bessel functions of the first kind: orders 0, 1, and `n`, respectively.

Y0( ), **y1( )**, and **yn( )** return the result of Bessel functions of the second kind: orders 0, 1, and `n`, respectively. If `x` is negative, functions set **errno** to EDOM, print a DOMAIN error message to *stderr*, and return -HUGE_VAL.

This function does not return standard ANSI domain or range errors.

bsearch

Performs a binary search of a sorted array.

Syntax

```
#include <stdlib.h>
      #include <search.h>
      void *bsearch (const void *key, const void *base, size_t
                     num,size_t width, int (*compare) (const
      void
                     *elem1, const void *elem2));
```

Parameters

`key`	Value being sought.
`base`	Pointer to base of array to be searched.
`num`	Number of elements in the array.
`width`	Width of elements in bytes.

`compare`

Pointer to a user-supplied routine that compares two array elements, `elem1` and `elem2`, and returns a value specifying their relationship:

Value	**Meaning**
< 0	elem1 less than elem2
= 0	elem1 identical to elem2
> 0	elem1 greater than elem2

`elem1`	Pointer to the key for the search.
`elem2`	Pointer to the array element to be compared with the key.

Additional Information

The function calls the `compare` routine one or more times during the search, passing pointers to two array elements on each call.

If the array you are searching is not in ascending sort order, **bsearch( )** does not work properly. If the array contains duplicate records with identical keys, there is no way to predict which of the duplicate records will be located by **bsearch( )**.

See also: lfind( ), lsearch( ), qsort( )

Returns

A pointer to the first occurrence of `key` in the array pointed to by `base`.

A null pointer if a match is not found.

cabs

Calculates the absolute value of a complex number.

Syntax

```
#include <math.h>
      double_cabs(struct_complex z):
```

Parameter

`z`	Complex number.

Additional Information

The complex number `z` must be a structure of type `_complex`. The structure `z` is composed of a real component `x` and an imaginary component `y`. A call to **cabs** is equivalent to:

```
sqrt(z.x*z.x + z.y*z.y)
```

See also: abs( ), fabs( ), labs( )

Returns

On overflow, this function calls **matherr( )**, returns HUGE_VAL, and sets **errno** to ERANGE.

calloc

Allocates and clears an array in memory; initializes each element to 0.

Syntax

```
#include <stdlib.h>
      void *calloc (size_t num, size_t size);
```

Parameters

`num`	Number of elements to allocate storage space for.
`size`	Length in bytes of each element.

Additional Information

The allocated memory is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than `void`, use a type cast on the return value.

See also: free( ), malloc( ), realloc( )

Returns

A pointer to the allocated space.

ceil

Calculates the *ceiling* (the smallest integer that is greater than or equal to the value) of a double value.

Syntax

```
#include <math.h>
      double ceil (double x);
```

Parameter

x	Value to calculate ceiling for.

See also: floor( ), fmod( )

Returns

The ceiling result.

No error return.

cgets

Gets a character string from the console and stores it.

Syntax

```
#include <conio.h>
      char *cgets (char *buffer);
```

Parameter

`buffer`

Storage location for data. Must be a pointer to a character array. The first element of the array, `buffer[0]`, must contain the maximum length in characters of the string to be read. The array must contain enough elements to hold the string, a terminating null character `\0`, and two additional bytes.

Additional Information

This function continues to read characters until it reads a carriage-return line-feed (<CR><LF>) combination, or the specified number of characters. If it reads a <CR><LF> combination, it replaces the <CR><LF> with a null character `\0` before storage. The **cgets( )** function then stores the actual length of the string in the second array element, `buffer[1]`.

See also: getch( ), getche( )

Returns

A pointer to the start of the string, at `buffer[2]`.

No error return.

chmod

Changes the permission mode of a file.

Syntax

```
#include <sys/types.h>
      #include <sys/stat.h>
      #include <io.h>
      int chmod (const char *filename, mode_t pmode);
```

Parameters

`filename`
Pathname of existing file.

`pmode` New permission mode for file, which controls file ownership and access rights.

Additional Information

`Pmode` contains one or more of the manifest constants defined in <*sys/stat.h*>. The meaning of `pmode` is:

Value	**Meaning**
S_IRGRP	Read permission bit for POSIX file group
S_IROTH	Read permission bit for POSIX World owner
S_IRUSR	Read permission bit for POSIX file owner
S_IRWXG	Mask for POSIX file group
S_IRWXO	Mask for POSIX World (other) owner
S_IRWXU	Mask for POSIX file owner
S_ISGID	Set group ID on execution
S_ISUID	Set user ID on execution
S_IWGRP	Write permission bit for POSIX file group
S_IWOTH	Write permission bit for POSIX World owner
S_IWUSR	Write permission bit for POSIX file owner
S_IXGRP	Execute or search permission bit for POSIX file group
S_IXOTH	Execute or search permission bit for POSIX World owner
S_IXUSR	Execute or search permission bit for POSIX file owner

Join more than one constant with the bitwise-OR operator (`|`).

This function translates POSIX file ownership rights this way:

POSIX Owner	iRMX Owner
Owner	Owner 1 (first accessor)
Group	Owner 2 (second accessor)
World (other)	World

This function also translates POSIX access rights to the iRMX OS equivalent this way:

POSIX Access Rights	iRMX Access Rights
Read	Read
Write	Delete, Append, and Update
Execute	Ignored (no iRMX OS equivalent)

See also: creat( ), fstat( ), open( ), stat( )

Returns

Value	Meaning
0	Successful
-1	Error occurred; the function sets **errno** to ENOENT, indicating that the specified file could not be found.

chsize

Extends or truncates the size of a file to the specified length.

Syntax

```
#include <io.h>
      int chsize (int handle, long size);
```

Parameters

`handle`	Descriptor referring to an open file. The file must be open in a mode that permits writing.
`size`	New length of file in bytes.

Additional Information

If the file is extended, null characters `\0` are appended. If the file is truncated, all data from the end of the shortened file to the original length of the file is lost.

The directory update is done when a file is closed. Consequently, while a program is running, requests to determine the amount of free disk space may receive inaccurate results.

See also: close( ), creat( ), open( )

Returns

Value	**Meaning**	
0	Successful	
-1	Error occurred; the function sets **errno** to one of these values:	
	EACCES	Specified file is locked against access.
	EBAD	Specified file is read-only or an invalid file descriptor.
	ENOSPC	No space is left on device.

clearerr

Resets the error and end-of-file indicators for a stream.

Syntax

```
#include <stdio.h>
      void clearerr (FILE *stream);
```

Parameter

`stream` Pointer to `FILE` structure.

Additional Information

Once the error indicator for a specified stream is set, operations on that stream continue to return an error value. Invoke **clearerr( )** to reset the error indicator. You can also call **fseek( )**, **fsetpos( )**, or **rewind( )** to do the same thing.

See also: eof( ), feof( ), ferror( ), fseek( ), fsetpos( ), perror( ), rewind( )

Returns

Nothing.

clock

Measures the time used by the calling task, from when the calling task first began execution to the current time.

Syntax

```
#include <time.h>
      clock_t clock (void);
```

Additional Information

In the multitasking iRMX OS environment, this does not tell how much processor time has been used by the calling task.

See also: difftime( ), time( )

Returns

The product of the time in seconds and the value of the CLOCKS_PER_SEC constant. Divide the return value by the CLOCKS_PER_SEC constant to obtain the actual time.

-1, cast as `clock_t`, if unsuccessful.

close

Closes a file.

Syntax

```
#include <io.h>
      int close (int handle);
```

Parameter

`handle`	Descriptor referring to an open file.

See also: chsize( ), creat( ), dup( ), dup2( ), open( ), unlink( )

Returns

Value	**Meaning**
0	Successful
-1	Error occurred; the function sets **errno** to EBADF, indicating an invalid file descriptor argument.

closedir

Closes the directory stream associated with the directory. The directory stream descriptor directory is not available after this call.

Syntax

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dir);
```

Returns

The closedir() function returns 0 on success or -1 on failure.

EBADF	Invalid directory stream descriptor dir.
See also:	close(2), opendir(3), readdir(3), rewinddir(3)

cos, cosh

Cos calculates the cosine and **cosh** calculates the hyperbolic cosine of an angle.

Syntax

```
#include <math.h>
      double cos (double x);
      double cosh (double x);
```

Parameter

x	Angle in radians.

See also: acos( ), asin( ), atan( ), matherr( ), sin( ), tan( )

Returns

The cosine or hyperbolic cosine.

Cos( )	Returns a PLOSS error if x is large and a partial loss of significance in the result occurs; function sets **errno** to ERANGE.
	Prints a TLOSS message to *stderr* and returns 0 if x is so large that significance in the result is completely lost; function sets **errno** to ERANGE.
Cosh( )	Returns HUGE_VAL and sets **errno** to ERANGE if the result is too large.

This function does not return standard ANSI domain or range errors.

cprintf

Formats a string and prints to the console.

Syntax

```
#include <conio.h>
      int cprintf (char *format [, argument] ...);
```

Parameters

`format` Format-control string.

`argument`
Optional arguments.

Additional Information

This function uses the **putch( )** function to output characters.

Each `argument` (if any) is converted and output according to the corresponding format specification.

The `format` argument has the form and function described in the **printf( )** function.

Cprintf( ) does not translate line-feed characters into carriage-return line-feed combinations on output, unlike the **fprintf( )**, **printf( )**, and **sprintf( )** functions.

See also: fprintf( ), printf( ), sprintf( ), vprintf( )

Returns

The number of characters printed.

cputs

Writes a null-terminated string directly to the console.

Syntax

```
#include <conio.h>
      int cputs (char *string);
```

Parameter

`string`	Output string; must be null-terminated. A carriage-return line-feed (<CR><LF>) combination is not automatically appended.

See also: **putch( )**

Returns

Value	**Meaning**
0	Successful
EOF	Unsuccessful

creat

Creates a new file and opens it for writing in the specified permission mode or opens an existing file for writing and truncates it to length 0, destroying the previous contents.

Syntax

```
#include <sys/types.h>
      #include <sys/stat.h>
      #include <io.h>
      int creat (const char *filename, mode_t pmode);
```

Parameters

`filename`
Pathname of file to be opened for writing.

`pmode` Permission mode, one or more of the manifest constants described in **chmod( )**. Join multiple constants with the bitwise-OR operator (`|`). Applies to newly created files only.

Additional Information

The **creat( )** function applies the default file-permission mask (set with the **umask( )** function) to `pmode` before setting the permissions. A new file receives the specified ownership and access rights after it is closed for the first time.

By default, files opened by this function are sharable by all tasks. If O_EXCL is ORed with `pmode,` the file is opened with share-with-none permission, like UNIX.

This function translates POSIX file ownership rights and access rights as described in **chmod( )**.

See also: chmod( ), chsize( ), close( ), dup( ), dup2( ), open( ), sopen( ), umask( )

Returns

A descriptor for the created file.

-1 and sets **errno** to one of these values if unsuccessful:

EACCES	Pathname specifies an existing read-only file or specifies a directory instead of a file.
EMFILE	No more file descriptors available (too many open files).

ENOENT	Pathname not found.

cscanf

Reads formatted data from the console into the specified locations.

Syntax

```
#include <conio.h>
      int cscanf (char *format [, argument] ...);
```

Parameters

`format` Format-control string.

`argument`
Optional arguments; must be a pointer to a variable with a type that corresponds to a type specifier in `format`.

Additional Information

The `format` controls the interpretation of the input fields and has the same form and function as described in **scanf( )**.

While **cscanf( )** normally echoes the input character, it does not if the last call was to **ungetch( )**.

This function uses **getche( )** to read characters.

See also: fscanf( ), scanf( ), sscanf( )

Returns

The number of fields that were successfully converted and assigned; does not include fields that were read but not assigned.

0 if no fields were assigned.

EOF for an attempt to read at end-of-file. This may occur when keyboard input is redirected at the operating system command-line level.

_cstop

Deletes the C resources allocated for a task.

Syntax

```
#include <rmx_c.h>
      _cstop (selector task_t);
```

Parameter

`task_t` iRMX task token; 0 indicates remove the current task. If the task to be removed is not the current task, it must not be using C library functions when you remove it.

Additional Information

Applications that dynamically create and delete C tasks should call **rq_suspend_task**, then **_cstop( )** before deleting a task using **rq_delete_task**. The deleted C resources for the task include connections to *stdin*, *stdout*, *stderr*, the C library information structure `CINFO_STRUCT`, and other bookkeeping segments.

Each C task maintains its own resources. The minimum resources assigned to each task consist of `CINFO_STRUCT` and two synchronization semaphores for the task. These are allocated on the first call to any C library function by the task. A task can obtain the data in `CINFO_STRUCT` with the **_get_info( )** function. The C task resources also include storage space for the task's context, and a temporary storage area for information pushed onto the stack by the C library.

Additional resources are established for a task on the first call to any *stdio* function. These are:

- Additional bookkeeping area for `CINFO_STRUCT` (about 400 bytes)
- Connections to *stdin*, *stdout*, and *stderr*, along with two I/O synchronization mailboxes and one synchronization semaphore for each mailbox
- Two 512-byte I/O buffers, one each for *stdin* and *stdout*, allocated from the job heap using **malloc( )**

Any **malloc( )** segments and the **malloc( )** mutual exclusion semaphore are not deleted until the parent job is deleted, since they are global to the parent job.

Minimize the total amount of resources required by an application by dynamically creating and deleting tasks that call *stdio* functions.

See also: exit( ), _get_info( ), malloc( ), <*rmx_c.h*>, stat( )

Returns

Nothing.

cstr, udistr

Cstr converts a count-prefixed iRMX-style string to a null-terminated C-style string and stores it. **Udistr( )** converts a null-terminated C-style string to a count-prefixed iRMX-style string and stores it.

Syntax

```
#include <string.h>
      char *cstr (char *c_str, const char *udi_str);
      char *udistr (char *udi_ptr, const char *c_ptr);
```

Parameters

`c_str`	Pointer to a null-terminated (C convention) string.
`udi_str`	Pointer to a count-prefixed (iRMX convention) string.

Additional Information

The string buffer for **cstr( )** must be large enough to hold the string and the null character `\0` string terminator. Since count-prefixed strings are restricted to 0 to 255 characters (range of the one-byte count), plus the terminating null character, the string buffer can be 1 to 256 bytes long.

The string buffer for **udistr( )** must be large enough to hold the string and the leading one-byte length field for the count. Use **strlen( )** to determine the required length of the destination buffer. The buffer must be one byte longer than the value returned by **strlen**, since it returns the number of characters in the string excluding the terminating null character `\0`. The behavior of **udistr( )** for strings longer than 255 bytes is undefined.

The two pointers `c_ptr` and `udi_ptr` normally point to separate string buffers. However, if the arguments are identical, **udistr( )** and **cstr( )** still work correctly, converting the indicated string in place.

See also: **strlen( )**, *<udi_c.h>*

Returns

A pointer to the converted string.

No error return.

ctime

Converts a time stored as a `time_t` value to a character string.

Syntax

```
#include <time.h>
      char *ctime (const time_t *timer);
```

Parameter

`timer`	Stored time value to convert, usually obtained from a call to **time( )**.

Additional Information

The converted string contains exactly 26 characters and has this form:

```
Wed Jan 02 02:03:55 1980\n\0
```

All elements have a constant width. The newline character `\n` and the null character `\0` occupy the last two positions of the string.

A 24-hour clock is used.

Calls to the **ctime( )** function modify the single statically allocated buffer used by the **gmtime( )** and the **localtime( )** functions. Each call to one of these functions destroys the result of the previous call.

The **ctime( )** function also shares a static buffer with the **asctime( )** function. Thus, a call to **ctime( )** destroys the results of any previous call to **asctime( )**, **localtime( )**, or **gmtime( )**.

See also: asctime( ), gmtime( ), localtime( ), time( )

Returns

A pointer to the character string.

A null pointer if time represents a date before epoch time.

difftime

Finds the difference between two time values.

Syntax

```
#include <time.h>
      double difftime (time_t timer1, time_t timer0);
```

Parameters

`timer0`	Beginning time.
`timer1`	Ending time.

See also: **time( )**

Returns

The elapsed time in seconds.

div

Divides the numerator by the denominator, computing the quotient and the remainder of two integer values.

Syntax

```
#include <stdlib.h>
      div_t div (int numer, int denom);
```

Parameters

`numer`	Numerator.
`denom`	Denominator. If 0, the program will terminate with an error message.

See also: **ldiv( )**

Returns

A `div_t` structure, described in <*stdlib.h*>.

The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer that is less than the absolute value of the mathematical quotient.

_dos_allocmem

Allocates a block of memory.

Syntax

```
#include <dos.h>
      unsigned _dos_allocmem (unsigned size, unsigned *seg);
```

Parameters

`size`	Block size to allocate in paragraphs (16-byte units).
`seg`	Pointer to where segment token is returned.

Additional Information

Allocated blocks are always paragraph aligned. The memory heap is not used.

An iRMX segment is always created. This applies to all memory models, including 32-bit flat.

If the request cannot be satisfied, the maximum possible size (in paragraphs) is returned instead.

See also: calloc( ), _dos_freemem( ), malloc( )

Returns

Value	Meaning
0	Successful
-1	Error occurred; the function sets **errno** to ENOMEM, indicating insufficient memory.

_dos_close

Closes a file.

Syntax

```
#include <dos.h>
      unsigned _dos_close (int handle);
```

Parameter

`handle`	Target file to close (`handle` was returned by the call that created or last opened the file).

Additional Information

See also:	close( ), creat( ), _dos_creat, _dos_open( ), _dos_read( ), _dos_write( ), dup( ), open( )

Returns

Value	**Meaning**
0	Successful
-1	Error occurred; the function sets **errno** to EBADF, indicating an invalid file handle.

_dos_creat, _dos_creatnew

These functions create and open a new file with the specified access attributes.

Syntax

```
#include <dos.h>
      unsigned _dos_creat (const char *filename, unsigned
      attrib,
                           int *handle);
      unsigned _dos_creatnew (const char *filename, unsigned
      attrib,
                              int *handle);
```

Parameters

`filename`
File pathname.

`attrib` File attributes.

`handle` Pointer to handle return buffer. The new file's handle is copied into the location `handle` points to.

Additional Information

The file is opened for both read and write access. If file sharing is installed, the file is opened in compatibility mode.

The **_dos_creat( )** function erases an existing file's contents and leaves its attributes unchanged.

The **_dos_creatnew( )** function fails if the file already exists.

Returns

Value	**Meaning**	
0	Successful	
-1	Error occurred; the function sets **errno** to one of these values:	
	EACCES	Access denied because the directory is full or, for **_dos_creat( )** only, the file exists and cannot be overwritten.
	EEXIST	File already exists (**_dos_creatnew( )** only).
	EMFILE	Too many open file handles.

ENOENT	Path or file not found.

_dos_findfirst, _dos_findnext

_dos_findfirst finds the first file with the specified name and attributes; **_dos_findnext** finds the next file.

Syntax

```
#include <dos.h>
      unsigned _dos_findfirst(const char *filename, unsigned
      attrib,
                              struct find_t *fileinfo );
      unsigned _dos_findnext(struct find_t *fileinfo);
```

Parameters

`filename`
Target filename; may use wildcards * and ?.

`attrib` Target file attributes.

`fileinfo`
Pointer to file-information buffer.

Additional Information

The `attrib` argument can be any of these manifest constants:

_A_NORMAL	Normal. File can be read or written without restriction.
_A_RDONLY	Read-only. File cannot be opened for writing, and a file with the same name cannot be created. Returns information about normal files as well as about files with this attribute.
_A_SUBDIR	Subdirectory. Returns information about normal files as well as about files with this attribute.

Combine multiple constants with the bitwise-OR operator (`|`).

If the `attrib` argument to either of these functions is _A_RDONLY or _A_SUBDIR, the function also returns any normal attribute files that match the `filename` argument; a normal file does not have a read-only or directory attribute.

Information is returned in a `find_t` structure, defined in <*dos.h*>.

The time format is:

Time Bits	**Contents**
0-4	Number of 2-second increments (0-29)
5-10	Minutes (0-59)
11-15	Hours (0-23)

The date format is:

Date Bits	**Contents**
0-4	Day of month (1-31)
5-8	Month (1-12)
9-15	Year (relative to 1980)

Do not alter the contents of the `fileinfo` buffer between a call to **_dos_findfirst( )** and all subsequent calls to the **_dos_findnext( )** function.

The **_dos_findnext( )** function finds the next name, if any, that matches the arguments specified in a prior call to **_dos_findfirst( )**. The `fileinfo` argument must point to a `find_t` structure initialized by a previous call to **_dos_findfirst( )**. The contents of the structure will be altered as described if a match is found.

Returns

Value	**Meaning**
0	Successful
-1	Error occurred; the function sets **errno** to ENOENT, indicating that the filename could not be matched.

_dos_freemem

Releases a block of memory previously allocated by **_dos_allocmem( )**.

Syntax

```
#include <dos.h>
      unsigned _dos_freemem (unsigned seg);
```

Parameter

`seg`	Block to be released, a value returned by a previous call to **_dos_allocmem( )**.

Additional Information

The freed memory can no longer be used by the application program.

See also: _dos_allocmem( ), free( )

Returns

Value	**Meaning**
0	Successful
-1	Error occurred; the function sets **errno** to ENOMEM, indicating a bad offset value (one that does not correspond to an offset returned by a previous **_dos_allocmem( )** call) or invalid arena headers.

_dos_getdate

Gets the current system date.

Syntax

```
#include <dos.h>
      void _dos_getdate (struct dosdate_t *date);
```

Parameter

`date`	Current system date.

Additional Information

The date is returned in a `dosdate_t` structure, defined in *<dos.h>*.

See also: _dos_gettime( ), _dos_setdate( ), _dos_settime( ), gmtime( ), localtime( ), mktime( ), time( )

Returns

Nothing.

_dos_getftime

Gets the date and time that a file was last written.

Syntax

```
#include <dos.h>
      unsigned _dos_getftime (int handle, unsigned *date,
                              unsigned *time);
```

Parameters

`handle`	Target file; the file must be opened with a call to **_dos_open( )** or **_dos_creat( )**.
`date`	Date-return buffer.
`time`	Time-return buffer.

Additional Information

The date and time are returned in the words pointed to by `date` and `time`. The values appear in the DOS date and time format as described in **_dos_findfirst**.

See also: fstat( ), stat( )

Returns

Value	**Meaning**
0	Successful
-1	Error occurred; the function sets **errno** to EBADF, indicating that an invalid file handle was passed.

_dos_gettime

Gets the current system time.

Syntax

```
#include <dos.h>
      void _dos_gettime (struct dostime_t *time);
```

Parameter

`time`	Current system time.

Additional Information

The time is returned in a `dostime_t` structure, defined in *<dos.h>*.

See also: _dos_getdate( ), _dos_setdate( ), _dos_settime( ), gmtime( ), localtime( )

Returns

Nothing.

_dos_open

Opens an existing file.

Syntax

```
#include <dos.h>
      #include <fcntl.h>
      #include <share.h>
      unsigned _dos_open (const char *filename, unsigned mode,
                          int *handle);
```

Syntax

Parameters

`filename`
Path to an existing file.

`mode` Specifies the file's access, sharing, and inheritance permissions.

`handle` Pointer to the handle for the opened file.

Additional Information

The `mode` argument specifies the file's access, sharing, and inheritance modes by combining (with the OR operator) manifest constants from the three groups shown below. At most, one access mode and one sharing mode can be specified at a time.

Constant	**Mode**	**Meaning**
O_RDONLY	Access	Read-only
O_WRONLY	Access	Write-only
O_RDWR	Access	Both read and write
SH_COMPAT	Sharing	Compatibility
SH_DENYRW	Sharing	Deny reading and writing
SH_DENYWR	Sharing	Deny writing
SH_DENYRD	Sharing	Deny reading
SH_DENYNO	Sharing	Deny neither
O_NOINHERIT	Inheritance	File is not inherited by the child process

See also: _dos_close( ), _dos_read( ), _dos_write( )

Returns

Value	Meaning
0	Successful
-1	Error occurred; the function sets **errno** to one of these:

E	Access denied (possible reasons include specifying a directory or volume ID for filename, or opening a read-only file for write access).
E	Sharing mode specified when file sharing not installed, or access-mode value is invalid.
E	Too many open file handles.
E	Path or file not found.

_dos_read

Reads a specified number of bytes of data from a file.

Syntax

```
#include <dos.h>
      unsigned _dos_read (int handle, void *buffer, unsigned
      count,
                          unsigned *actual);
```

Parameters

`handle`	File to read.
`buffer`	Pointer to buffer to receive data.
`count`	Number of bytes to read.
`actual`	Pointer to the number of bytes actually read, which may be less than the number requested.

Additional Information

If the number of bytes actually read is 0, it means the function tried to read at end-of-file.

See also: _dos_close( ), _dos_open( ), _dos_write( ), read( )

Returns

Value	**Meaning**
0	Successful
-1	Error occurred; the function sets **errno** to one of these:

E	Access denied (handle is not open for read access).

E	File handle is invalid.

_dos_setdate

Sets the current system date.

Syntax

```
#include <dos.h>
      unsigned _dos_setdate (struct dosdate_t *date);
```

Parameter

`date`	New system date.

Additional Information

The date is stored in the `dosdate_t` structure, defined in <*dos.h*>.

See also: _dos_gettime( ), _dos_setdate( ), _dos_settime( ), gmtime( ), localtime( ), mktime( ), time( )

Returns

Value	**Meaning**
0	Successful
Not 0	Error occurred; the function sets **errno** to EINVAL, indicating an invalid date was specified.

_dos_setftime

Sets the date and time that a file was last written.

Syntax

```
#include <dos.h>
      unsigned _dos_setftime (int handle, unsigned date,
                              unsigned time);
```

Parameters

`handle`	Target file
`date`	Date of last write
`time`	Time of last write

Additional Information

Sets the `date` and `time` at which the file identified by `handle` was last written to. These values appear in the DOS date and time format:

Time Bits	**Meaning**
0-4	Number of two-second increments (0-29)
5-10	Minutes (0-59)
11-15	Hours (0-23)
Date Bits	**Meaning**
0-4	Day (1-31)
5-8	Month (1-12)
9-15	Year since 1980 (for example, 1989 is stored as 9)

See also: _dos_getftime( ), fstat( ), stat( )

Returns

Value	**Meaning**
0	Successful
Not 0	Not successful; function sets **errno** to EBADF, indicating that an invalid file handle was passed.

_dos_settime

Sets the current system time.

Syntax

```
#include <dos.h>
      unsigned _dos_settime (struct dostime_t *time);
```

Parameter

`time`	New system time.

Additional Information

Sets the current system time to the value stored in the `dostime_t` structure that `time` points to, as defined in *<dos.h>*.

See also: _dos_getdate( ), _dos_gettime( ), _dos_setdate( ), gmtime( ), localtime( ), mktime( )

Returns

Value	**Meaning**
0	Successful
Not 0	Error occurred; the function sets **errno** to EINVAL, indicating an invalid time was specified.

_dos_write

Writes a specified number of bytes from a buffer to a file.

Syntax

```
#include <dos.h>
      unsigned _dos_write (int handle, void const *buffer,
      unsigned
                          count, unsigned *actual);
```

Parameters

`handle`	File to write to.
`buffer`	Pointer to buffer to write from.
`count`	Number of bytes to write.
`actual`	Pointer to the number of bytes actually written, which can be less than the number requested.

Additional Information

See also: _dos_close( ), _dos_open( ), _dos_read( ), write( )

Returns

Value	Meaning
0	Successful
-1	Error occurred; the function sets **errno** to one of these:

E	Access denied (`handle` references a file not open for write access).
E	Invalid file handle.

dup, dup2

Dup creates a second file descriptor for an open file in the running task's file descriptor table and **dup2** reassigns a file descriptor in the table.

Syntax

```
#include <io.h>
      int dup (int handle);
      int dup2 (int handle1, int handle2);
```

Parameters

`handle, handle1`
: Descriptor referring to an open file.

`handle2`
: Any file descriptor value.

Additional Information

Operations on the file can be carried out using either the old or new file descriptor. The type of access allowed for the file is unaffected by the creation of a new file descriptor.

The **dup2( )** function forces `handle2` to refer to the same file as `handle1`. If `handle2` is associated with an open file at the time of the call, that file is closed.

The C library keeps track of the number of duplications on a file connection. The original connection will remain valid until the last duplication is closed or deleted.

See also: close( ), creat( ), open( )

Returns

Dup( ) returns the next available file descriptor for the file.

Dup2( ) returns 0 to indicate success.

Both functions return -1 if an error occurs and set **errno** to one of these values:

EBADF	Invalid file descriptor.
EMFILE	No more file descriptors available (too many open files).

ecvt

Converts a value to a character string.

Syntax

```
#include <stdlib.h>
      char *ecvt (double value, int count, int *dec, int
      *sign);
```

Parameters

`value`	Value to convert.
`count`	Number of digits stored as a string. The function appends a null character `\0`.
`dec`	Points to an integer value giving the position of the decimal point with respect to the beginning of the string. A 0 or negative integer value indicates that the decimal point lies to the left of the first digit.
`sign`	Points to an integer indicating the sign of the converted number.

Value	**Meaning**
0	Positive
Not 0	Negative

Additional Information

Only digits are stored in the string. If the number of digits in `value` exceeds `count`, the low-order digit is rounded. If there are fewer than `count` digits, the string is padded with 0s.

Obtain the position of the decimal point and the sign of `value` from `dec` and `sign` after the call.

This function uses a single statically allocated buffer for the conversion. Subsequent calls overwrite the result.

See also: atof( ), atoi( ), atol( ), fcvt( ), gcvt( )

Returns

A pointer to the string of digits.

No error return.

eof

Checks whether the file's current file pointer is EOF.

Syntax

```
#include <io.h>
      int eof (int handle);
```

Parameter

`handle`	Descriptor referring to an open file.

See also: clearerr( ), feof( ), ferror( ), perror( )

Returns

Value	**Meaning**
1	Current position is end-of-file
0	Current position is not end-of-file
-1	Error occurred; the function sets **errno** to EBADF, indicating an invalid file descriptor

exit, _exit

Exit( ) terminates the calling task after cleanup and **_exit( )** terminates it immediately.

Syntax

```
#include <process.h>  /* for _exit( ) */
      #include <stdlib.h>
      void exit (int status);
      void _exit (int status);
```

Parameter

`status`	Exit status.

Additional Information

Exit( ) performs complete C library termination procedures. It calls the functions registered by **atexit( )** and **onexit( )** in LIFO order. It flushes all file buffers before terminating the task and exits with the supplied status code.

_exit( ) performs quick C library termination procedures by invoking **rq_exit_io_job**. It terminates the task, and informs the parent job with the supplied status code. Typically, it sets `status` to 0 to indicate a normal exit or to some other value to indicate an error.

_exit( ) does not process **atexit( )** or **onexit( )** functions or flush stream buffers.

See also: abort( ), atexit( ), onexit( ),
rq_exit_io_job, *System Call Reference*

Returns

Nothing.

exp

Calculates the exponential of a double value.

Syntax

```
#include <math.h>
      double exp (double x);
```

Parameter

x	Value to calculate exponential for.
	See also: **log( )**

Returns

The exponential function, e^x.

HUGE_VAL on overflow, and the function sets **errno** to ERANGE.

0 on underflow, but the function does not set **errno**.

This function does not return standard ANSI domain or range errors.

fabs

Calculates the absolute value of a double value.

Syntax

```
#include <math.h>
      double fabs (double x);
```

Parameter

`x` Value to calculate absolute value for.

See also: abs( ), labs( ), cabs( )

Returns

The absolute value.

No error return.

fclose, fcloseall

Fclose closes a specified stream and **fcloseall** closes all open streams.

Syntax

```
#include <stdio.h>
      int fclose (FILE *stream);
      int fcloseall (void);
```

Parameter

`stream` Pointer to `FILE` structure.

Additional Information

The **fcloseall( )** function closes all open streams except *stdin*, *stdout*, and *stderr*. It also closes and deletes any temporary files created by **tmpfile( )**.

In both functions, all buffers associated with the stream are flushed prior to closing. System-allocated buffers are released when the stream is closed. Buffers assigned by the user with **setbuf( )** and **setvbuf( )** are not automatically released.

See also: close( ), fdopen( ), fopen( ), freopen( )

Returns

Fclose( ) returns 0 if successful.

Fcloseall( ) returns the total number of streams closed.

Both functions return EOF to indicate an error.

fcvt

Converts a double value to a null-terminated string, indicating the sign and decimal point location.

Syntax

```
#include <stdlib.h>
      char *fcvt (double value, int count, int *dec, int
      *sign);
```

Parameters

`value`	Value to convert. Stores the digits of `value` as a string and appends a null character `\0`.
`count`	Number of digits to store after decimal point. Excess digits are rounded off to `count` places. If there are fewer than `count` digits, the string is padded with 0s.
`dec`	Points to an integer value, which gives the position of the decimal point with respect to the beginning of the string. A 0 or negative integer value indicates that the decimal point lies to the left of the first digit.
`sign`	Points to an integer indicating the sign of `value`.

Value	**Meaning**
0	Positive
Not 0	Negative

Additional Information

Only digits are stored in the string. Obtain the position of the decimal point and the sign of `value` from `dec` and `sign` after the call.

The **fcvt( )** function uses a single statically allocated buffer for the conversion. Each call destroys the results of the previous call.

See also: atof( ), atoi( ), atol( ), ecvt( ), gcvt( )

Returns

A pointer to the string of digits.

No error return.

fdopen

Opens a stream associated with a file descriptor, allowing a file opened for low-level I/O to be buffered and formatted.

Syntax

```
#include <stdio.h>
      FILE *fdopen (int handle, char *mode);
```

Parameters

`handle`	Descriptor referring to an open file.
`mode`	Specifies the open mode (type of access permitted) for the file.

Additional Information

This list gives the `mode` string, including required quotes, as used in the **fopen( )** and **fdopen( )** functions. It also relates the `mode` string and the corresponding `oflag` arguments used in the **open( )** and **sopen( )** functions.

Value	**Meaning**
"r"	Opens for reading. If the file does not exist or cannot be found, the call will fail. Relates to O_RDONLY.
"w"	Opens an empty file for writing. If the given file exists, its contents are destroyed. Relates to O_WRONLY (usually O_WRONLY | O_CREAT | O_TRUNC).
"a"	Opens for writing at the end of the file (appending); creates the file first if it doesn't exist. Relates to O_WRONLY | O_APPEND (usually O_WRONLY | O_CREAT | O_APPEND).
"r+"	Opens for both reading and writing. The file must exist. Relates to O_RDWR.
"w+"	Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed. Relates to O_RDWR (usually O_RDWR | O_CREAT | O_TRUNC).
"a+"	Opens for reading and appending; creates the file first if it doesn't exist. Relates to O_RDWR | O_APPEND (usually O_RDWR | O_APPEND | O_CREAT)

Use the `"w"` and `"w+"` types with care, as they can destroy existing files.

When a file is opened with the `"a"` or `"a+"` open mode, all write operations occur at the end of the file, even if you've repositioned the file pointer using **fseek( )** or **rewind( )**. Thus, existing data cannot be overwritten.

When the `"r+"`, `"w+"`, or `"a+"` open mode is specified, both reading and writing are allowed (the file is open for update). However, when you switch between reading and writing, there must be an intervening **rewind( )** operation or **fsetpos( )** or **fseek( )**, which can reposition the file pointer, if desired.

In addition to these values, one of these characters can be included after `mode` but between the quotation marks to specify the translation mode for <LF> characters. The `t` and `b` characters correspond to the constants used in the **open( )** and **sopen( )** functions, as listed below.

Value	**Meaning**
t	Open in text (translated) mode. <CR><LF> combinations are translated into single <LF> characters on input and <LF> characters are translated to <CR><LF> combinations on output. <Ctrl-Z> is interpreted as an end-of-file character on input. In files opened for reading or for reading/writing, checks for and removes <Ctrl-Z> if possible, because <Ctrl-Z> may cause **fseek( )** to behave improperly near the end of the file. Relates to O_TEXT.
b	Open in binary (untranslated) mode; the above translations are suppressed. Relates to O_BINARY.

If `t` or `b` is not given in the `mode` string, the translation mode is defined by the default-mode variable `_fmode`, contained in *<stdlib.h>*.

The `t` option is not part of the ANSI standard for **fopen( )** and **fdopen( )**; do not use it where ANSI portability is desired.

See also: fopen( ), fclose( ), fcloseall( ), freopen( ), open( )

Returns

A pointer to the open stream.

A null pointer on error, such as `t` or `b` appearing before `mode`.

feof

Tests for end-of-file on a stream.

Syntax

```
#include <stdio.h>
      int feof (FILE *stream);
```

Parameter

`stream` Pointer to `FILE` structure.

Additional Information

Once end-of-file is reached, read operations return an end-of-file indicator until the stream is closed or until **rewind( )**, **fsetpos( )**, **fseek( )**, or **clearerr( )** is called. **Feof( )** is implemented as a macro.

See also: clearerr( ), eof( ), ferror( ), perror( )

Returns

Value	**Meaning**
0	The current position is not end-of-file
Not 0	This is the first read operation that attempted to read past end-of-file

No error return.

ferror

Tests for a read or write error on a stream.

Syntax

```
#include <stdio.h>
      int ferror (FILE *stream);
```

Parameter

`stream`	Pointer to `FILE` structure.

Additional Information

If an error occurred, the error indicator for the stream remains set until the stream is closed or rewound, or until **clearerr( )** is called. **Ferror( )** is implemented as a macro.

See also: clearerr( ), eof( ), feof( ), fopen( ), perror( )

Returns

Value	**Meaning**
0	Successful
Not 0	Error occurred

fflush

Flushes a buffered stream (has no effect on an unbuffered stream).

Syntax

```
#include <stdio.h>
      int fflush (FILE *stream);
```

Parameter

`stream` Pointer to `FILE` structure.

Additional Information

If the file associated with `stream` is open for output, **fflush( )** writes the contents of the buffer to the file. If `stream` is open for input, **fflush( )** clears the contents of the buffer.

The stream remains open after the call.

Buffers are automatically flushed when they are full, when `stream` is closed, or when a program terminates normally without closing `stream`.

See also: fclose( ), flushall( ), setbuf( )

Returns

Value	**Meaning**
0	Buffer successfully flushed or `stream` has no buffer or `stream` is open for reading only
EOF	Error occurred

fgetc, fgetchar

Fgetc( ) reads a single character from the current position of the specified stream and increments the file pointer to the next character; **fgetchar( )** reads from *stdin*.

Syntax

```
#include <stdio.h>
      int fgetc (FILE *stream);
      int fgetchar (void);
```

Parameter

`stream` Pointer to `FILE` structure.

Additional Information

The **fgetchar( )** function is equivalent to

```
fgetc (stdin)
```

Fgetc( ) and **fgetchar( )** are identical to **getc( )** and **getchar( ),** but they are functions, not macros.

See also: fputc( ), fputchar( ), getc( ), getchar( )

Returns

The integer value of the character read.

EOF on error or end-of-file. Since EOF is a legal integer value, use **feof( )** or **ferror( )** to distinguish between an error and an end-of-file condition.

fgetpos

Gets a stream's file pointer position-indicator and stores it. This function does not get the file pointer; use the **ftell( )** function instead.

Syntax

```
#include <stdio.h>
      int fgetpos (FILE *stream, fpos_t *pos);
```

Parameters

`stream`	Pointer to `FILE` structure.
`pos`	File pointer position-indicator storage.

Additional Information

The file pointer position-indicator value is stored in `fpos_t` format, which is used only by the **fgetpos( )** and **fsetpos( )** functions. The **fsetpos( )** function can use information stored in `pos` to reset the file pointer for `stream` to its position at the time **fgetpos( )** was called.

See also: **fsetpos( )**

Returns

Value	**Meaning**	
0	Successful	
Not 0	Error occurred; the function sets **errno** to one of these values:	
	EBADF	The specified stream is not a valid file descriptor or is not accessible.
	EINVAL	The stream value is invalid.

fgets

Reads a specified number of characters from a stream and stores them in a string.

Syntax

```
#include <stdio.h>
      char *fgets (char *string, int n, FILE *stream);
```

Parameters

`string`	Storage location for data. The newline character, if read, is included in the string. A null character `\0` is appended.
`n`	Number of characters stored. If `n` is 1, `string` is empty.
`stream`	Pointer to `FILE` structure.

Additional Information

Characters are read from the current stream position up to and including the first newline character `\n`, up to the end of the stream, or until the number of characters read is `n-1`, whichever comes first.

The **fgets( )** function is similar to the **gets( )** function; however, **gets( )** replaces the newline character with a null character.

See also: fputs( ), gets( ), puts( )

Returns

Returns the string.

A null pointer on error or end-of-file. Use **feof( )** or **ferror( )** to determine whether an error occurred.

filelength

Gets the length in bytes of a file.

Syntax

```
#include <io.h>
      long filelength (int handle);
```

Parameter

`handle` Descriptor referring to an open file, as returned by **creat( )** or **open( )**.

See also: chsize( ), creat( ), fileno( ), fstat( ), open( ), stat( )

Returns

The file length in bytes.

-1 on error. An invalid descriptor also sets **errno** to EBADF.

fileno

Gets the file descriptor associated with a stream.

Syntax

```
#include <stdio.h>
      int fileno (FILE *stream);
```

Parameter

`stream` Pointer to `FILE` structure.

Additional Information

This function lets you use the file descriptor I/O calls on streams; for example, **read( )**, **write( )**, and **lseek( )**. To mix the two I/O systems (**open( )** vs. **fopen( )**, **read( )** vs. **fread( )**, etc.), flush all I/O buffers when going from the buffered system (for example, **fwrite( )**) to the unbuffered system (for example, **write( )**). Otherwise, you are likely to lose data.

Fileno( ) automatically flushes the I/O buffers for the given stream.

See also: fdopen( ), filelength( ), fopen( ), freopen( )

Returns

The file descriptor currently associated with the stream. The result is undefined if `stream` does not specify an open file.

No error return.

floor

Calculates the *floor* (largest integer that is less than or equal to a value) of a double value.

Syntax

```
#include <math.h>
      double floor (double x);
```

Parameter

x	Value to calculate the floor for.

See also: ceil( ), fmod( )

Returns

The floor result.

No error return.

flushall

Writes the contents of all buffers associated with open output streams to their associated files.

Syntax

```
#include <stdio.h>
      int flushall (void);
```

Additional Information

Clears all input stream buffers of their current contents. All streams remain open after the call. The next read operation reads new data into the buffers.

Buffers are automatically flushed when they are full, when streams are closed, or when a program terminates normally without closing streams.

See also: **fflush( )**

Returns

The number of open streams (input and output).

No error return.

fmod

Calculates the floating-point remainder.

Syntax

```
#include <math.h>
      double fmod (double x, double y);
```

Parameters

x, y	Floating-point values.

Additional Information

Calculates f of x / y such that:

```
x = i * y + f
```

Where:

i	An integer.
f	The floating-point remainder. f has the same sign as x, and the absolute value of f is less than the absolute value of y.
See also:	ceil( ), fabs( ), floor( )

Returns

The remainder.

0 if y is 0.

This function does not return standard ANSI domain or range errors.

fopen

Opens a file with the specified open mode.

Syntax

```
#include <stdio.h>
      FILE *fopen (const char *filename, const char *mode);
```

Parameters

`filename`
Pathname of file.

`mode` Specifies the open mode (type of access permitted) for the file.

Additional Information

The character string `mode`, with required quotes, specifies the open mode for the file, as described in **fdopen( )**.

See also: fdopen( ), fclose( ), fcloseall( ), ferror( ), fileno( ), freopen( ), open( ), setmode( )

Returns

A pointer to the open file.

A null pointer on error.

fprintf

Prints formatted data to a stream.

Syntax

```
#include <stdio.h>
      int fprintf (FILE *stream, const char *format
                   [, argument]...);
```

Parameters

`stream`	Pointer to `FILE` structure.
`format`	Formatted string consisting of ordinary characters, escape sequences, and (if arguments follow) format specifications.
`argument`	Optional arguments.

Additional Information

The ordinary characters and escape sequences are copied to `stream` in order of their appearance.

The `format` and optional arguments have the same form and function as described in the **printf( )** function.

See also: fscanf( ), printf( ), sprintf( )

Returns

The number of characters printed.

A negative value on output error.

fputc, fputchar

Fputc writes a single character to an output stream at the current position; **fputchar** writes to *stdout*.

Syntax

```
#include <stdio.h>
      int fputc (int c, FILE *stream);
      int fputchar (int c);
```

Parameters

`c`	Character to be written.
`stream`	Pointer to `FILE` structure.

Additional Information

The **fputchar( )** function is equivalent to

```
fputc (c, stdout)
```

Fputc( ) and **fputchar( )** are similar to **putc( )** and **putchar( )**, but are functions rather than macros.

See also: fgetc( ), fgetchar( ), putc( ), putchar( )

Returns

The character written.

EOF on error. Since EOF is a legal integer value, use **ferror( )** to check for an actual error.

fputs

Writes a string to the stream at the current file pointer.

Syntax

```
#include <stdio.h>
      int fputs (const char *string, FILE *stream);
```

Parameters

`string`	String to be output. The terminating null character `\0` is not copied.
`stream`	Pointer to `FILE` structure.

See also: fgets( ), gets( ), puts( )

Returns

Value	**Meaning**
0	Successful
EOF	Unsuccessful

fread

Reads up to the specified number of items of the specified size from the input stream and stores them in a buffer.

Syntax

```
#include <stdio.h>
      size_t fread (void *buffer, size_t size, size_t count,
                    FILE *stream);
```

Parameters

`buffer`	Storage location for data.
`size`	Item size in bytes.
`count`	Maximum number of items to be read.
`stream`	Pointer to `FILE` structure.

Additional Information

The file pointer associated with `stream` (if there is one) is increased by the number of bytes actually read.

If the stream is opened in text mode, <CR><LF> pairs are replaced with single <LF> characters. The replacement has no effect on the file pointer or the return value.

The file pointer is indeterminate if an error occurs. The value of a partially read item cannot be determined.

See also: fwrite( ), read( )

Returns

The number of full items actually read, which may be less than `count` if an error occurs, if the end-of-file is encountered before reaching `count`, or if <CR>s were removed.

0 and the buffer contents are unchanged if `size` or `count` is 0.

0 on error. Use the **feof( )** or **ferror( )** function to distinguish a read error from an end-of-file condition.

free

Deallocates a memory block.

Syntax

```
#include <stdlib.h>
      void free (void *memblock);
```

Parameter

`memblock`

Points to a memory block previously allocated through a call to **calloc( )**, **malloc( )**, or **realloc( )**.

Additional Information

The number of bytes freed is the number of bytes specified when the block was allocated, or reallocated, in the case of **realloc( )**. After the call, the freed block is available for allocation.

Attempting to free a memory block not allocated with the appropriate call (such as the **sbrk( )** function) may affect subsequent allocation and cause errors.

See also: calloc( ), malloc( ), realloc( ), sbrk( )

Returns

Nothing.

freopen

Closes the file currently associated with a stream and reassigns a new file to the stream.

Syntax

```
#include <stdio.h>
      FILE *freopen (const char *filename, const char *mode,
                     FILE *stream);
```

Parameters

`filename`
Pathname of new file.

`mode` Open mode for the new file.

`stream` Pointer to `FILE` structure.

Additional Information

The **freopen( )** function is typically used to redirect *stdin*, *stdout*, and *stderr* to user-specified files.

The `mode` parameter is as described in **fdopen( )**.

See also: fclose( ), fcloseall( ), fdopen( ), fileno( ), fopen( ), open( ), setmode( )

Returns

A pointer to the newly opened file.

A null pointer value on error and the original file is closed.

frexp

Gets the mantissa and exponent of a double value.

Syntax

```
#include <math.h>
      double frexp (double x, int *expptr);
```

Parameters

`x`	Value to find exponent for.
`expptr`	Pointer to stored integer exponent `n`.

Additional Information

Breaks down the value `x` into a mantissa `m` and an exponent `n`, such that the absolute value of `m` is greater than or equal to 0.5 and less than 1.0, and $x = m * 2^{n}$.

See also: ldexp( ), modf( )

Returns

The mantissa value.

0 for both the mantissa and the exponent if `x` is 0.

No error return.

fscanf

Reads and formats character data from the current position of a stream into the specified locations.

Syntax

```
#include <stdio.h>
      int fscanf (FILE *stream, const char *format [,
      argument]...);
```

Parameters

`stream` Pointer to `FILE` structure.

`format` Null-terminated format-control string, which controls the interpretation of the input fields.

`argument`
Optional argument(s) specify location. Each argument must be a pointer to a variable with a type that corresponds to a type specifier in `format`. The results are unpredictable if there are not enough arguments for the format specification. If there are too many arguments, the extra arguments are evaluated but ignored.

Additional Information

The **fscanf( )** function reads all characters in `stream` up to the first whitespace character (space, tab, or newline), or the first character that cannot be converted according to `format`.

The `format` parameter is as described in the **scanf( )** function.

See also: fprintf( ), scanf( ), sscanf( )

Returns

The number of fields that were successfully converted and assigned, not including fields that were read but not assigned.

EOF for an error or end-of-file on stream before the first conversion.

0 if no fields were assigned.

fseek

Moves the file pointer to a specified location in a stream.

Syntax

```
#include <stdio.h>
      int fseek (FILE *stream, long offset, int origin);
```

Parameters

`stream` Pointer to `FILE` structure.

`offset` Number of bytes from origin.

`origin` Initial position, specified as one of these, or beyond end-of-file. An attempt to position the pointer before the beginning of the file causes an error.

Value	**Meaning**
SEEK_CUR	Current position of file pointer
SEEK_END	End of file
SEEK_SET	Beginning of file

Additional Information

This function clears the end-of-file indicator.

The next operation on the stream takes place at the new location. On a stream open for update, the next operation can be either a read or a write.

When a file is opened for appending data, the last I/O operation determines the current file pointer position, not where the next write would occur. If no I/O operation has yet occurred on a file opened for appending, the file position is the start of the file.

For streams opened in text mode, **fseek( )** has limited use because <CR><LF> translations can cause unexpected results. The only **fseek( )** operations guaranteed to work on streams opened in text mode are seeking with an offset of 0 relative to any `origin` value, or from the beginning of the file with an offset value returned by **ftell( )**.

Results are undefined on devices incapable of seeking, like terminals and printers.

See also: ftell( ), lseek( ), rewind( )

Returns

Value	**Meaning**
0	Successful
Not 0	Unsuccessful

fsetpos

Sets a stream's file pointer position-indicator.

Syntax

```
#include <stdio.h>
      int fsetpos (FILE *stream, const fpos_t *pos);
```

Parameters

`stream`	Pointer to `FILE` structure.
`pos`	File pointer position-indicator storage, which is obtained in a prior call to **fgetpos( )**.

Additional Information

This function clears the end-of-file indicator. After this call, the next operation on the stream may be either input or output.

See also: fgetpos( )

Returns

Value	**Meaning**	
0	Successful	
Not 0	Error occurred; the function sets **errno** to one of these values:	
	EBADF	The specified stream is not a valid file descriptor or is not accessible.
	EINVAL	The stream value is invalid.

fstat

Gets information on the file associated with the specified file descriptor.

Syntax

```
#include <sys/types.h>
      #include <sys/stat.h>
      int fstat (int handle, struct stat *buffer);
```

Parameters

`handle`	Descriptor referring to an open file.
`buffer`	Pointer to file-status structure `stat`.

Additional Information

The file-status structure `stat` is defined in <*sys/stat.h*>.

If `handle` refers to a device, the `size` and `time` elements in the `stat` structure are not meaningful.

Fstat( ) invokes the system call **rq_a_get_file_status** and adds the number of seconds between epoch time and January 1, 1978, plus the local timezone factor, defined in **tzset( )**. This adjusts the time stamps of iRMX files to POSIX-standard values.

This function performs a translation of iRMX OS file ownership rights and iRMX OS access rights to POSIX as described in <*sys/stat.h*>.

See also: chmod( ), filelength( ), stat( ), <*sys/stat.h*>, tzset( )

Returns

Value	**Meaning**
0	Successful
-1	Error occurred and the function sets **errno** to EBADF, indicating an invalid file descriptor

ftell

Gets the current position of the file pointer for a stream.

Syntax

```
#include <stdio.h>
      long ftell (FILE *stream);
```

Parameter

`stream`	Pointer to `FILE` structure.

Additional Information

When a file is opened for appending data, the last I/O operation determines the current file pointer position, not where the next write would occur. For example, if a file is opened for an append and the last operation was a read, the file position is the point where the next read operation would start, not where the next write would start. When a file is opened for appending, the file pointer is moved to end-of-file before any write operation. If no I/O operation has yet occurred on a file opened for appending, the file position is the beginning of the file.

On devices incapable of seeking, such as terminals and printers, or when `stream` does not refer to an open file, the return value is undefined.

See also: fseek( ), lseek( )

Returns

The current file position expressed as an offset relative to the beginning of `stream`. The value returned may not reflect the physical byte offset for streams opened in text mode, since text mode causes <CR><LF> translation. Use **ftell( )** with the **fseek( )** function to return to file locations correctly.

1L on error, and the function sets **errno** to one of these values:

EBADF	Bad file number. The stream argument is not a valid file descriptor value or does not refer to an open file.
EINVAL	Invalid argument. An invalid stream argument was passed to the function.

ftoa

Converts a double value to a formatted string.

Syntax

```
#include <stdlib.h>
      char *ftoa (double value, char *string, unsigned int
      iplaces,
                  unsigned int fplaces);
```

Parameters

`value` Value to convert.

`string` Pointer to a character array where a null-terminated character string is written.

`iplaces`
Desired number of significant integer digits (`iii`) in the string.

`fplaces`
Desired number of significant fractional digits (`fff`) in the string. An integer exponent (`eee`) also returns in the string.

Additional Information

The converted string has this format:

```
[-]iii.fffE[-]eee
```

The value of the number is truncated, not rounded. The algorithm that **ftoa( )** uses is accurate to eighteen significant digits. If `iplaces` plus `fplaces` exceeds eighteen, they are adjusted so that only eighteen significant digits are used.

For portability, use the **sprintf( )** `%e` conversion specifier. Use the optional field width and precision to control the number of fractional digits. The **sprintf( )** `%e` conversion specifier produces a string in the format `[-]d.dddE+ee`, with one integer digit left of the decimal point.

See also: **sprintf( )**

Returns

A pointer to the converted string.

No error return.

fwrite

Writes a specified number of characters to a stream.

Syntax

```
#include <stdio.h>
      size_t fwrite (const void *buffer, size_t size, size_t
      count,
                     FILE *stream);
```

Parameters

`buffer`	Pointer to data to be written.
`size`	Item size in bytes.
`count`	Maximum number of items to be written.
`stream`	Pointer to `FILE` structure.

Additional Information

The file pointer associated with `stream` (if there is one) is incremented by the number of bytes actually written.

If `stream` is opened in text mode, each <CR> is replaced with a <CR><LF> pair. The replacement has no effect on the return value.

See also: fread( ), write( )

Returns

The number of full items actually written, which may be less than `count` if an error occurs.

On error, the file-position indicator cannot be determined.

gcvt

Converts a double value to a string of significant digits, and places them in a specified location.

Syntax

```
#include <stdlib.h>
      char *gcvt (double value, int digits, char *buffer);
```

Parameters

`value`	Value to convert.
`digits`	Number of significant digits stored.
`buffer`	Storage location for result. Should be large enough to accommodate the converted value plus a terminating null character `\0`, which is automatically appended.

Additional Information

There is no provision for overflow.

The **gcvt( )** function attempts to produce significant digits in decimal format. If this is not possible, it produces them in exponential format. Trailing zeros may be suppressed in the conversion.

See also: atof( ), atoi( ), atol( ), ecvt( ), fcvt( )

Returns

A pointer to the string.

No error return.

_get_arguments

Sets up the standard C command line parser.

Syntax

```
#include <rmx_c.h>
      int _get_arguments (int *argc, char **argv, int
      argv_size,
                          char *cmd_buf, int buf_size);
```

Parameters

`argc` Count of command line arguments.

`argv` Array of pointers to arguments.

`argv_size`
: Size of `argv` array.

`cmd_buf`
: Buffer containing parsed arguments pointed to by `argv` elements.

`buf_size`
: Size of `cmd_buf` array.

Additional Information

This function makes successive calls to **rq_c_get_char** to retrieve characters one at a time, parsing the command line into the standard `argc/argv` for main( ).

The **_get_arguments( )** function can be called during run-time; however, the startup code normally invokes this function before your application calls main( ). You can modify the startup code if you have any application-specific initialization requirements that need to be performed before main( ). You can also modify the startup code indirectly with the iRMX configuration process.

See also: Configuring the C library, System Configuration and Administration

Command Line Parsing

Since **_get_arguments** uses **rq_c_get_char**, the HI CLI is bypassed. This allows UNIX-style "-x" flags to be interpreted exactly as expected by a portable C application. Also, the case of each command line argument is preserved; the arguments are not forced to either upper or lower case.

Apostrophe (') and quotation (") characters delimit strings on the command line. Quoted strings permit the use of HI special characters within the string, removing the semantics of any characters within the string. For example, if an ampersand (`&`) is enclosed in quotation characters, the ampersand is no longer recognized as the continuation character. The other special characters are the semicolon (`;`), the pipe symbol (`|`), brackets (`[` and `]`), and the space.

Each of the pair of delimiters surrounding the string must be the same. To include the quoting apostrophe or quotation character inside the string, you must specify the quoting character twice, for example: `"Enter the ""quoted string"" at the prompt"`. You can achieve the same effect by using the apostrophe, for example: `"can't"`.

The parser reduces two successive apostrophes or quotation characters outside of another pair of apostrophes or quotation characters to one apostrophe or quotation character. For example, `""here""` outside all pairs of quotation marks is reduced to `"here"`. This takes place before parsing of the command line.

When a backslash (\) appears on the command line, the backslash is removed and the next character is passed on to the application without interpretation. This is helpful in porting programs that expect and use \ as an escape character.

See also: **rq_c_get_char**, *System Call Reference*, **getopt( )**

Returns

0 always returns.

getc, getchar

Getc( ) reads a single character from a stream and increments the associated file pointer to point to the next character; **getchar( )** reads from *stdin*.

Syntax

```
#include <stdio.h>
      int getc (FILE *stream);
      int getchar (void);
```

Parameter

`stream` Pointer to `FILE` structure.

Additional Information

The **getchar( )** macro is identical to:

```
getc (stdin)
```

Getc( ) and **getchar( )** are identical to **fgetc( )** and **fgetchar( )**, but are macros rather than functions.

See also: fgetc( ), fgetchar( ), putc( ), putchar( )

Returns

The integer value of the character read.

EOF on error or end-of-file. Since EOF is a legal integer value, use **feof( )** or **ferror( )** to distinguish between an error and an end-of-file condition.

getch, getche

Getch( ) reads a single character from the console without echoing; **getche( )** echoes the character read.

Syntax

```
#include <conio.h>
      int getch (void);
      int getche (void);
```

Additional Information

Neither function reads <Ctrl>-<C>.

When reading a function key or cursor-moving key, these functions must be called twice; the first call returns 0 or 0xe0, and the second call returns the actual key code.

See also: cgets( ), getchar( ), ungetch( )

Returns

The character read.

No error return.

getenv

Searches the environment-variable table for a specified entry.

Syntax

```
#include <stdlib.h>
      char *getenv (const char *varname);
```

Parameters

`varname`

Name of environment variable being sought. The `varname` argument should match the case of the environment variable.

Additional Information

The **getenv( )** function is case-sensitive.

The first call to **getenv( )** sets up an environment-variable table shared by all tasks using the C library. A prototype for the table is contained in the file *:config:r?env*. When **getenv( )** is called for the first time, the table is initialized from *:config:r?env*. You can create an environment-variable file locally, *:prog:r?env*, that **getenv( )** uses in addition to *:config:r?env*, as a basis for the table. The maximum allowable number of entries in the environment-variable table is 40. Entries in the *r?env* files are of this form:

```
varname = [ASCII string]
```

A space character is required on both sides of the equal sign for **fscanf( )** parsing. For example, a typical entry in *:config:r?env* appears like this.

```
TZ = PST8PDT
```

See also: **putenv( )**, **tzset( )**,
Environment variables, *System Configuration and Administration*

Returns

A pointer to the environment-variable table entry containing the current string value of `varname`. To update the entry, pass this pointer to the **putenv( )** call.

A null pointer if the given variable is not currently defined.

_get_cs

Returns an application’s current code segment.

Syntax

```
#include <rmx_c.h>
      selector _get_cs (void);
```

Additional Information

Use this function for obtaining an application’s code segment. This function can be used for all memory models, i.e., compact and large, and it is the only function which can used for accessing a flat model application’s code segment.

See also: **_get_ds( )**, **_get_ss( )** commands

Returns

The current value of the code segment register.

_get_ds

Returns an application’s current data segment.

Syntax

```
#include <rmx_c.h>
      selector _get_ds (void);
```

Additional Information

Use this function for obtaining an application’s data segment. This function can be used for all memory models, i.e., compact and large, and it is the only function which can used for accessing a flat model application’s data segment.

See also: **_get_cs( )**, **_get_ss( )** commands

Returns

The current value of the data segment register.

_get_ss

Returns an application's current stack segment.

Syntax

```
#include <rmx_c.h>
      selector _get_ss (void);
```

Additional Information

Use this function for obtaining an application's stack segment. This function can be used for all memory models, i.e., compact and large, and it is the only function which can used for accessing a flat model application's stack segment.

See also: **_get_cs( )**, **_get_ds( )** commands

Returns

The current value of the stack segment register.

_get_info

Obtains the C library information `CINFO_STRUCT` for the calling task.

Syntax

```
#include <rmx_c.h>
       int _get_info (unsigned int count, CINFO_STRUCT *cinfo);
```

Parameters

`count`	Number of elements to be returned in `CINFO_STRUCT`.
`cinfo`	Pointer to `CINFO_STRUCT`.

Additional Information

The `CINFO_STRUCT`, part of the resources allocated to each task that uses the C library, contains these elements:

Element	**Description**
int num_eios_bufs	Number of EIOS buffers per open file connection allocated on behalf of the calling task. This is used in the call to **rq_s_open** made by the **fopen( )** or **open( )** functions.
unsigned long * accounting	Pointer to an array containing a counter for each configured function in the C library. The C library uses this array to keep track of the number of times a function has been called since the library was loaded, and to indicate whether or not a function is configured.
unsigned short num_accounting	Size of the accounting array.
int num_clib_functs	Number of functions implemented in this version of the C Library.
unsigned char *flags	One entry per function indicating whether the function is configured.

Note
For flat model applications only, treat the accounting and flags parameters as two separate fields each in the structure. The first field has the parameter name listed above and is a near pointer. The second field has the same name with _seg appended at the end. It is a segment selector for the pointer. For example, accounting is a pointer and accounting_seg is the selector to it.

See also: **_cstop( )**, <*rmx_c.h*>, **_set_info( )**

Returns

Value	**Meaning**
0	Successful
-1	Unsuccessful

getopt

Gets the next argument option letter that matches recognized option letters.

Syntax

```
#include <udistd.h>
      char getopt (int argc, char **argv, char optstring);
      char *optarg /* Global variables affected by getopt( ) */
      int optind
```

Parameters

`argc, argv`
: Standard command line arguments passed to main( ).

`optstring`
: A string of recognized option letters.

Additional Information

This function compares command line arguments found in `argv` with `optstring`. The found argument is indicated in the global variables `optarg` and `optind`, where `optarg` points to the argument, and `optind` is set to the `argv` index of the next argument on the command line. On return from `getopt`, `optarg` is set to point to the start of the option argument, if any.

If a letter in `optstring` is followed by a colon, the option is expected to have an argument that may be separated by white space in the command line.

See also: _get_arguments

Returns

The next letter in `argv` that matches a letter in `optstring`.

EOF when all options have been processed.

getpid, getuid

Getpid gets the calling task's connection token (process ID); **getuid** gets the calling task's user ID.

Syntax

```
#include <process.h>
      pid_t getpid (void);
      uid_t getuid (void);
```

Additional Information

Getuid( ) invokes the system calls **rq_get_default_user** and **rq_inspect_user**.

See also:	rq_get_default_user, rq_inspect_user, *System Call Reference*, mktemp( )

Returns

No error return.

_get_rmx_conn

Translates a file descriptor to a valid iRMX connection token, usable as a parameter in iRMX system calls.

Syntax

```
#include <rmx_c.h>
      selector _get_rmx_conn (int handle);
```

Parameter

`handle` Descriptor referring to an open file.

Additional Information

Use this function in code that mixes C library functions with direct iRMX system calls.

File descriptors are maintained on a per-task basis. When a file is opened, a small, non-negative file descriptor is returned as specified by POSIX. The file descriptor is not an iRMX connection; it is an index into an internal table of iRMX connections.

➠ **Note**
C string tokens are `char` values separated by delimiter characters; an iRMX connection token is a `selector` value. Do not confuse the C concept of a character string token with the iRMX connection token.

See also: **_put_rmx_conn**, *<rmx_c.h>*

Returns

A valid iRMX connection token.

-1 if unsuccessful.

gets

Gets a line from *stdin* and stores it in the specified location.

Syntax

```
#include <stdio.h>
      char *gets (char *buffer);
```

Parameter

`buffer` Storage location for input string.

Additional Information

The line consists of all characters up to and including the first newline character `\n`. The **gets( )** function replaces the newline character with a null character `\0` before returning the line.

The **fgets( )** function retains the newline character.

See also: fgets( ), fputs( ), puts( )

Returns

Returns its argument if successful.

A null pointer on error or end-of-file. Use **ferror( )** or **feof( )** to determine which one has occurred.

getw

Reads the next integer from a stream and increments the associated file pointer (if there is one) to point to the next unread value.

Syntax

```
#include <stdio.h>
      int getw (FILE *stream);
```

Parameter

`stream` Pointer to `FILE` structure.

Additional Information

The **getw( )** function does not assume any special alignment of items in the stream.

The **getw( )** function is provided primarily for compatibility with previous libraries. Portability problems may occur with **getw( )**, since the integer size and byte ordering can differ across systems.

See also: **putw( )**

Returns

The integer value read.

EOF on error or end-of-file. Since the EOF value is also a legitimate integer value, use **feof( )** or **ferror( )** to verify an end-of-file or error condition.

gmtime

Converts a time value to a structure.

Syntax

```
#include <time.h>
      struct tm *gmtime (const time_t *timer);
```

Parameter

`timer`	Pointer to stored `tm` structure, which represents the seconds elapsed since epoch time. This value is usually obtained from a call to the **time( )** function.

Additional Information

The **gmtime( )** function breaks down the `timer` value and stores it in a `tm` structure. The structure result reflects GMT, not local time.

The **gmtime( )**, **mktime( )**, and **localtime( )** functions use a single statically allocated structure to hold the result. Subsequent calls to these functions destroy the result of any previous call.

See also: **asctime( )**, **localtime( )**, **time( )**, *<time.h>* for description of `tm` structure

Returns

A pointer to the `tm` structure.

No error return.

is Functions

Test integers representing ASCII characters for specified conditions.

Syntax

```
#include <ctype.h>
      int isalnum (int c);
      int isalpha (int c);
      int isascii (int c);
      int iscntrl (int c);
      int isdigit (int c);
      int isgraph (int c);
      int islower (int c);
      int isprint (int c);
      int ispunct (int c);
      int isspace (int c);
      int isupper (int c);
      int isxdigit (int c);
```

Parameter

c	Integer to be tested.

Additional Information

These functions are implemented as functions and macros. The test conditions are:

Function	**Test Conditions**
isalnum( )	Alphanumeric (A-Z, a-z, or 0-9)
isalpha( )	Letter (A-Z or a-z)
isascii( )	ASCII character (0x00-0x7F)
iscntrl( )	Control character (0x00-0x1F or 0x7F)
isdigit( )	Digit (0-9)
isgraph( )	Printable character except space
islower( )	Lowercase letter (a-z)
isprint( )	Printable character (0x20-0x7E)
ispunct( )	Punctuation character
isspace( )	White-space character (0x09-0x0D or 0x20)
isupper( )	Uppercase letter (A-Z)
isxdigit( )	Hexadecimal digit (A-F, a-f, or 0-9)

All of these functions except **isascii( )** produce a defined result only for integer values corresponding to the ASCII character set, or for the nonASCII value EOF.

See also: toascii( ), tolower( ), toupper( )

Returns

Value	**Meaning**
Not 0	The integer satisfies the test condition.
0	It does not.

isatty

Determines whether a file descriptor is associated with a character device: a terminal, console, printer, or serial port.

Syntax

```
#include <io.h>
      int isatty (int handle);
```

Parameter

`handle`	Descriptor referring to device to be tested.

Returns

Value	**Meaning**
Not 0	The device is a character device.
0	It is not. If `handle` is an invalid file descriptor, the function also sets **errno** to EBADF.

itoa

Converts an integer of the specified base to a null-terminated string of characters and stores it.

Syntax

```
#include <stdlib.h>
      char *itoa (int value, char *string, int radix);
```

Parameters

`value`	Number to convert.
`string`	String result, up to 17 bytes.
`radix`	Specifies the base of `value`; must be in the range 2-36.

Additional Information

If `radix` equals 10 and `value` is negative, the first character of the stored `string` is the minus sign (`-`).

If `radix` is greater than 10, digits in the converted string representing values 10 through 35 are the characters `a` through `z`.

See also: ltoa( ), ultoa( )

Returns

A pointer to the converted string.

No error return.

itoh

Converts an integer into the equivalent null-terminated, hexadecimal string and stores it.

Syntax

```
#include <stdlib.h>
      char *itoh (int n, char *buffer);
```

Parameters

`n`	Integer to convert.
`buffer`	Pointer to a string. The buffer must be large enough to hold the largest integer on the target system.

Additional Information

The **itoh( )** function converts all non-numeric hexadecimal characters to lower case. This function also does not place a leading 0 character in the buffer.

For portability, use the **sprintf( ) %x** conversion specifier.

See also: **sprintf( )**

Returns

A pointer to the converted string.

No error return.

labs

Calculates the absolute value of a long integer.

Syntax

```
#include <stdlib.h>
      #include <math.h>
      long labs (long n);
```

Parameter

n	Long integer to calculate absolute value for.

See also: abs( ), fabs( ), cabs( )

Returns

The absolute value result.

No error return.

ldexp

Computes a real number from the mantissa and exponent.

Syntax

```
#include <math.h>
      double ldexp (double x, int exp);
```

Parameters

`x`	Mantissa value.
`exp`	Integer exponent.

See also: frexp( ), modf( )

Returns

Returns `x` * 2^{exp}.

±HUGE_VAL (depending on the sign of `x`) on overflow, and the function sets **errno** to ERANGE.

This function does not return standard ANSI domain or range errors.

ldiv

Divides numerator by denominator, and computes the quotient and remainder.

Syntax

```
#include <stdlib.h>
      ldiv_t ldiv (long int numer, long int denom);
```

Parameters

`numer`	Numerator.
`denom`	Denominator. If the denominator is 0, the program will terminate with an error message.

Additional Information

The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer that is less than the absolute value of the mathematical quotient.

The **ldiv( )** function is similar to the **div( )** function, except that the arguments and the members of the returned structure are long integers.

See also: **div( )**

Returns

A `ldiv_t` structure, comprising both the quotient and the remainder, defined in *<stdlib.h>*.

lfind

Performs a linear search for a specified key in an unsorted array.

Syntax

```
#include <search.h>
      char *lfind (const void *key, const void *base,
                   unsigned int *num, unsigned int width,
                   int (*compare) (const void *elem1,
                   const void *elem2));
```

Parameters

`key`	Value being sought.
`base`	Pointer to base of the array to be searched.
`num`	Number of elements in the array.
`width`	Width of elements in bytes.
`compare`	Pointer to a user-supplied routine that compares two array elements, `elem1` and `elem2`, and returns a value specifying their relationship.
`elem1`	Pointer to the key for the search.
`elem2`	Pointer to the array element to be compared with the key.

Additional Information

The **lfind( )** function calls the `compare` routine one or more times during the search, passing pointers to two array elements on each call. This routine must compare the elements, then return a non-0 value if the elements are different, or 0 if the elements are identical.

See also: bsearch( ), lsearch( ), qsort( )

Returns

A pointer to the array element that matches `key`.

A null pointer if a match is not found.

localeconv

Gets detailed information on locale settings.

Syntax

```
#include <locale.h>
      struct lconv *localeconv (void);
```

Additional Information

This information is stored in a `lconv` structure, defined in <*locale.h*>. Subsequent calls to **setlocale( )** with category values of LC_ALL, LC_MONETARY, or LC_NUMERIC will overwrite the contents of this structure.

See also: <*locale.h*>, setlocale( ), strcoll( ), strftime( ), strxfrm( )

Returns

A pointer to an `lconv` structure.

localtime

Converts a time stored as a `time_t` value and corrects for the local timezone.

Syntax

```
#include <time.h>
      struct tm *localtime (const time_t *timer);
```

Parameter

`timer`	Pointer to stored time, which represents the seconds elapsed since epoch time; this value is usually obtained from the **time( )** function.

Additional Information

The **localtime( )** function makes corrections for the local timezone if the user first sets the environment variable `TZ`. Then, three other environment variables (`timezone`, `daylight`, and `tzname`) are automatically set as well.

See also: Description of these variables in **tzset( )**

`TZ` is not part of the ANSI standard definition of **localtime( )**.

The **gmtime( )**, **mktime( )**, and **localtime( )** functions use a single statically allocated `tm` structure for the conversion. Each call to one of these functions destroys the result of the previous call.

See also: asctime( ), ctime( ), gmtime( ), time( )

Returns

A pointer to the `tm` structure, which has the integer elements described in *<time.h>*.

log, log10

Log( ) calculates the natural logarithm of a value and **log10( )** calculates the base-10 logarithm.

Syntax

```
#include <math.h>
      double log (double x);
      double log10 (double x);
```

Parameter

`x`	Value to find logarithm for.

See also: exp( ), matherr( ), pow( )

Returns

The logarithm of the argument `x`.

-HUGE_VAL if *x* is negative; the function prints a DOMAIN error message to *stderr* and sets **errno** to EDOM.

-HUGE_VAL if `x` is 0; the function prints a SING error message to *stderr* and sets **errno** to ERANGE.

These functions do not return standard ANSI domain or range errors.

longjmp

Restores the context, previously saved by **setjmp( )**.

Syntax

```
#include <setjmp.h>
      void longjmp (jmp_buf context, int value);
```

Parameters

`context`
: Context previously stored by **setjmp( )**.

`value` Value to be returned to **setjmp( )**; must be non-0. If 0, the value 1 is returned to the previous **setjmp( )** call.

Additional Information

The previous call to **setjmp( )** causes the current context to be saved in `context`. A subsequent call to **longjmp( )** restores the context and returns control to the point immediately following the corresponding **setjmp( )** call. Execution resumes as if `value` had just been returned by **setjmp( )**.

The values of all local variables (except register variables) that are accessible to the routine receiving control contain the values they had when **longjmp( )** was called. The values of register variables are unpredictable.

Observe these restrictions when using **longjmp( )**:

- Do not assume that the values of the register variables will remain the same. The values of register variables in the routine calling **setjmp( )** may not be restored to the proper values after **longjmp( )** is executed.
- Do not use **longjmp( )** to transfer control out of an interrupt-handling routine.

See also: **setjmp( )**

Returns

Nothing.

lsearch

Performs a linear search for a specified value in an unsorted array, appending the value to the array if not found.

Syntax

```
#include <search.h>
      char *lsearch (const void *key, const void *base,
      unsigned int
                     *num, unsigned int width, int (*compare)
                     (const void *elem1, const void *elem2));
```

Parameters

`key`	Value being sought.
`base`	Pointer to base of the array to be searched.
`num`	Number of elements in the array.
`width`	Width of elements in bytes.
`compare`	Pointer to a user-supplied routine that compares two array elements, `elem1` and `elem2`, and returns a value specifying their relationship.
`elem1`	Pointer to the key for the search.
`elem2`	Pointer to the array element to be compared with the key.

Additional Information

The **lsearch( )** function calls the `compare` routine one or more times during the search, passing pointers to two array elements on each call. This routine must compare the elements, then return a non-0 value if the elements are different, or 0 if the elements are identical.

See also: bsearch( ), lfind( )

Returns

A pointer to the array element that matches `key`.

A pointer to the newly added element in the array if a match is not found.

lseek

Moves the file pointer to a location specified as an offset from the origin in a file.

Syntax

```
#include <io.h>
      #include <unistd.h>
      off_t lseek (int handle, off_t offset, int origin);
```

```
long64 _lseek64(int handle, long64 offset, int origin);
```

Parameters

`handle` Descriptor referring to an open file.

`offset` Number of bytes from `origin`, specified as one of these constants, or beyond end-of-file.

Value	**Meaning**
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End of file

`origin` Initial position.

Additional Information

The next operation on the file occurs at the new location.

The **lseek( )** function can reposition the pointer anywhere in a file and beyond the end of the file. An attempt to position the pointer before the beginning of the file causes an error.

Results are undefined on devices incapable of seeking, like terminals and printers.

The **_lseek64( )** function allows the use of 64-bit offsets used with the extended iRMX filesystems.

See also: **fseek( )**

Returns

The offset, in bytes, of the new position from the beginning of the file.

-1L on error, and the function sets **errno** to one of these values:

EBADF	Invalid file descriptor.
EINVAL	Invalid value for `origin`, or position specified by `offset` is before the beginning of the file.

ltell

Returns the absolute position of the file pointer for the next I/O operation.

Syntax

```
#include <io.h>
      long ltell (int handle);

long64 _ltell64 (int handle);
```

Parameter

`handle` Descriptor referring to an open file.

Additional Information

This function is equivalent to:

```
lseek (handle, 0L, SEEK_CUR)
```

The **_tell64( )** function returns the offset as a 64-bit offset, for use with the extended iRMX filesystems.

See also: **lseek( )**

Returns

The absolute position of the next byte in the file.

-1 with **errno** set to EBADF if unsuccessful.

ltoa

Converts a long integer of the specified base to a null-terminated string of characters and stores it.

Syntax

```
#include <stdlib.h>
      char *ltoa (long value, char *string, int radix);
```

Parameters

`value`	Number to convert.
`string`	String result, up to 34 bytes.
`radix`	Base of `value`; must be in the range 2-36.

Additional Information

If `radix` equals 10 and `value` is negative, the first character of the stored string is the minus sign (-).

If `radix` is greater than 10, digits in the converted string representing values 10 through 35 are the characters `a` through `z`.

See also: itoa( ), ltos( ), utoa( )

Returns

A pointer to the converted string.

No error return.

ltoh

Converts a long integer to a null-terminated hexadecimal string and stores it.

Syntax

```
#include <stdlib.h>
      char *ltoh (unsigned long value, char *string);
```

Parameters

`value`	Integer to convert.
`string`	String result, up to 34 bytes.

Additional Information

This function does not place leading 0 characters in the result.

This function produces hexadecimal characters in lower case (`a-f`). For portability, use the **sprintf( ) %lx** conversion specifier.

See also: **sprintf( )**

Returns

A pointer to the converted string.

No error return.

ltos

Converts a long integer to a null-terminated string of characters and stores it; negative base values are acceptable.

Syntax

```
#include <stdlib.h>
      char *ltos (long value, char *string, int radix);
```

Parameters

`value`	Number to convert.
`string`	String result, up to 34 bytes.
`radix`	Base of `value`; must be in the range 2 to 36 or -2 to -36.

Additional Information

The absolute value of `radix` is passed to this function as the number base.

Digits in the converted string representing values 10 through 35 are the characters `a` through `z`.

See also: ltoa( ), ltoh( )

Returns

A pointer to the converted string.

No error return.

malloc

Allocates a memory block of the specified size.

Syntax

```
#include <stdlib.h>
      void *malloc (size_t size);
```

Parameter

`size`	Bytes to allocate.

Additional Information

The allocated block may be larger than the specified size, including space required for alignment and maintenance information. The memory is suitably aligned for storage of any type of object.

Always examine the return from **malloc( )**, even if the amount of memory requested is small.

See also: calloc( ), free( ), realloc( )

Returns

A pointer to the allocated space. To get a pointer to a type other than `void`, use a type cast on the return value.

For a size of 0 bytes, **malloc()** returns a NULL.

If unsuccessful, it returns a NULL pointer.

Note
For a size of 0 bytes, the NULL returned by **malloc()** is a non-standard implementation.

matherr

Processes errors generated by the functions of the math library.

Syntax

```
#include <math.h>
      int matherr (struct exception *except);
```

Parameter

`except`	Pointer to an exception structure.

Additional Information

When an error occurs in a math function, **matherr( )** is called with a pointer to the `exception` structure defined in <*math.h*>.

See also: acos( ), asin( ), atan( ), Bessel functions, cos( ), exp( ), log( ), pow( ), sin( ), sqrt( ), tan( )

Returns

Value	**Meaning**
Not 0	Successful
0	Error occurred

mblen

Gets the length and determines the validity of a multibyte character.

Syntax

```
#include <stdlib.h>
      int mblen (const char *mbstr, size_t count);
```

Parameters

`mbstr`	A pointer to a sequence of bytes (a multibyte character) to check.
`count`	The number of bytes to check.

See also: mbstowcs( ), mbtowc( ), wcstombs( ), wctomb( )

Returns

The length, in bytes, of the multibyte character.

0 if `mbstr` is a null pointer or the object that it points to is the wide-character null.

-1 if the object that `mbstr` points to does not form a valid multibyte character within the first `count` characters, up to MB_CUR_MAX.

mbstowcs

Converts a sequence of multibyte characters to a sequence of wide characters, as determined by the current locale; stores the resulting wide-character string at the specified address.

Syntax

```
#include <stdlib.h>
      size_t mbstowcs (wchar_t *wcstr, const char *1mbstr,
                       size_t count);
```

Parameters

`wcstr`	The address of a sequence of wide characters.
`mbstr`	The address of a sequence of multibyte characters.
`count`	The number of multibyte characters to convert.

Additional Information

If **mbstowcs( )** encounters the null character `\0` either before or when `count` occurs, it converts the null character to a wide-character null and stops. Thus, the wide-character string at `wcstr` is null-terminated only if it encounters a null character during conversion.

If the sequences pointed to by `wcstr` and `mbstr` overlap, the behavior is undefined.

The result is similar to a series of calls to **mbtowc( )**.

See also: mblen( ), mbtowc( ), wcstombs( ), wctomb( )

Returns

The number of converted multibyte characters or `count` if the wide-character string is not null-terminated.

-1 on encountering an invalid multibyte character.

mbtowc

Converts a multibyte character to a corresponding wide character.

Syntax

```
#include <stdlib.h>
      int mbtowc (wchar_t *wchar, const char *mbchar, size_t
      count);
```

Parameters

`wchar`	A pointer to the wide character produced.
`mbchar`	A pointer to a sequence of bytes (a multibyte character).
`count`	The number of bytes to check.

Additional Information

Mbtowc( ) will not examine more than MB_CUR_MAX bytes.

See also: mblen( ), mbstowcs( ), wcstombs( ), wctomb( )

Returns

The length in bytes of the multibyte character.

0 if `mbchar` is a null pointer or the object that it points to is a wide-character null.

-1 if the object that `mbchar` points to does not form a valid multibyte character within the first `count` characters.

memccpy

Copies characters from one buffer to another, halting when the specified character is copied or when the specified number of bytes have been copied.

Syntax

```
#include <string.h>
      void * memccpy (void *dest, void *src, int c,
                      unsigned int count);
```

Parameters

`dest`	Pointer to destination buffer.
`src`	Pointer to source buffer.
`c`	Last character to copy.
`count`	Number of characters.

See also: memchr( ), memcmp( ), memcpy( ), memset( )

Returns

A pointer to the byte in `dest` that immediately follows the character `c`.

A null pointer if unsuccessful.

memchr

Finds the first occurrence of a character in a buffer and stops when it finds the character or when it has checked the specified number of bytes.

Syntax

```
#include <string.h>
      void *memchr (const void *buf, int c, size_t count);
```

Parameters

`buf`	Pointer to buffer.
`c`	Character to look for.
`count`	Number of characters to check for.

See also: memccpy( ), memcmp( ), memcpy( ), memset( ), strchr( )

Returns

A pointer to the first location of `c` in `buf`.

A null pointer if unsuccessful.

memcmp

Compares the specified number of bytes of two buffers and returns a value indicating their relationship.

Syntax

```
#include <string.h>
      int memcmp (const void *buf1, const void *buf2, size_t
      count);
```

Parameters

`buf1`	First buffer.
`buf2`	Second buffer.
`count`	Number of characters.

See also: memccpy( ), memchr( ), memcpy( ), memset( ), strcmp( ), strncmp( )

Returns

Value	**Meaning**
< 0	buf1 less than buf2
= 0	buf1 identical to buf2
> 0	buf1 greater than buf2

memcpy

Copies specified number of bytes from a source buffer to a destination buffer.

Syntax

```
#include <string.h>
      void *memcpy (void *dest, const void *src, size_t count);
```

Parameters

`dest`	Buffer to copy to.
`src`	Buffer to copy from.
`count`	Number of characters to copy.

Additional Information

If the source and destination overlap, **memcpy( )** does not ensure that the original source bytes in the overlapping region are copied before being overwritten. Use **memmove( )** to handle overlapping regions.

See also: memccpy( ), memchr( ), memcmp( ), memmove( ), memset( ), strcpy( ), strncpy( )

Returns

A pointer to `dest`.

memicmp

Compares characters in two buffers byte-by-byte (case-insensitive).

Syntax

```
#include <string.h>
      int memicmp (void *buf1, void *buf2, unsigned int count);
```

Parameters

`buf1`	First buffer.
`buf2`	Second buffer.
`count`	Number of characters to compare.

See also: memccpy( ), memchr( ), memcmp( ), memcpy( ), memset( ), stricmp( ), strnicmp( )

Returns

The relationship of the two buffers.

Value	**Meaning**
< 0	buf1 less than buf2
= 0	buf1 identical to buf2
> 0	buf1 greater than buf2

memmove

Moves a specified number of bytes from a source buffer to a destination buffer.

Syntax

```
#include <string.h>
      void *memmove (void *dest, const void *src, size_t
      count);
```

Parameters

`dest`	Pointer to destination buffer.
`src`	Pointer to source buffer.
`count`	Number of characters to copy.

Additional Information

If some regions of the source area and the destination overlap, this function ensures that characters in the overlapping region are copied before being overwritten.

See also: memccpy( ), memcpy( ), strncpy( )

Returns

A pointer to `dest`.

memset

Sets characters in a buffer to a specified character.

Syntax

```
#include <string.h>
      void *memset (void *dest, int c, size_t count);
```

Parameters

`dest`	Pointer to destination.
`c`	Character to set to.
`count`	Number of characters to set.

See also: memccpy( ), memchr( ), memcmp( ), memcpy( ), strnset( )

Returns

A pointer to `dest`.

mkdir

Creates a new directory with the specified ownership and access rights.

Syntax

```
#include <direct.h>
      int mkdir (const char *pathname, mode_t pmode);
```

Parameters

`pathname`

Pathname of the directory to create. Name the new directory according to the rules for the iRMX OS.

See also: *Command Reference* for rules for naming directories

`pmode` Permission mode: the ownership and access rights as one or more of the manifest constants described in **chmod( )**. Join more than one constant with the bitwise-OR operator (`|`).

Additional Information

The **mkdir( )** function applies the default file-permission mask (set with the **umask( )** function) to `pmode` before setting the permissions.

By default, this function creates directories that all tasks can share. If O_EXCL is ORed with `pmode`, the file is opened with share-with-none permission, like UNIX.

This function performs a translation of POSIX file ownership rights and POSIX access rights to the iRMX OS equivalent as described in **chmod( ).**

See also: <*errno.h*>, **chmod( )**, **umask( )**

Returns

Value	**Meaning**
0	Successful
-1	Unsuccessful; the function sets **errno** to EACCES, EEXIST, ENOENT, ENOSPC, or ENOTDIR

mktemp

Creates a unique temporary filename.

Syntax

```
#include <io.h>
      char *mktemp (char *template);
```

Parameter

`template`
: Filename template.

Additional Information

Creates a unique filename by modifying a template argument in the form:

```
baseXXXXXX
```

Where:

`base`	Is the part of the new filename that you supply, and the `X`s are placeholders for the part supplied by **mktemp( )**.

This function preserves `base` and replaces the six trailing `X`'s with an alphanumeric character followed by a five-digit value. The alphanumeric character is `0` the first time **mktemp( )** is called with a given template. The five-digit value is a unique number based upon the calling task ID.

In subsequent calls from the same task with copies of the same template, **mktemp( )** checks to see if previously returned names have already been used to create files. If no file exists for a given name, **mktemp( )** returns that name. If files exist for all previously returned names, **mktemp( )** creates a new name by replacing the alphanumeric character in the name with the next available lowercase letter. For example, if the first name returned is `t012345` and this name is used to create a file, the next name returned will be `ta12345`. When creating new names **mktemp( )** uses, in order, `0` and then the lowercase letters `a` through `z`.

The first call to **mktemp( )** modifies the original template. If you call **mktemp( )** again with the same template (that is, the original one), an error returns.

The **mktemp( )** function does not create or open files, only filenames.

See also: fopen( ), getpid( ), open( ), tmpnam( ), tmpfile( )

Returns

A pointer to the modified template.

A null pointer if the `template` argument is badly formed or no more unique names can be created from the given template.

mktime

Converts the time/date structure into a fully defined structure with normalized values and then converts it to calendar time.

Syntax

```
#include <time.h>
      time_t mktime (struct tm *timedate);
```

Parameter

`timedate`

Time/date structure, `tm`, possibly incomplete.

Additional Information

The converted time has the same encoding as the values returned by the **time( )** function.

The elements of the `tm` structure contain the values described in <*time.h*>.

The original values of the `tm_wday` and `tm_yday` components in `tm`, and the original values of the other components are not restricted to their normal ranges. If successful, **mktime( )** sets the values of `tm_wday` and `tm_yday` appropriately, and sets the other components to represent the specified calendar time, but with their values forced to the normal ranges; the final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined.

The **gmtime( )** and **localtime( )** functions use a single statically allocated buffer for the conversion. If you supply this buffer to **mktime( )**, it destroys the previous contents .

See also: asctime( ), ctime( ), gmtime( ), localtime( ), time( ), <*time.h*>

Returns

The specified calendar time encoded as a `time_t`.

-1 cast as type `time_t` if the calendar time cannot be represented.

-1 if `timedate` references a date before epoch time.

modf

Splits a value into fractional and integer parts, retaining the sign.

Syntax

```
#include <math.h>
      double modf (double x, double *intptr);
```

Parameters

`x`	Value to split.
`intptr`	Pointer to integer portion stored as a double value.

See also: frexp( ), ldexp( )

Returns

The signed fractional portion of `x`.

No error return.

onexit

Registers a function to be called when the task terminates normally.

Syntax

```
#include <stdlib.h>
      onexit_t onexit (onexit_t func);
```

Parameter

`func`	Pointer to function(s) to be called on normal termination using **exit( )**. The functions passed to **onexit( )** cannot take parameters.

Additional Information

Successive calls to **onexit( )** create a register of functions that execute in LIFO (last-in, first-out) order. You can register a maximum of 32 functions.

The ANSI-standard **atexit( )** function does the same thing as **onexit( )**; use it instead of **onexit( )** when ANSI portability is desired.

See also: atexit( ), exit( )

Returns

A pointer to the function(s) to call.

A null pointer if the number of functions exceeds 32.

open

Opens a file and prepares it for subsequent reading or writing.

Syntax

```
#include <fcntl.h>
      #include <io.h>
      #include <sys/stat.h>
      int open (const char *filename, int oflag [, int pmode]);
```

Parameters

`filename`
: Filename of file to open.

`oflag` Open mode (type of operations allowed) as an integer expression formed from one or more of the manifest constants defined in *<fcntl.h>*. `Oflag` must contain either O_RDONLY, O_RDWR, or O_WRONLY. Combine two or more of the constants with the bitwise-OR operator (`|`). There is no default.

`pmode` Permission mode, required when specifying O_CREAT. Ignored if the file exists. Specifies the file's ownership and access rights, which are set when the new file is closed for the first time. Contains one or more of the manifest constants described in **chmod( )**.

Additional Information

The **open( )** function applies the default file-permission mask set with the **umask( )** function to `pmode` before setting the permissions.

By default, this function creates files that all tasks can share. If O_EXCL is ORed with `pmode`, the file is opened with share-with-none permission, like UNIX.

This function makes the system call **rq_s_open** and performs a translation of POSIX file ownership rights and POSIX access rights to the iRMX OS equivalent as described in **chmod( )**.

See also: **chmod( )**, **close( )**, **creat( )**, **dup( )**, **dup2( )**, *<fcntl.h>*, **fopen( )**, *<sys/stat.h>*, **sopen( )**, **umask( )**, in this manual
rq_s_open, *System Call Reference*

Returns

A file descriptor for the opened file.

-1 on error, and the function sets **errno** to one of these values:

EACCES	Given pathname is a directory; or an attempt was made to open a read-only file for writing; or a sharing violation occurred (the file's share mode does not allow the specified operations).
EEXIST	The O_CREAT and O_EXCL flags are specified, but the named file already exists.
EINVAL	An invalid `oflag` or `pmode` argument was given.
EMFILE	No more file descriptors available (too many open files).
ENOENT	File or pathname not found.

opendir

Opens a directory stream that corresponds to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

Syntax

```
#include <sys/types.h>
      #include <dirent.h>
```

```
DIR *opendir(const char *name);
```

Parameters

`name`

Name of directory to open.

Returns

The opendir() function returns a pointer to the directory stream or NULL if an error occurred.

EACCES	Permission denied.
EMFILE	Too many file descriptors in use by process.
ENFILE	Too many files are currently open in the system.
ENOENT	Directory does not exist, or name is an empty string.
ENOMEM	Insufficient memory to complete the operation.
ENOTDIR	Name is not a directory.
See also:	open(2), readdir(3), closedir(3), rewinddir(3)

perror

Prints an error message to *stderr*.

Syntax

```
#include <stdio.h>
      void perror (const char *string);
```

Parameter

`string` Message to print.

Additional Information

The `string` argument prints first, followed by a colon, the system error message for the last library call that produced the error, and a newline character.

If `string` is a null pointer or a pointer to a null string, **perror( )** prints only the system error message.

The actual error number is stored in the variable **errno**. The system error messages are accessed through `sys_errlist`, an array of messages ordered by error number. The **perror( )** function prints the appropriate error message by using the **errno** value as an index to `sys_errlist`. The value of the variable `sys_nerr` is defined as the maximum number of elements in the `sys_errlist` array.

To produce accurate results, call **perror( )** immediately after an error occurs. Otherwise, the **errno** value may be overwritten by subsequent calls.

See also: clearerr( ), <*errno.h*>, ferror( ), strerror( )

Returns

Nothing.

pow

Computes a value raised to the power of another value.

Syntax

```
#include <math.h>
      double pow (double x, double y);
```

Parameters

x	Number to be raised.
y	Power to raise x to.

Additional Information

The **pow( )** function does not recognize integral double values greater than 2^{64}, such as 1.0E100.

See also: exp( ), log( ), sqrt( )

Returns

The value of x^{y}.

1 if x is not 0.0 and y is 0.0.

0, and the function sets **errno** to EDOM if x is 0.0 and y is negative.

0 ,and the function sets **errno** to EDOM and prints a DOMAIN error message to *stderr* if both x and y are 0.0, or if x is negative and y is not an integer.

±HUGE_VAL, and sets **errno** to ERANGE if an overflow results. No message is printed on overflow or underflow.

This function does not return standard ANSI domain or range errors.

printf

Prints formatted data to *stdout*.

Syntax

```
#include <stdio.h>
      int printf (const char *format [, argument]...);
```

Parameters

`format` Formatted string consisting of ordinary characters, escape sequences, and (if arguments follow) format specifications that determine the output format for the arguments.

`argument`
Optional arguments.

Additional Information

The ordinary characters and escape sequences are copied to *stdout* in order of their appearance. For example, the line:

```
printf("Line one\n\t\tLine two\n");
```

produces the output:

```
Line one
        Line two
```

Format specifications always begin with a percent sign (`%`) and are read left to right. When **printf( )** encounters the first format specification, it converts and outputs the value of the first argument after `format`. The second format specification causes **printf( )** to convert and output the second argument, and so on. If there are more arguments than format specifications, **printf( )** ignores the extra arguments. The results are undefined if there are fewer arguments than format specifications.

Format Specification

A format specification, consisting of optional and required elements, has the form:

```
%[flags] [width] [.precision] [ F | N | h | l | L ] type
```

Each element of the format specification is a single character or number signifying a particular format option. The optional `argument` list provides values for the `width` and `precision` fields. The simplest format specification contains only the percent sign and a `type` character (for example, `%s`). The optional fields, appearing before the required `type` character, control other aspects of the formatting.

These are the fields in a **printf( )** format specification:

Field	**Description**
flags	Optional character or characters that control output justification and sign printing, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag can appear in a format specification. See also: Flag Directives
width	Optional number that specifies minimum number of output characters.
precision	Optional number that specifies maximum number of characters printed for all or part of the output field, or minimum number of digits printed for integer values. See also: Precision Specification
F, N	Optional prefixes that refer to the distance to the object being printed (near or far). F and N are not part of the ANSI definition for **printf( )**.
h, l, L	Optional prefixes that determine the size of the argument expected, as shown below:

h	Used with the integer types d, i, o, x, and X to specify that the argument is short integer, or with u to specify short unsigned int. If used with %p, it indicates a 16-bit pointer, which is ignored.
l	Used with d, i, o, x, and X types to specify that the argument is long integer, or with u to specify long unsigned integer; also used with e, E, f, g, and G types to specify double rather than float. If used with %p, it indicates a 32-bit pointer.
L	Used with e, E, f, g, and G types to specify long double. Also used with d, i, o, x, and b types to specify 64-bit integer.

type	Required character that determines whether the associated argument is interpreted as a character, a string, or a number. See also: Type Field Characters

If a percent sign is followed by a character that has no meaning as a format field, the character is copied to *stdout*. For example, to print a percent-sign character, use `%%`.

Flag Directives

These `flag` directives may appear in a format specification:

Flag	**Meaning**	**Default**
-	Left justify the result within the given field width.	Right justify.
+	Prefix the output value with a + or - sign if the output value is of a signed type.	- sign appears only for negative signed values.
0	If width is prefixed with 0, 0s are added until the minimum width is reached. If 0 and - appear, the 0 is ignored. If 0 is specified with an integer format (i, u, x, X, o, d), the 0 is ignored.	No padding.
blank	Prefix the output value with a blank if the output value is signed and positive; the blank is ignored if both the blank and + flags appear.	No blank appears.
#	When used with the o, x, or X format, the # flag prefixes any non-0 output value with 0, 0x, or 0X, respectively.	No blank appears.
	When used with the e, E, or f format, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	When used with the g or G format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing 0s.	Decimal point appears only if digits follow it. Trailing 0s are truncated.
	Ignored when used with c, d, i, u, or s.	

Width Specification

The `width` specification is a non-negative decimal integer that controls the minimum number of printed characters. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values, depending on whether the - flag is specified until the minimum width is reached. If width is prefixed with 0, **printf( )** adds `0`s until the minimum width is reached (not useful for left-justified numbers).

The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or `width` is not given, all characters of the value are printed, subject to the precision specification.

The width specification may be an asterisk (*), in which case an integer argument from the `argument` list supplies the value. The width specification must precede the value being formatted in the `argument` list. A nonexistent or small field width does not truncate a field; if the result of a conversion is wider than the field width, the field expands to contain the conversion result.

Precision Specification

The `precision` specification specifies a non-negative decimal integer, preceded by a period (.), which specifies the number of characters to print, the number of decimal places, or the number of significant digits. The precision specification can cause truncation of the output value, or rounding in the case of a double value. If **printf( )** specifies `precision` is `0` and the value to convert is `0`, the result is no characters output, as shown below:

```
printf( "%.0d", 0 );  /* No characters output */
```

The precision specification may be an asterisk (*), in which case an integer argument from the argument list supplies the value. The precision argument must precede the value being formatted in the argument list.

The interpretation of the precision value and the default precision (if omitted) depend on the `type`, as shown below:

Type	**Meaning**	**Default**
d, i, u, o, x, X	The precision specifies the minimum number of digits to print. If the number of digits in the argument is less than precision, the output value is padded on the left with 0s. The value is not truncated when the number of digits exceeds precision.	If precision is 0 or omitted entirely, or if the period (.) appears without a number following it, the precision is set to 1.
e, E	The precision specifies the number of digits to print after the decimal point. The last printed digit is rounded.	Default precision is 6; if precision is 0 or the period (.) appears without a number following it, no decimal point is printed.

Type	**Meaning**	**Default**
f	The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 6; if precision is 0, or if the period (.) appears without a number following it, no decimal point is printed.
g, G	The precision specifies the maximum number of significant digits printed. If specified as 0, treated as 1.	Six significant digits are printed, with any trailing 0s truncated.
c	The precision has no effect.	Character is printed.
s	The precision specifies the maximum number of characters to print. Characters in excess of precision are not printed.	Characters are printed until a null character is encountered.

If the argument corresponding to a double specifier is infinite, indefinite, or not a number (NAN), the **printf( )** function gives this output:

Value	**Output**
+ infinity	1.#INF*random-digits*
- infinity	-1.#INF*random-digits*
Indefinite	*digit*.#IND*random-digits*
Not a number (NAN)	*digit*.#NAN*random-digits*

Distance and Size Specification

The format specification fields `F` and `N` refer to the distance to the object being read (near or far), and `h` and `l` refer to the size of the object being read (16-bit short or 32-bit long). The `F` and `N` specifications are accepted, for compatibility with other compilers, but they are ignored. This list provides some example usage of `F`, `N`, `h`, `l,` and `L`.

Program Code	**Action**
printf ("%Ns");	Print near string
printf ("%Fs");	Print far string
printf ("%Nn");	Store char count in near int
printf ("%Fn");	Store char count in far int
printf ("%hp");	Print a 16-bit pointer (xxxxxxxx)
printf ("%lp");	Print a 32-bit pointer (xxxxxxxx)
printf ("%Nhn");	Store char count in near short int
printf ("%Nln");	Store char count in near long int
printf ("%Fhn");	Store char count in far short int
printf ("%Fln");	Store char count in far int

The specifications `"%hs"` and `"%ls"` are meaningless to **printf**( ). The specifications `"%Np"` and `"%Fp"` are aliases for `"%hp"` and `"%lp"` for compatibility with earlier compilers.

Type Field Characters

The `type` character is the only required format field for the **printf( )** function. It appears after any optional format fields and determines how the associated argument is interpreted.

Char	**Type**	**Output Format**
d	int	Signed decimal integer.
i	int	Signed integer.
u	int	Unsigned decimal integer.
o	int	Unsigned octal integer.
x	int	Unsigned hexadecimal integer, using abcdef.
X	int	Unsigned hexadecimal integer, using ABCDEF.
f	double	Signed value having the form [-]dddd.dddd, where dddd is one or more decimal digits, depending upon the magnitude of the number, and the requested precision.
e	double	Signed value having the form [-]d.dddd e [sign]ddd, where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or -.
E	double	Same as the e format, except that E introduces the exponent.
g	double	Signed value printed in f or e format (the one most compact for the given value and precision). e is used only when the exponent of the value is less than -4 or greater than or equal to the precision. Trailing 0s are truncated and the decimal point appears only if any digits follow it.

Char	**Type**	**Output Format**
G	double	Same as the g format, except that G introduces the exponent (where appropriate).
c	int	Single character.
s	string	Characters printed up to the first null character \0 or until the precision value is reached.
n	pointer	Points to number of characters successfully written so far to the stream or buffer; this value is stored in the integer whose address is given as the argument.
p	pointer	Prints the address pointed to by the argument in a form dependent on the memory model: 16-bit large or compact model caller: xxxx:yyyy which is <segment>:<16-bit offset> 32-bit compact model caller: xxxx:yyyyyyyy which is <segment>:<32-bit offset> 32-bit flat model caller: yyyyyyyy which is <32-bit offset> only

See also: fprintf( ), scanf( ), sprintf( ), vfprintf( ), vprintf( ), vsprintf( )

Returns

The number of characters printed.

A negative value on error.

putc, putchar

Putc( ) writes a character to a specified stream at the current position; **putchar( )** writes to *stdout*.

Syntax

```
#include <stdio.h>
      int putc (int c, FILE *stream);
      int putchar (int c);
```

Parameters

`c`	Character to be written.
`stream`	Pointer to `FILE` structure.

Additional Information

The **putchar( )** function is identical to:

```
putc (c, stdout)
```

Any integer can be passed to **putc( )**, but it only writes the lower 8 bits.

These functions are implemented as both macros and functions.

See also: fputc( ), fputchar( ), getc( ), getchar( )

Returns

The character written.

EOF on error.

putch

Writes a character directly (without buffering) to the console.

Syntax

```
#include <conio.h>
      int putch (int c);
```

Parameter

c	Character to be output.

See also: getch( ), getche( )

Returns

Value	**Meaning**
c	Successful
EOF	Unsuccessful

putenv

Adds new environment variables or modifies the values of existing ones.

Syntax

```
#include <stdlib.h>
      int putenv (const char *envstring);
```

Parameter

`envstring`

Environment-variable table entry definition, which must be a character string of this form:

```
varname = string
```

Where:

`varname`	The name of the environment variable to be added or modified.
`string`	The variable's value. A space character is required on both sides of the equal sign for **fscanf( )** parsing.

Additional Information

Environment variables customize the environment in which a task executes. This function affects only the current environment; it does not modify the environment-variable table files.

If `varname` is already part of the environment, its value is replaced by `string`; otherwise, the new variable is placed in the first empty slot in the environment-variable table. If you specify a valid `varname` and null `string`, the environment variable is removed.

There is one environment-variable table shared by all tasks using the C library. If the table has not been initialized by a previous call to **getenv( )**, **putenv( )** first calls **getenv( )** before proceeding.

See also: **getenv( )**, in this manual
Environment variables, *System Configuration and Administration*

Returns

Value	**Meaning**
0	Successful
-1	Error occurred

_put_rmx_conn

Places an iRMX connection token into the file descriptor table and returns a valid file descriptor, usable as an argument in C library calls.

Syntax

```
#include <rmx_c.h>
      int _put_rmx_conn (selector connection);
```

Parameter

`connection`

Valid iRMX file connection token.

Additional Information

Use this function in code that mixes direct iRMX system calls with C library functions.

A file descriptor table, managed internally by the C library, is associated with each task using the library. This table maps C file descriptors to iRMX file connections. The table is fixed in size. The maximum number of open files per task is 32 for compatibility with UNIX systems process limit.

See also: *<rmx_c.h>*, **_get_rmx_conn**

Returns

A valid file descriptor for the iRMX connection token.

-1 if unsuccessful.

puts

Writes a string to *stdout*, replacing the string's terminating null character `\0` with a newline character `\n`.

Syntax

```
#include <stdio.h>
      int puts (const char *string);
```

Parameter

`string`	String to be output.

See also: fputs( ), gets( )

Returns

A non-negative value.

EOF if unsuccessful.

putw

Writes an integer to the current position of a stream.

Syntax

```
#include <stdio.h>
      int putw (int binint, FILE *stream);
```

Parameters

`binint`	Binary integer to be output.
`stream`	Pointer to `FILE` structure.

Additional Information

The **putw( )** function does not affect the alignment of items in the stream, nor does it assume any special alignment.

See also: **getw( )**

Returns

The value written.

EOF on error. Since EOF is also a legitimate integer value, use **ferror( )** to verify an error.

qsort

Performs a quick sort of an array, overwriting the input array with the sorted elements.

Syntax

```
#include <stdlib.h>
      #include <search.h>
      void qsort (void *base, size_t num, size_t width,
                  int (*compare)(const void *elem1,
                  const void *elem2));
```

Parameters

`base`	Pointer to the base of the array to be sorted and overwritten.
`num`	Array size in number of elements.
`width`	Element size in bytes.

`compare`

Pointer to a user-supplied routine that compares two array elements (`elem1` and `elem2`) and returns a value specifying their relationship:

Value	**Meaning**
< 0	elem1 less than elem2
= 0	elem1 equivalent to elem2
> 0	elem1 greater than elem2

`elem1`	Pointer to the key for the sort.
`elem2`	Pointer to the array element to compare with the key.

Additional Information

The **qsort( )** function calls the `compare` routine one or more times during the sort, passing pointers to two array elements on each call:

```
compare (( void *) elem1, (void *) elem2);
```

The function sorts the array in ascending order, as defined by the `compare` routine. To sort the array in descending order, reverse the sense of greater-than and less-than in the `compare` routine.

See also: bsearch( ), lsearch( )

Returns

Nothing.

raise

Sends a signal to the executing program.

Syntax

```
#include <signal.h>
      int raise (int sig);
```

Parameter

`sig`	Signal to send.

Additional Information

If a signal-handling routine for `sig` has been installed by a prior call to **signal( )**, **raise( )** causes that routine to execute. Signal-handling is maintained locally to the calling task, not globally to all tasks using the C library.

If no handler routine has been installed for a particular signal, the default signal-handling is as follows:

Signal	**Meaning**	**Default Action**
SIGABRT	Abnormal termination	Calls _exit(3)
SIGALLOC	Memory allocation failure	Returns without error
SIGBREAK	<Ctrl-Break> signal	Ignored
SIGFPE	Floating-point exception	Calls _exit(3)
SIGFREE	Bad free pointer	Calls _exit(3)
SIGILL	Illegal instruction	Calls _exit(3)
SIGINT	Interactive attention	Calls _exit(3)
SIGREAD	Read error	Ignored
SIGSEGV	Segment violation	Sets errno to EDOM and returns
SIGTERM	Termination request	Calls _exit(3)
SIGUSR1	User-defined	Ignored
SIGUSR2	User-defined	Ignored
SIGUSR3	User-defined	Ignored
SIGWRITE	Write error	Ignored

See also: abort( ), _exit( ), signal( )

This function is implemented in the C interface library (not in the shared C library), and is private to each application.

Returns

Value	**Meaning**
0	Successful
Not 0	Unsuccessful

rand

Generates a pseudo-random number.

Syntax

```
#include <stdlib.h>
      int rand (void);
```

Additional Information

Use the **srand( )** function to seed the pseudo-random-number generator before calling **rand( )**.

See also: **srand( )**

Returns

A pseudo-random integer in the range 0 to RAND_MAX.

No error return.

read

Reads the specified number of bytes from a file into a buffer, beginning at the current position of the file pointer.

Syntax

```
#include <io.h>
      int read (int handle, char *buffer, unsigned int count);
```

Parameters

`handle`	Descriptor referring to an open file.
`buffer`	Storage location for data.
`count`	Maximum number of bytes to read.

Additional Information

After the read operation, the file pointer points to the next unread character.

In text mode, each <CR><LF> pair is replaced with a single <LF> character. Only the single <LF> character is counted in the return value. The replacement does not affect the file pointer.

See also: creat( ), fread( ), open( ), write( )

Returns

The number of bytes actually read, usually `count`. Less than `count` if there are fewer than `count` bytes left in the file, or if the file was opened in text mode.

0 indicates an attempt to read at end-of-file.

-1 indicates an error, and the function sets **errno** to EBADF, indicating that the given descriptor is invalid, the file is not open for reading, or the file is locked.

readdir

Reads a directory and then returns a pointer to a **dirent** structure representing the next directory entry in the directory stream pointed to be dir. It returns NULL on reaching the end-of-file or if an error occurred.

The data returned by this call is overwritten by subsequent calls to readdir() for the same directory stream.

According to POSIX, the **dirent** structure contains a field, char d_name[], of unspecified size, with at most NAME_MAX characters preceding the terminating null character. Use of other fields will harm the portability of your programs.

Syntax

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dir);
```

Returns

Returns a pointer to a **dirent** structure, or NULL if an error occurs or end-of-file is reached.

EBADF	Invalid directory stream descriptor dir.
See also:	read(2), opendir(3), closedir(3), rewinddir(3)

realloc

Changes the size of a previously allocated memory block or allocates a new one.

Syntax

```
#include <stdlib.h>
      void *realloc (void *memblock, size_t size);
```

Parameters

`memblock`
: Pointer to the beginning of the previously allocated memory block or to a block that has been freed, as long as there has been no intervening call to the corresponding **calloc( )**, **malloc( )**, or **realloc( )** function.

`size` New size in bytes.

Additional Information

If `memblock` is a null pointer, **realloc( )** functions in the same way as **malloc( )** and allocates a new block of `size` bytes. If `memblock` is not a null pointer, it should be a pointer returned by **calloc( )**, **malloc( )**, or a prior call to **realloc( )**.

The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block may be in a different location.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than `void`, use a type cast on the return value.

See also: calloc( ), free( ), malloc( )

Returns

A void pointer to the reallocated (and possibly moved) memory block. The reallocated block is marked in use.

A null pointer if `size` is 0 and the `memblock` argument is not a null pointer, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second, the original block is unchanged.

rename

Renames a file or directory.

Syntax

```
#include <stdio.h>
      #include <io.h>
      int rename (const char *oldname, const char *newname);
```

Parameters

`oldname`
Pathname of an existing file or directory to change.

`newname`
Pathname of a new file or directory.

Additional Information

This function invokes the system call **rq_s_rename_file** to rename the file or directory to the new name.

See also: **rq_s_rename_file**, *System Call Reference*

Returns

Value	**Meaning**	
0	Successful	
Not 0	Unsuccessful and the function sets **errno** to one of these values:	
	EACCES	File or directory specified by newname already exists or could not be created (invalid path); or
		oldname is a directory and newname specifies a different path.
	ENOENT	File or pathname specified by oldname not found.
	EXDEV	Attempt to move a file to a different device.

rewind

Repositions the file pointer to the beginning of a file and clears the end-of-file indicator.

Syntax

```
#include <stdio.h>
      void rewind (FILE *stream);
```

Parameter

`stream` Pointer to `FILE` structure.

Additional Information

A call to **rewind( )** is nearly equivalent to:

```
(void) fseek (stream, 0L, SEEK_SET);
```

Rewind( ) clears the error indicators for the `stream`; **fseek( )** does not. **Fseek( )** returns a value that indicates whether the pointer was successfully moved; **rewind( )** does not.

You can use the **rewind( )** function to clear the keyboard buffer. Specify *stdin*, associated with the keyboard by default, as `stream`.

Returns

Nothing.

rewinddir

Resets the position of the directory stream to the beginning of the directory.

Syntax

```
#include <sys/types.h>
#include <dirent.h>

void rewinddir(DIR *dir);
```

Returns

Nothing.

See also: opendir(3), readdir(3), closedir(3)

rmdir

Deletes a directory.

Syntax

```
#include <direct.h>
      int rmdir (const char *dirname);
```

Parameter

`dirname`

Pathname of the directory to be removed. The directory must be empty, and it must not be the current working directory or the root directory.

See also: **mkdir( )**

Returns

Value	**Meaning**
0	Successful
-1	Unsuccessful and the function sets **errno** to one of these values:

E	The given pathname is not a directory; or the directory is not empty; or the directory is the current working directory or the root directory.
E	Pathname not found.

rmtmp

Removes all the temporary files that were created by **tmpfile( )** from the current directory.

Syntax

```
#include <stdio.h>
      int rmtmp (void);
```

Additional Information

Use **rmtmp( )** only in the same directory in which the temporary files were created.

See also: flushall( ), tmpfile( ), tmpnam( )

Returns

The number of temporary files closed and deleted.

sbrk

Creates iRMX segments of the specified number of bytes.

Syntax

```
#include <stdlib.h>
      void *sbrk (unsigned segsize);
```

Parameter

`segsize`
Number of bytes to be acquired; must be greater than 0.

Additional Information

For non-flat model applications, this function uses the system call **rq_create_segment**. To return segments acquired by **sbrk( )** to the memory pool, use the system call **rq_delete_segment**.

For flat model applications, **sbrk( )** uses the system call **rqv_allocate** instead of **rq_create_segment**. Also, you should use **rqv_free**, instead of **rq_delete_segment**, to delete segments acquired by **sbrk( )**.

To return the created segment to the heap using **free( )** or **realloc( )**, use **malloc( )** to get memory instead of **sbrk( )**.

See also:	free( ), malloc( ), realloc( ), in this manual
	rq_create_segment, rq_delete_segment, *System Call Reference*

Returns

The address of the acquired memory area.

A null pointer if the allocation request cannot be satisfied.

scanf

Reads from *stdin* at current position, and formats character data.

Syntax

```
#include <stdio.h>
      int scanf (const char *format [,argument]...);
```

Parameters

`format` Null-terminated format-control string, which determines the interpretation of the input field. Can contain whitespace and nonwhitespace characters, and format specifications.

`argument`
Optional argument(s), which may include the location to read to; must be a pointer to a variable corresponding to a type specified in the `format` argument. If there are too many arguments for the given `format`, the extra arguments are evaluated but ignored. The results are unpredictable if there are not enough arguments.

Additional Information

The **scanf( )** function reads all characters in *stdin* up to the first whitespace character (space, tab, or newline), or the first character that cannot be converted according to `format`; this is the input field.

The format string is read from left to right. A whitespace character in `format` causes **scanf( )** to read, but not store, all consecutive whitespace characters in the input field up to the next nonwhitespace character. A nonwhitespace character in `format` causes **scanf( )** to read, but not store, all matching characters. A format specification causes **scanf( )** to read and convert applicable characters in the input field into values of a particular type, to be stored in the optional arguments as they are read from *stdin*.

Format specifications always have a preceding percent sign (`%`) followed by a format-control character. Additional optional format-control characters may also appear. If `%` is followed by a character that has no meaning as a format-control character, that character and these characters (up to the next `%`) are treated as an ordinary sequence of characters that is, a sequence of characters that must match the input. For example, to specify a percent-sign character to be input, use `%%`.

An asterisk (*) following the `%` suppresses storage of the next input field that is interpreted as a field of the specified type. The field is scanned but not stored.

If a character in *stdin* conflicts with the format specification, **scanf( )** terminates. The character is left in *stdin* as if it had not been read.

Here are some example **scanf( )** statements:

Statement	**Meaning**
scanf("%Ns", &x);	Read a string into memory
scanf("%Fs", &x);	Read a string into memory
scanf("%Nd", &x);	Read an int into memory
scanf("%Fd", &x);	Read an int into memory
scanf("%Nld", &x);	Read a long int into memory
scanf("%Fld", &x);	Read a long int into memory
scanf("%Nhp", &x);	Read a 16-bit pointer into memory
scanf("%Nlp", &x);	Read a 32-bit pointer into memory
scanf("%Fhp", &x);	Read a 16-bit pointer into memory
scanf("%Flp", &x);	Read a 32-bit pointer into memory

Format Specification

A format specification, which consists of optional and required fields, has this form:

```
%[*] [width] [{F | N}] [{h | l}]type
```

Each field of the format specification is a single character or number signifying a particular format option. The optional fields appear before the required `type` character. These are the fields in a **scanf( )** format specification:

Field	**Description**
width	A positive decimal integer controlling the maximum number of characters to be read from *stdin*. No more than width characters are converted and stored at the corresponding argument. Fewer than width characters may be read if a white-space character (space, tab, or newline) or a character that cannot be converted according to the given format occurs before width is reached.
F, N	The optional F and N prefixes are accepted for compatibility with other compilers, but they are ignored. F and N refer to the distance to the object being read in (far or near). The F and N prefixes are not part of the ANSI definition for **scanf( )** and should not be used when ANSI portability is desired.

Field	**Description**
h, l, L	Optional prefixes that determine the type required for the argument expected (l and h are ignored if specified for any other type), as shown below: h Used with the integer types d, i, o, x, and X to specify that the argument is short integer, or with u to specify short unsigned int. If used with %p, it indicates a 16-bit pointer, which is ignored. l Used with d, i, o, x, and X type characters to specify that the argument is long integer, or with u to specify long unsigned integer; also used with e, E, f, g, and G types to specify double rather than float. If used with %p, it indicates a 32-bit pointer. L Used with e, E, f, g, and G types to specify long double. Also used with integer types to specify that the argument is a 64-bit data type.
type	Required character that determines the required type for the associated argument.

Type Field Characters

These are the `type` characters and their meanings:

Character	**Input Type**	**Argument Type**
d	Decimal integer	Pointer to int.
o	Octal integer	Pointer to int.
x	Hex integer	Pointer to int. Since the input for %x format specifier is always interpreted as a hexadecimal number, the input should not include a leading 0x. (If 0x is included, the 0 is interpreted as a hexadecimal input value.)
i	Decimal, hexadecimal, or octal integer.	Pointer to int.
u	Unsigned decimal integer	Pointer to unsigned int.
U	Unsigned decimal integer	Pointer to unsigned int.

Character	**Input Type**	**Argument Type**
e, E, f, g, G	Double. Value consisting of an optional sign (+ or -), a series of one or more digits containing a decimal point, and an optional exponent (e or E) followed by an optionally signed integer value.	Pointer to double.
1c	Character. Whitespace characters that are ordinarily skipped are read when c is specified; to read the next nonwhitespace character, use %1s.	Pointer to char.
s	String.	Pointer to character array large enough for input field plus a terminating null character \0, which is automatically appended.
n	No input read.	Pointer to int, into which the number of characters successfully read is stored.
p	Address in a form dependent on the memory model: 16-bit large or compact model caller: xxxx:yyyy which is <segment>:<16-bit offset> 32-bit compact model caller: xxxx:yyyyyyyy which is <segment>:<32-bit offset> 32-bit flat model caller: yyyyyyyy which is <32-bit offset>	Pointer to pointer to void.

Additional Information

To read strings not delimited by space characters, substitute a set of characters in brackets (`[ ]`) for the `s` (string) type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a caret (^), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

The format specifications `%[a-z]` and `%[z-a]` are interpreted as equivalent to `%[abcde...z]`. This is not required by the ANSI specification.

To store a string without storing a terminating null character `\0`, use the specification `%nc`, where `n` is a decimal integer. Then the `c type` character indicates that the argument is a pointer to a character array. The next `n` characters are read from the input stream into the specified location, and no null character `\0` is appended. If `n` is not specified, the default value for it is 1.

See also: fscanf( ), printf( ), sscanf( ), vfprintf( ), vprintf( ), vsprintf( )

Returns

The number of fields converted and assigned, which may be less than the number requested. Does not include fields that were read but not assigned.

EOF if the end-of-file is encountered in the first attempt to read a character.

setbuf

Allows the user to control buffering for a stream.

Syntax

```
#include <stdio.h>
      void setbuf (FILE *stream, char *buffer);
```

Parameters

`stream`	Pointer to `FILE` structure; must refer to an open stream file that has not been read or written.
`buffer`	User-allocated buffer.

Additional Information

If the `buffer` argument is a null pointer, the stream is unbuffered. If not, the buffer must point to a character array of length BUFSIZ. This user-specified buffer is used for I/O buffering instead of the default system-allocated buffer for the given stream.

The *stderr* stream is unbuffered by default, but may be assigned buffers with **setbuf( )**.

Use the **setvbuf( )** function for new code; **setbuf( )** is retained for compatibility with existing code.

See also: fclose( ), fopen( ), setvbuf( )

Returns

Nothing.

_set_info

Modifies the `num_eios_bufs` (number of EIOS buffers per open file connection) field for a task in the C library information structure `CINFO_STRUCT`.

Syntax

```
#include <rmx_c.h>
      int _set_info (unsigned int count, CINFO_STRUCT *cinfo);
```

Parameters

`count`	Number of elements in `CINFO_STRUCT`, obtained from **cinfo_count** constant.
`cinfo`	Pointer to `CINFO_STRUCT` for a task.

Additional Information

All of the other fields in `CINFO_STRUCT` are read-only.

Verify the change using the **_get_info( )** function.

See also: **_get_info( )**, *<rmx_c.h>*

Returns

Value	**Meaning**
0	Successful
-1	Unsuccessful

setjmp

Saves the current context of the executing program and stores it in the specified location.

Syntax

```
#include <setjmp.h>
      int setjmp (jmp_buf context);
```

Parameter

`context`

Structure in which the current context is stored.

Additional Information

The **jmp_buf** structure is usable only as an argument for the subsequent **longjmp( )** call; **jmp_buf** is defined internally to the C library.

Used together, **setjmp( )** and **longjmp( )** provide a way to execute a nonlocal goto. They typically pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A subsequent call to **longjmp( )** restores the context and resumes execution at the point **setjmp( )** was called. All local variables except register variables, accessible to the routine receiving control, contain the values they had when **setjmp( )** was called. Global variables are unaffected.

See also: **longjmp( )**, *<setjmp.h>*

Returns

0 after saving the context of the executing program.

When **setjmp( )** returns as a result of a **longjmp( )** call, it returns the `value` argument of **longjmp( )** or returns 1 if the `value` argument of **longjmp( )** is 0.

No error return.

setlocale

Sets the task's current entire locale or specified portions of it.

Syntax

```
#include <locale.h>
      char *setlocale (int category, const char *locale);
```

Parameters

`category`
Specifies which portion of a task's locale information to use.

`locale` Pointer to a string containing the name of the locale for which certain aspects of your program can be customized. C specifies the minimal ANSI-conforming locale for C translation. If `locale` points to an empty string, the locale is the implementation-defined native locale.

Additional Information

Some locale-dependent aspects include the formatting of dates and the display format for monetary values.

These are the manifest constants used for the `category` argument and the parts of the program affected:

Value	**Program Parts Affected**
LC_ALL	All categories listed below.
LC_COLLATE	The **strcoll( )** and **strxfrm( )** functions.
LC_CTYPE	The character-handling functions except for **isdigit( )** and **isxdigit( )**, which are unaffected.
LC_MONETARY	Monetary formatting information returned by the **localeconv( )** function.
LC_NUMERIC	Decimal point character for the formatted output functions such as **printf( )**, for the data conversion functions, and for the nonmonetary formatting information returned by the **localeconv( )** function.
LC_TIME	The **strftime( )** function.

See also: localeconv( ), strcoll( ), strftime( ), strxfrm( )

Returns

One of these:

- A pointer to the string associated with the specified category for the new locale. Use the pointer in subsequent calls to restore that part of the program's locale information. Later calls to **setlocale( )** will overwrite the string.
- A pointer to the string associated with the category of the program's locale. It does not change the program's current locale setting if the `locale` argument is a null pointer.
- A null pointer. It does not change the program's current locale settings if the locale or category is invalid.

setmode

Sets binary or text translation mode of a file.

Syntax

```
#include <fcntl.h>
      #include <io.h>
      int setmode (int handle, int mode);
```

Parameters

`handle`	Descriptor referring to an open file.
`mode`	New translation mode.

Additional Information

The `mode` must be one of these manifest constants:

Value	**Meaning**
O_TEXT	Sets text (translated) mode. <CR><LF> combinations are translated into a single <LF> character on input. <LF> characters are translated into <CR><LF> combinations on output.
O_BINARY	Sets binary (untranslated) mode and suppresses the above translations.

The **setmode( )** function is typically used to modify the default translation mode of *stdin*, *stdout*, and *stderr*, but can be used on any file.

Note

If multiple tasks or jobs are collecting data from the same file or stream, use binary mode. Otherwise, the task or job receives scrambled data.

Do not try to change a stream's mode while the stream buffer is active. Call **fflush( )** first.

See also: creat( ), fopen( ), open( )

Returns

The previous translation mode.

-1 on error, and the function sets **errno** to one of these values:

EBADF	Invalid file descriptor.
EINVAL	Invalid mode argument (neither O_TEXT nor O_BINARY).

setvbuf

Controls stream buffering and buffer size.

Syntax

```
#include <stdio.h>
      int setvbuf (FILE *stream, char *buffer, int mode,
                   size_t size);
```

Parameters

`stream`	Pointer to `FILE` structure; must refer to an open stream file that has not been read from or written to since it was opened.
`buffer`	Pointer to a user-allocated character array used for buffering. If a null pointer references `buffer`, a buffer of `size` bytes is automatically allocated.
`mode`	Buffering mode.

Value	**Meaning**
_IOFBF	Full buffering; that is, buffer is used as the buffer and size is used as the size of the buffer.
_IONBF	No buffer is used, regardless of buffer or size.

`size`	Size of buffer. Legal values are greater than 0 and less than INT_MAX.

See also: **fclose( )**, **fopen( )**, *<limits.h>*, **setbuf( )**

Returns

Value	**Meaning**
0	Successful
Not 0	An illegal type or buffer size was specified

signal

Sets up one of several ways for a task to handle an interrupt signal from the OS.

Syntax

```
#include <signal.h>
      void (*signal (int sig, void (*func)(int sig [,int
      subcode])))
                    (int sig);
```

Parameters

`sig` Signal value. Must be one of the manifest constants defined in <*signal.h*>

`func` Specifies what action is taken. Must be either a function address or one of the manifest constants defined in <*signal.h*>.

`subcode`
Optional subcode to the signal number.

Additional Information

This function is implemented in the shared C library interface library (not in the shared C library), and is private to each application.

The `sig` argument must be one of these manifest constants:

Value	**Meaning**
SIGABRT	Abnormal termination
SIGALLOC	Memory allocation failure
SIGBREAK	<Ctrl-Break> signal
SIGFPE	Floating-point exception
SIGFREE	Bad free pointer
SIGILL	Illegal instruction
SIGINT	Interactive attention
SIGREAD	Read error
SIGSEGV	Segment violation
SIGTERM	Termination request
SIGUSR1	User-defined
SIGUSR2	User-defined
SIGUSR3	User-defined
SIGWRITE	Write error

The `func` must be either a function address or one of these manifest constants:

Value	**Meaning**
SIG_DFL	Uses system-default response. The system-default response for all signals except SIGUSR1, SIGUSR2, and SIGUSR3 is to abort the calling program using **_exit( )**. The default response for SIGUSR1, SIGUSR2, and SIGUSR3 is to ignore the signal.
SIG_IGN	Ignores interrupt signal. This value should never be given for SIGFPE, since the floating-point state of the process is left undefined.
Function address	Installs the specified function as the handler for the given signal.

Additional Information

For all signals except SIGFPE and SIGUSRx, the function is passed the `sig` argument and executed.

For SIGFPE, the function pointed to by `func` is passed two arguments, SIGFPE and an integer error subcode, FPE_xxx; then the function is executed. The value of `func` is not reset upon receiving the signal. To recover from floating-point exceptions, use **setjmp( )** in conjunction with **longjmp( )**. If the function returns, the calling task resumes execution with the floating-point state of the process left undefined.

If the function returns, the calling task resumes execution immediately following the point at which it received the interrupt signal. This is true regardless of the type of signal or operating mode.

Before the specified function is executed, the value of `func` is set to SIG_DFL. The next interrupt signal is treated as described above for SIG_DFL, unless an intervening call to **signal( )** specifies otherwise. This allows the program to reset signals in the called function.

Since signal-handler routines are normally called asynchronously when an interrupt occurs, it is possible that your signal-handler function will assume control when an operation is incomplete and in an unknown state. Certain restrictions therefore apply to the C functions used in your signal-handler routine:

- Do not issue low-level or standard I/O functions, for example, **printf( )**, **read( )**, **write( )**, and **fread( )**.
- Do not call heap routines or any function that uses the heap routines, for example, **malloc( )**, **strdup( )**, or **putenv( )**.
- Do not use the **longjmp( )** function.

See also: **abort( )**, **raise( )**, **_exit( )**, <*signal.h*>

Returns

The previous value of `func`. For example, if the previous value of `func` was `SIG_IGN`, the return value will be SIG_IGN.

-1 on error such as invalid `sig` or `func` values, and the function sets **errno** to EINVAL.

sin, sinh

Sin calculates the sine and **sinh** calculates the hyperbolic sine of an angle.

Syntax

```
#include <math.h>
      double sin (double x);
      double sinh (double x);
```

Parameter

x	Angle in radians.

See also: acos( ), asin( ), atan( ), cos( ), tan( )

Returns

Sin( ) Returns the sine of `x`.

Generates a PLOSS error if `x` is large and partial loss of significance in the result occurs; function sets **errno** to ERANGE.

Prints a TLOSS message to *stderr* and returns 0 if `x` is so large that significance is completely lost; function sets **errno** to ERANGE.

Sinh( ) Returns the hyperbolic sine of `x`.

Returns ±HUGE_VAL, and the function sets **errno** to ERANGE if the result is too large.

These functions do not return standard ANSI domain or range errors.

sleep

Suspends a task for a specified number of seconds.

Syntax

```
#include <process.h>
      unsigned int sleep (unsigned int seconds);
```

Parameter

`seconds`
Number of seconds to suspend a task.

Additional Information

This function invokes the system call **rq_sleep**.

See also: **rq_sleep**, System Call Reference

Returns

Always returns 0.

sopen

Opens a file for shared reading or writing.

Syntax

```
#include <fcntl.h>
      #include <share.h>
      #include <sys/stat.h>
      #include <io.h>
      int sopen (const char *filename, int oflag, int shflag,
                 int pmode);
```

Parameters

`filename`
Filename to be opened.

`oflag` Type of operations allowed (open mode). Combine one or more of the manifest constants described in **open( )** with the bitwise-OR operator (`|`).

`shflag` Type of sharing allowed (share mode).

`pmode` Permission mode, which specifies the file's ownership and access rights; required only when O_CREAT is specified. Otherwise, argument is ignored. The manifest constants are described in **chmod( )**. Join them with the bitwise-OR operator (`|`).

Additional Information

`Shflag` must be one of these manifest constants:

Value	**Meaning**
SH_DENYRW	Denies read and write access to file.
SH_DENYWR	Denies write access to file.
SH_DENYRD	Denies read access to file.
SH_DENYNO	Permits read and write access.

Ownership and access rights are set when the new file is closed for the first time.

The **sopen( )** function applies the default file-permission mask (set with the **umask( )** function) to `pmode` before setting the permissions.

This function performs a translation of POSIX file ownership rights and POSIX access rights to the iRMX OS equivalent as described in **chmod( )**.

See also: close( ), creat( ), fopen( ), open( ), umask( )

Returns

A descriptor for the opened file.

-1 indicates an error, and the function sets **errno** to one of these values:

EACCES	Given pathname is a directory; or The file is read-only but an open for writing was attempted; or A sharing violation occurred because the file's share mode does not allow the specified operations.
EEXIST	The O_CREAT and O_EXCL flags are specified, but the named file already exists.
EINVAL	An invalid `oflag` or `shflag` argument was given.
EMFILE	No more file descriptors available (too many open files).
ENOENT	File or pathname not found.

sprintf

Prints formatted data to a string.

Syntax

```
#include <stdio.h>
      int sprintf (char *buffer, const char *format [,
      argument]...);
```

Parameters

`buffer`	Output string.
`format`	Formatted string consisting of ordinary characters, escape sequences, and, if arguments appear, format specifications. The `format` and optional arguments have the same form and function as the **printf( )** function.
`argument`	Optional arguments.

Additional Information

The ordinary characters and escape sequences are copied to `buffer` in order of their appearance.

A null character `\0` is appended to the end of the characters written.

See also: fprintf( ), printf( ), sscanf( )

Returns

The number of characters stored in `buffer`, not counting the terminating null character.

sqrt

Calculates the square root of a number.

Syntax

```
#include <math.h>
      double sqrt (double x);
```

Parameter

`x` Nonnegative value to calculate root for.

See also: exp( ), log( ), matherr( ), pow( )

Returns

The square-root result.

0 if `x` is negative, prints a DOMAIN error message to *stderr* and sets **errno** to EDOM.

This function does not return standard ANSI domain or range errors.

square

Calculates the square of a number.

Syntax

```
#include <math.h>
      double square (double x);
```

Parameter

x	Number to be squared.

See also: exp( ), log( ), matherr( ), pow( )

Returns

The square result.

This function does not return standard ANSI domain or range errors.

srand

Sets the starting point for generating a series of pseudorandom integers.

Syntax

```
#include <stdlib.h>
      void srand (unsigned int seed);
```

Parameter

`seed`	Starting point for random-number generation. Use 1 to reinitialize the generator.

Additional Information

The **rand( )** function retrieves pseudorandom numbers. Calling **rand( )** before any call to **srand( )** generates the same sequence as calling **srand( )** with `seed` passed as 1.

See also: **rand( )**

Returns

Nothing.

sscanf

Reads and formats character data from a string.

Syntax

```
#include <stdio.h>
      int sscanf (const char *buffer, const char *format
                  [, argument ]...);
```

Parameters

`buffer` Source string.

`format` Null-terminated format-control string which controls the interpretation of the input fields and has the same form and function as the `format` argument as in the **scanf( )** function.

`argument`
Optional argument. Must be a pointer to a variable with a type that corresponds to a type specifier in `format`.

Additional Information

Reads data from `buffer` into the locations given by `argument` (if any).

The **sscanf( )** function reads all characters in `buffer` up to the first whitespace character (space, tab, or newline), or the first character that `format` cannot convert. If there are too many arguments for the given `format`, the extra arguments are evaluated but ignored. The results are unpredictable if there are not enough arguments for the format specification.

See also: fscanf( ), scanf( ), sprintf( )

Returns

The number of fields that were successfully converted and assigned, but not fields that were read but not assigned.

0 if no fields were assigned.

EOF if the attempted read was at end-of-string.

stat

Gets information on a file.

Syntax

```
#include <sys/types.h>
      #include <sys/stat.h>
      int stat (const char *filename, struct stat *buffer);
```

Parameters

`filename`
Pathname of an open file to get information on.

`buffer` Pointer to file-status structure `stat`. The fields of `stat` are described in <*sys/stat.h*>.

Additional Information

Stat( ) invokes the system call **rq_a_get_file_status** and adds the number of seconds between epoch time and January 1, 1978, plus the local timezone factor, an environment variable described in **tzset( )**. This adjusts the time stamps of iRMX files to POSIX-standard values.

Stat( ) caches up to two directory connections and the associated pathnames to provide a performance boost for tasks that make repeated calls to **stat( )** for files under either of the two cached directories. The cache reduces the overhead incurred while parsing a long pathname and attaching each directory along the way. If the directory is not in the cache, a connection is obtained through a call to **rq_attach_file**, and entered into the cache. The oldest entry in the two-deep cache is then deleted. The cache is part of the single C library environment; required memory is allocated on the first use of **stat( )**.

This function performs a translation of iRMX OS file ownership rights and iRMX OS access rights to POSIX as described in <*sys/stat.h*>.

See also: chmod( ), filelength( ), fstat( ), <*sys/stat.h*>

Returns

Value	**Meaning**
0	File-status information is obtained
-1	Error occurred; the function sets **errno** to EBADF, indicating an invalid filename

strcat

Appends a null-terminated string to another string.

Syntax

```
#include <string.h>
      char *strcat (char *string1, const char *string2);
```

Parameters

`string1`
: Destination string; must contain a null character marking the end of the string.

`string2`
: Source string appended to `string1`; must contain a null character marking the end of the string.

Additional Information

Terminates the resulting string with a null character `\0`. No overflow checking is performed when strings are appended.

See also: strncat( ), strncmp( ), strncpy( ), strnicmp( ), strrchr( ), strspn( )

Returns

A pointer to the concatenated string.

strchr

Searches for a character in a null-terminated string.

Syntax

```
#include <string.h>
      char *strchr (const char *string, int c);
```

Parameters

`string`	String to search; must contain a null character `\0` marking the end of the string; the terminating null character is included in the search.
`c`	Character to be located.

See also: strcspn( ), strncat( ), strncmp( ), strncpy( ), strnicmp( ), strpbrk( ), strrchr( ), strspn( ), strstr( )

Returns

A pointer to the first occurrence of `c` in the string. The character may be the null character `\0`.

A null pointer if the character is not found.

strcmp, strcmpi, stricmp

Compare two null-terminated strings lexicographically.

Syntax

```
#include <string.h>
      int strcmp (const char *string1, const char *string2);
      int strcmpi (const char *string1, const char *string2);
      int stricmp (const char *string1, const char *string2);
```

Parameters

`string1, string2`

Strings to compare; must contain null characters `\0` marking the end of the strings.

Additional Information

The **strcmpi( )** and **stricmp( )** functions are case-insensitive versions of **strcmp( )**. They work identically in all other respects.

See also: memcmp( ), memicmp( ), strncat( ), strncmp( ), strncpy( ), strnicmp( ), strrchr( ), strspn( )

Returns

A value indicating the relationship:

Value	**Meaning**
< 0	string1 less than string2
= 0	string1 identical to string2
> 0	string1 greater than string2

strcoll

Compares null-terminated strings using locale-specific collating sequences.

Syntax

```
#include <string.h>
      int strcoll (const char *string1, const char *string2);
```

Parameters

`string1, string2`

Strings to compare; must contain null characters `\0` marking the end of the strings.

See also: localeconv( ), setlocale( ), strcmp( ), strncmp( ), strxfrm( )

Returns

A value indicating the relationship:

Value	**Meaning**
< 0	string1 less than string2
= 0	string1 identical to string2
> 0	string1 greater than string2

strcpy

Copies a null-terminated string.

Syntax

```
#include <string.h>
      char *strcpy (char *string1, const char *string2);
```

Parameters

`string1`
: Destination string; must contain a null character `\0` marking the end of the string.

`string2`
: Source string, including the terminating null character.

Additional Information

No overflow checking is performed when strings are copied.

See also: strcat( ), strcmp( ), strncat( ), strncmp( ), strncpy( ), strnicmp( ), strrchr( ), strspn( )

Returns

Returns `string1`.

strcspn

Finds a null-terminated substring in a string.

Syntax

```
#include <string.h>
      size_t strcspn (const char *string1, const char
      *string2);
```

Parameters

`string1`
: Source string; must contain a null character `\0` marking the end of the string.

`string2`
: Character set to search for; must contain a null character `\0` marking the end of the string.

Additional Information

Terminating null characters are not considered in the search.

See also: strncat( ), strncmp( ), strncpy( ), strnicmp( ), strrchr( ), strspn( )

Returns

The index of the first character in `string1` belonging to the set of characters specified by `string2`. This value is equivalent to the length of the initial substring of `string1` consisting entirely of characters not in `string2`.

0 if `string1` begins with a character from `string2`.

strdup

Duplicates null-terminated strings.

Syntax

```
#include <string.h>
      char *strdup (const char *string);
```

Parameter

`string`	Source string; must contain a null character `\0` marking the end of the string.

Additional Information

The function allocates storage space from the heap for a copy of string, using **malloc( )**.

See also: strcat( ), strcmp( ), strncat( ), strncmp( ), strncpy( ), strnicmp( ), strrchr( ), strspn( )

Returns

A pointer to the storage space containing the copied string.

A null pointer if storage cannot be allocated.

strerror

Gets a system error message.

Syntax

```
#include <string.h>
      char *strerror (int errnum);
      char *_strerror (const char *string);
```

Parameter

`errnum`	Error number to map to an error-message string.

Additional Information

The function itself does not actually print the message. To send or print the message, use an output function such as **perror( )**.

See also: clearerr( ), ferror( ), perror( )

Returns

A pointer to the error-message string.

strftime

Formats a time string.

Syntax

```
#include <time.h>
      size_t strftime (char *string, size_t maxsize, const char
                       *format, const struct tm *timedate);
```

Parameters

`string` Output string.

`maxsize`
Maximum length of string.

`format` Format control string; normal characters and format specifications.

`timedate`
Time/date structure, `tm`.

Additional Information

Format specifications have a preceding percent sign (`%`); preceding characters are copied unchanged to `string`. The LC_TIME category of the current locale affects the output formatting of **strftime( )**.

The format specifications are:

Format	**Description**
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for the locale
%d	Day of the month as a decimal number (01 - 31)
%H	Hour in 24-hour format (00 - 23)
%I	Hour in 12-hour format (01 - 12)
%j	Day of the year as a decimal number (001 - 366)
%m	Month as a decimal number (01 - 12)
%M	Minute as a decimal number (00 - 59)
%p	Current locale's AM/PM indicator for a 12-hour clock
%S	Second as a decimal number (00 - 61)

Format	**Description**
%U	Week of year as decimal number; Sunday is first day of week (00 - 53)
%w	Weekday as a decimal number (0 - 6; Sunday is 0)
%W	Week of year as decimal number; Monday is first day of week (00 - 53)
%x	Date representation for current locale
%X	Time representation for current locale
%y	Year without the century as a decimal number (00 - 99)
%Y	Year with the century as a decimal number
%z	Timezone name or abbreviation; no characters if timezone is unknown
%%	Percent sign

See also: asctime( ), localeconv( ), setlocale( ), strxfrm( )

Returns

The number of characters placed in `string` if the total number of resulting characters, including the terminating null, is not more than `maxsize`.

0 and the contents of the string are indeterminate if the result is larger than `maxsize`.

strlen

Gets the length of a null-terminated string.

Syntax

```
#include <string.h
      size_t strlen (const char *string);
```

Parameter

`string` String to find length of.

Returns

The string length in bytes of `string`, not including the terminating null character `\0`.

No error return.

strlwr

Converts uppercase letters in a null-terminated string to lowercase. Other characters are not affected.

Syntax

```
#include <string.h>
      char *strlwr (char *string);
```

Parameter

`string` String to convert.

See also: **strupr( )**

Returns

A pointer to the converted string.

No error return.

strncat

Appends characters to a string.

Syntax

```
#include <string.h>
      char *strncat (char *string1, const char *string2 size_t
                     count);
```

Parameters

`string1`
: Destination string.

`string2`
: Source string.

`count` Number of characters to be appended.

Additional Information

Appends at most the first `count` characters of `string2` to `string1` and terminates the resulting string with a null character. If `count` is greater than the length of `string2`, the length of `string2` is used in place of `count`.

See also: strcat( ), strcmp( ), strcpy( ), strncmp( ), strncpy( ), strnicmp( ), strrchr( ), strset( ), strspn( )

Returns

A pointer to the concatenated string.

strncmp

Compares substrings.

Syntax

```
#include <string.h>
      int strncmp (const char *string1, const char *string2,
                   size_t count);
```

Parameters

`string1, string2`
Strings to compare.

`count` Number of characters compared.

Additional Information

Lexicographically compares the first `count` characters of `string1` and `string2`.

The **strnicmp( )** function is a case-insensitive version of **strncmp**.

See also: strcat( ), strcmp( ), strcpy( ), strncat( ), strncpy( ), strrchr( ), strset( ), strspn( )

Returns

A value indicating the relationship between the substrings:

Value	**Meaning**
< 0	string1 less than string2
= 0	string1 identical to string2
> 0	string1 greater than string2

strncpy

Copies the specified number of characters from one string to another.

Syntax

```
#include <string.h>
      char *strncpy (char *string1, const char *string2,
                     size_t count);
```

Parameters

`string1`
Destination string.

`string2`
Source string.

`count` Number of characters copied.

Additional Information

Copies `count` characters of `string2` to `string1`.

If `count` is less than the length of `string2`, a null character `\0` is not appended automatically to the copied string. If `count` is greater than the length of `string2`, the `string1` result is padded with null characters up to length `count`.

The behavior of **strncpy( )** is undefined if the address ranges of the source and destination strings overlap.

See also: strcat( ), strcmp( ), strcpy( ), strncat( ), strncmp( ), strnicmp( ), strrchr( ), strset( ), strspn( )

Returns

Returns `string1`.

strnicmp

Compares substrings without regard to case.

Syntax

```
#include <string.h>
      int strnicmp (const char *string1, const char *string2,
                    size_t count);
```

Parameters

`string1, string2`
Strings to compare.

`count` Number of characters compared.

Additional Information

Lexicographically compares the first count characters of `string1` and `string2`.

The **strnicmp( )** function is a case-insensitive version of **strncmp( )**.

See also: strcat( ), strcmp( ), strcpy( ), strncat( ), strncpy( ), strrchr( ), strset( ), strspn( )

Returns

A value indicating the relationship:

Value	**Meaning**
< 0	string1 less than string2
= 0	string1 identical to string2
> 0	string1 greater than string2

strnset

Sets the specified number of characters in a string to a character.

Syntax

```
#include <string.h>
      char *strnset (char *string, int c, size_t count);
```

Parameters

`string`	String to be set.
`c`	Character to set the string to.
`count`	Maximum number of characters to set.

Additional Information

If `count` is greater than the length of `string`, the length of `string` is used in place of `count`.

See also: strcat( ), strcmp( ), strcpy( ), strset( )

Returns

A pointer to the altered string.

strpbrk

Searches a string for the first occurrence of any character in the specified character set.

Syntax

```
#include <string.h>
      char *strpbrk (const char *string1, const char *string2);
```

Parameters

`string1`
String to search.

`string2`
Character set to search for.

Additional Information

The terminating null character `\0` is not included in the search.

See also: strchr( ), strrchr( )

Returns

A pointer to the found character.

A null pointer if `string1` and `string2` have no characters in common.

strrchr

Searches a string for the last occurrence of a character.

Syntax

```
#include <string.h>
      char *strrchr (const char *string, int c);
```

Parameters

`string`	String to search.
`c`	Character to find.

Additional Information

The string's terminating null character `\0` is included in the search.

Use **strchr( )** to find the first occurrence of `c` in `string`.

See also: strchr( ), strcspn( ), strncat( ), strncmp( ), strncpy( ), strnicmp( ), strpbrk( ), strspn( )

Returns

A pointer to the last occurrence of the character in the string.

A null pointer if the character is not found.

strrev

Reverses the order of the characters in a string.

Syntax

```
#include <string.h>
      char *strrev (char *string);
```

Parameter

`string`	String to be reversed.

Additional Information

The terminating null character `\0` remains in place.

See also: **strcpy**, **strset**

Returns

A pointer to the altered string.

No error return.

strset

Sets all characters in a string to a specified character.

Syntax

```
#include <string.h>
      char *strset (char *string, int c);
```

Parameters

`string`	String to be set.
`c`	Character to set the string to.

Additional Information

Does not set the terminating null character `\0` to `c`.

See also: memset( ), strcat( ), strcmp( ), strcpy( ), strnset( )

Returns

A pointer to the altered string.

No error return.

strspn

Finds the first character in a string that does not belong to a set of characters in a substring.

Syntax

```
#include <string.h>
      size_t strspn (const char *string1, const char *string2);
```

Parameters

`string1`
String to search.

`string2`
Character set.

Additional Information

The null character `\0` terminating `string2` is not considered in the matching process.

See also: strcspn( ), strncat( ), strncmp( ), strncpy( ), strnicmp( ), strrchr( )

Returns

An integer value specifying the length of the segment in `string1` consisting entirely of characters in `string2`.

0 if `string1` begins with a character not in `string2`.

strstr

Finds a substring within a string.

Syntax

```
#include <string.h>
      char *strstr (const char *string1, const char *string2);
```

Parameters

`string1`
String to search.

`string2`
String to search for.

See also: strcspn( ), strncat( ), strncmp( ), strncpy( ), strnicmp( ), strpbrk( ), strrchr( ), strspn( )

Returns

A pointer to the first occurrence of `string2` in `string1`.

A null pointer if the string is not found.

strtod, strtol, strtoul

Strtod converts a string to double; **strtol** converts to long; **strtoul** converts to unsigned long.

Syntax

```
#include <stdlib.h>
      double strtod (const char *nptr, char **endptr);
      long strtol (const char *nptr, char **endptr, int base);
      unsigned long strtoul (const char *nptr, char **endptr,
                             int base);
```

Parameters

`nptr`	String to convert; a sequence of characters that can be interpreted as a numerical value of the specified type.
`endptr`	End of scan.
`base`	Number base to use.

Additional Information

The **strtod( )** function expects `nptr` to point to a string with this form:

```
[whitespace] [sign] [digits] [.digits] [ d | D | e | E [sign] digits]
```

The first character that does not fit this form stops the scan.

The **strtol( )** function expects `nptr` to point to a string with this form:

```
   [whitespace] [sign] [0] [ x | X ] [digits]
```

The **strtoul( )** function expects `nptr` to point to a string with this form:

```
   [whitespace] [ + | - ] [ 0 ] [ x | X ] [digits]
```

These functions stop reading the string at the first character they cannot recognize as part of a number. This may be the null character `\0` at the end of the string. With **strtol( )** or **strtoul( )**, this terminating character can also be the first numeric character greater than or equal to `base`. If `endptr` is not a null pointer, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`.

If no conversion can be performed (no valid digits are found or an invalid base is specified), the value of `nptr` is stored at the location pointed to by `endptr`.

Base	**Meaning**

Between 2 and 36	Base used as the base of the number.
0	The initial characters of the string pointed to by nptr determine the base.
1st char = 0 and 2nd char not = x or X	The string is interpreted as an octal integer; otherwise, it is interpreted as a decimal number.
1st char = 0 and 2nd char = x or X	The string is interpreted as a hexadecimal integer.
1st char = 1 through 9	The string is interpreted as a decimal integer.
a through z or A through Z	Are assigned the values 10 through 35; only letters whose assigned values are less than base are permitted.

The **strtoul( )** function allows a plus (+) or minus (-) sign prefix; a leading minus sign indicates that the return value is negated.

See also: atof( ), atol( )

Returns

Strtod( ) Returns the converted value.

Returns ±HUGE_VAL when the representation would cause an overflow.

Returns 0 if no conversion could be performed or an underflow occurred.

Strtol( ) Returns the converted value.

Returns LONG_MAX or LONG_MIN when the representation would cause an overflow.

Returns 0 if no conversion could be performed.

Strtoul( ) Returns the converted value, if any.

Returns 0 if no conversion can be performed.

Returns ULONG_MAX on overflow.

strtok

Finds the next token in a string.

Syntax

```
#include <string.h>
      char *strtok (char *string1, const char *string2);
```

Parameters

`string1`
String containing token(s); may be separated by one or more of the delimiters from `string2`.

`string2`
Set of delimiter characters.

Additional Information

This function reads `string1` as a series of zero or more tokens and `string2` as the set of characters serving as delimiters of the tokens in `string1`.

Use a series of calls to **strtok( )** to break out tokens from `string1`. In the first call, **strtok( )** searches for the first token in `string1,` skipping leading delimiters. To read the next token from `string1`, call **strtok( )** with a null pointer value for the `string1` argument. The null pointer argument causes **strtok( )** to search for the next token in the previous token string. The set of delimiters may vary from call to call, so `string2` can take any value.

Calls to this function will modify `string1`, since each time **strtok( )** is called it inserts a null character `\0` after the token in `string1`.

See also: _get_rmx_conn( ), strcspn( ), strspn( )

➠ **Note**
C string tokens are `char` values separated by delimiter characters; an iRMX connection token is a `selector` value obtained from a call to **_get_rmx_conn( )** or iRMX system calls. Do not confuse the C concept of a character string token with the iRMX connection token.

Returns

A pointer to the first token in `string1` the first time **strtok( )** is called. All tokens are null-terminated.

A pointer to the next token in the string on subsequent calls with the same token string.

A null pointer means there are no more tokens.

strupr

Converts any lowercase letters in a null-terminated string to uppercase.

Syntax

```
#include <string.h>
      char *strupr (char *string);
```

Parameter

`string`	String to be capitalized.

Additional Information

Does not affect characters other than lowercase.

See also: **strlwr( )**

Returns

A pointer to the converted string.

No error return.

strxfrm

Transforms a string based on locale-specific information and stores the result.

Syntax

```
#include <string.h>
      size_t strxfrm (char *string1, const char *string2,
                      size_t count);
```

Parameters

`string1`
: String to which transformed version of `string2` is returned.

`string2`
: String to transform.

`count` Maximum number of characters to be placed in `string1`.

Additional Information

The transformation is made using the information in the locale-specific LC_COLLATE macro.

The value of this expression is the size of the array needed to hold the transformation of the source string:

```
1 + strxfrm (NULL, string, 0);
```

The C libraries support the C locale only; thus **strxfrm( )** is equivalent to these commands:

```
strncpy (_string1, _string2, _count);
return (strlen (_string2) );
```

After the transformation, a call to **strcmp( )** with the two transformed strings will yield identical results to a call to **strcoll( )** applied to the original two strings.

See also: localeconv( ), setlocale( ), strncmp( )

Returns

The length of the transformed string, not counting the terminating null character.

If the return value is greater than or equal to `count`, the contents of `string1` are unpredictable.

swab

Copies while swapping bytes.

Syntax

```
#include <stdlib.h>
      void swab (const char *src, char *dest, int n);
```

Parameters

`src`	Points to the source buffer.
`dest`	Points to a buffer to which the source buffer is copied, with each pair of bytes swapped.
`n`	The number of bytes to be copied.

Additional Information

Use **swab** to copy `n` bytes from the `src` buffer while swapping each pair of adjacent bytes.

If `n` is odd, the last byte is copied directly from the `src` buffer to the `dest` buffer, with no byte swapping.

Returns

Nothing.

system

Invokes the system call **rq_c_send_command** to execute an iRMX command line.

Syntax

```
#include <stdlib.h>
      int system (const char *command);
```

Parameter

`command`

Command to be executed; it can be any valid HI command, user program, or alias.

Additional Information

The **system( )** function may be invoked multiple times with an ampersand (&) in the last character of `command`, to extend the command line. The connection is maintained until **system( )** is invoked without an &.

See also: **rq_c_send_command**, *System Call Reference*

Returns

Value	**Meaning**	
0	Successful; `command` is not NULL and the command interpreter is successfully started.	
0	And sets errno to ENOENT, if the command interpreter is not found.	
Not 0	If `command` is NULL and the command interpreter is found.	
-1	Error occurred, and the function sets **errno** to one of these values:	
	E2BIG	Command line exceeds 128 bytes.
	ENOMEM	One of these:
		Not enough memory is available to execute the command, or
		The available memory has been corrupted, or
		An invalid block exists, indicating that the process making the call was not allocated properly.

tan, tanh

Tan( ) calculates the tangent and **tanh( )** calculates the hyperbolic tangent of the number.

Syntax

```
#include <math.h>
      double tan (double x);
      double tanh (double x);
```

Parameter

`x`	Angle to calculate in radians.

See also: acos( ), asin( ), atan( ), cos( ), sin( )

Returns

Tan( ) Returns the tangent of `x`.

Returns a PLOSS error and sets **errno** to ERANGE if `x` is large and a partial loss of significance in the result may occur.

Returns 0, prints a TLOSS error message to *stderr*, and sets **errno** to ERANGE if `x` is so large that significance is totally lost.

Tanh( ) Returns the hyperbolic tangent of `x`.

No error return for **tanh( )**.

These functions do not return standard ANSI domain or range errors.

time

Gets the system time.

Syntax

```
#include <time.h>
      time_t time (time_t *timer);
```

Parameter

`timer`	Storage location for the return value. This parameter may be a null pointer, in which case the return value is not stored.

Additional Information

This function calls the system call **rq_get_time** and adds an adjustment factor: the number of seconds between epoch time and January 1, 1978, plus the local timezone factor `TZ`, described in **tzset( )**. This adjusts the iRMX OS time value to a POSIX-standard value.

See also: asctime( ), ctime( ), gmtime( ), localtime( ), tzset( )

Returns

The number of seconds elapsed since epoch time, according to the system clock.

No error return.

time macros, _tzset_ptr

Accesses daylight, timezone, and tzname environment variables.

Syntax

```
#include <time.h>
      #include <reent.h>
      struct _tzset {
            char *_tzname[2];
            long _timezone;
            int _daylight;
      }
      struct _tzset  *_tzset_ptr (void);
      #define daylight (_tzset_ptr( )->_daylight)
      #define timezone (_tzset_ptr( )->_timezone)
      #define tzname (_tzset_ptr( )->_tzname);
```

Additional Information

The **daylight( )** macro accesses the `_daylight` flag.

Value	**Meaning**
1	Daylight-savings-time is in effect (default).
0	Daylight-savings-time is not in effect.

The **timezone( )** macro accesses the value that represents the difference in seconds between GMT and local time.

The **tzname( )** macro accesses a pair of pointers to the timezone name and daylight-savings-time name. For example, `tzname[0]` could point to EST and `tzname[1]` could point to EDT. The default strings are PST and PDT.

The **_tzset_ptr** function uses the `_tzset` structure that contains members corresponding to **tzname**, **timezone**, and **daylight**. Each of these macros calls **_tzset_ptr**.

See also: **tzset( )**, <*time.h*>

Returns

Pointer to `_tzset`.

Null pointer if unsuccessful.

tmpfile

Creates a temporary file, opens in it binary read/write mode, and returns a stream pointer to it.

Syntax

```
#include <stdio.h>
      FILE *tmpfile (void);
```

Additional Information

The temporary file is automatically deleted when the file is closed, when the program terminates normally, or when **rmtmp( )** is called, assuming that the current working directory does not change.

See also: rmtmp( ), open( ), tmpnam( )

Returns

A stream pointer.

A null pointer if unsuccessful.

tmpnam

Creates a temporary filename, which can open a temporary file without overwriting an existing file.

Syntax

```
#include <stdio.h>
      char *tmpnam (char *string);
```

Parameter

`string`	Pointer to the temporary filename.

Additional Information

If `string` is a null pointer, **tmpnam( )** leaves the result in an internal static buffer. Thus any subsequent calls destroy this value.

If `string` is not a null pointer, it is assumed to reference a string buffer of at least `L_tmpnam` bytes. The function will generate unique filenames for up to TMP_MAX calls.

The character string that **tmpnam( )** creates consists of the path prefix, defined by `P_tmpdir`, followed by a sequence consisting of the digit characters 0 through 9; the numerical value of this string can range from 1 to 65,535.

Changing the definitions of `L_tmpnam` or `P_tmpdir` in <*stdio.h*> does not change the operation of **tmpnam( )**.

See also: mktmp( ), tmpfile( )

Returns

A pointer to the temporary filename generated.

A null pointer if it is impossible to create the name or the name is not unique.

toascii, tolower, _tolower, toupper, _toupper

Convert single characters.

Syntax

```
#include <ctype.h>
      #include <stdlib.h>
      int toascii (int c);
      int tolower (int c);
      int _tolower (int c);
      int toupper (int c);
      int _toupper (int c);
```

Parameter

c	Character to convert.

Additional Information

These functions are implemented both as functions and as macros. To use the function versions, remove the macro definitions through `#undef` directives, or do not include *<ctype.h>*.

Function	Description
toascii( )	Converts c to ASCII character. The **toascii( )** function sets all but the low-order 7 bits of c to 0, so that the converted value represents an ASCII character. If c already represents an ASCII character, c is unchanged.
tolower( )	Converts c to lowercase if c represents an uppercase letter.
_tolower( )	Converts c to lowercase only when c represents an uppercase letter; the result is undefined if c is not.
toupper( )	Converts c to uppercase if c represents a lowercase letter.
_toupper( )	Converts c to uppercase only when c represents a lowercase letter; the result is undefined if c is not.

See also: **is** functions

Return Value

The converted character.

No error return.

tzset

Sets the time environment variables.

Syntax

```
#include <time.h>
      void tzset (void);
      int daylight /* Global variables set by function */
      long timezone;
      char *tzname[2]
```

Additional Information

This function calls **getenv( )** to obtain the current setting of the environment variable `TZ`, then assigns values to three global variables: `daylight`, `timezone`, and `tzname`. The **localtime( )** function uses these variables to make corrections from GMT to local time, and **time( )** uses these variables to compute GMT from system time.

The `TZ` environment variable has the following syntax:

```
[:]<std><std_offset>[<dst>[<dst_offset>][,<sdate>[/<stime>]
,<edate>[/<etime>]]]
```

Where:

`[:]`, indicates how the system clock is set. If a semi-colon is present, the time is set to Local Time. No semi-colon indicates that the POSIX-compliant setting of Universal Constant Time (UCT) is used.

Where:

Local Time means that functions will not need to do shifts for timezone, but will not shift for daylight savings time. The user must reset the system clock twice a year by hand to account for these. All iRMX file timestamps and CUSPs report the local time.

UCT means that functions will automatically handle timezone shifts and daylight savings time switches. All iRMX file timestamps are in UCT. The iRMX date/time CUSPs report in UCT even though the system says Local Time.

`<std>` (Standard Time) and `<dst>` (Daylight Savings Time) are `_POSIX_TZNAME_MAX` in length and are typically a three character string of the form *x*`ST` or *x*`DT`, such as `PST`.

`<std_offset>,` `<dset_offset>`, `<stime>`, and `<etime>` have the format:

```
[+|-]<hours>[:<minutes>[:<seconds>]]
```

The default is `2:00:00`.

`<sdate>` (DST start date) and `<edate>` (DST end date) have the format:

```
<julian0>|J<julian1>|M<month>.<week>.<day>
```

Where:

`<julian0>` is 0 to 365
`<julian1>` is 1 to 366
`<month>` is 1 to 12
`<week>` is 1 to 5 where 5 is the last week of the month
`<day>` is 0 (Sunday) to 6 (Saturday)

The default is implementation-specific (U.S. law since 1987 states "`M4.1.0`" and `M10.5.0`").

These values are assigned to the variables `daylight`, `timezone`, and `tzname` when **tzset( )** is called:

Variable	**Value and Meaning**
`daylight`	Indicates whether daylight savings time is observed locally (1) or not (0). To check the state of this variable, call the **localtime()** function and see if the `tm_isdst` field is 1 or 0.
`timezone`	Seconds west of UCT if positive or seconds east of UCT if negative.
`tzname[0]`	String value of the timezone name from the TZ setting; default is PST
`tzname[1]`	String value of the daylight savings time name; default is PDT. An empty string must appear if daylight savings time is never in effect, as in certain states and localities.

See also: asctime( ), getenv( ), gmtime( ), localtime( ), putenv( ), time( ), time macros

Returns

Nothing.

ultoa, utoa

Ultoa converts unsigned long and **utoa** converts an integer to a null-terminated string and stores it, without overflow checking.

Syntax

```
#include <stdlib.h>
      char *ultoa (unsigned long value, char *string, int
      radix);
      char *utoa (unsigned int value, char *string, int radix);
```

Parameters

`value`	Number to convert.
`string`	String result.
`radix`	Base of `value`; must be in the range 2-36.

Additional Information

The `string` buffer must be large enough to accommodate the largest representation of a long integer that `radix` calls for. For example, on an iRMX system, the largest signed values represented in a 32-bit integer are -2,147,483,648 and +2,147,483,647. In base 2, their binary representations are 1 and thirty-one trailing 0s, and 0 and thirty-one trailing 1s, respectively. With the sign and terminating null character, the minimum buffer size would be thirty-four bytes for binary representation.

For portability, use **sprintf**'s `%lo`, `%ld`, or `%lx` conversion specifiers, if `radix` is 8, 10, or 16, when calling **ultoa( )**. Use **sprintf**'s `%o`, `%d`, or `%x` conversion specifiers, if `radix` is 8, 10, or 16, when calling **utoa( )**.

With `radix` greater than 10, digits in the converted string representing values 10 through 35 are the characters `a` through `z`.

See also: itoa( ), ltoa( ), sprintf( )

Returns

A pointer to the string.

No error return.

umask

Sets the default file-permission mask of the current process to the specified mode.

Syntax

```
#include <io.h>
      #include <sys/stat.h>
      #include <sys/types.h>
      mode_t umask (mode_t pmode);
```

Parameter

`pmode` Default permission mode.

Additional Information

The file-permission mask is applied to the permission mode specified in calls to **creat( )**, **open( )**, or **sopen( )**. The permission mode determines the file's ownership and access rights; the file-permission mask affects only access rights. If a bit in the mask is 1, the corresponding bit in the file's requested permission mode value is set to 0 (disallowed). If a bit in the mask is 0, the corresponding bit is left unchanged. The permission mode for a new file is not set until the file is closed for the first time.

The argument `pmode` is a constant expression containing one or more of the manifest constants defined in <*sys/stat.h*>. Join more than one constant with the bitwise-OR operator (`|`).

Value	**Meaning**
S_IRGRP	Read permission bit for POSIX file group
S_IROTH	Read permission bit for POSIX World (other) owner
S_IRUSR	Read permission for POSIX file owner
S_IWGRP	Write permission bit for POSIX file group
S_IWOTH	Write permission bit for POSIX World owner
S_IWUSR	Write permission for POSIX file owner
S_IXGRP	Execute or search permission bit for POSIX file group
S_IXOTH	Execute or search permission bit for POSIX World owner
S_IXUSR	Execute or search permission for POSIX file owner

See also: chmod( ), creat( ), mkdir( ), open( ), <*sys/stat.h*>

Returns

The previous value of `pmode`.

No error return.

ungetch

Pushes a character back to the console, causing that character to be the next character read.

Syntax

```
#include <conio.h>
      int ungetch (int c);
```

Parameter

c	Character to be pushed; must not be EOF

Additional Information

Read the next character using **getch( )** or **getche( )**. This function fails if it is called more than once before the next read.

See also: cscanf( ), getch( ), getche( )

Returns

Value	**Meaning**
c	Successful
EOF	Error

unlink

Deletes a file.

Syntax

```
#include <io.h>    /* OR */
      #include <stdio.h>
      int unlink (const char *filename);
```

Parameter

`filename`

Name of file to delete.

See also: close( ), remove( )

Returns

Value	Meaning	
0	Successful	
-1	Error. The function sets **errno** to one of these values:	
	EACCES	Pathname specifies a read-only file.
	ENOENT	File or pathname not found, or pathname specifies a directory.

utime

Sets the modification time for a file.

Syntax

```
#include <sys\types.h>
      #include <sys/utime.h>
      int utime (const char *filename, struct utimbuf *times);
```

Parameters

`filename`
File on which to set modification time. The process must have write access to the file.

`times` Pointer to stored time values. If `times` is a NULL pointer, the modification time is set to the current time. Otherwise, `times` must point to a `utimbuf` structure, defined in *sys\utime.h*.

Additional Information

The modification time is set from the modtime field in the `utimbuf` structure. Although this structure contains a field for access time, only the modification time is set.

See also: asctime( ), ctime( ), fstat( ), ftime( ), gmtime( ), localtime( ), stat( ), time( )

Returns

Value	**Meaning**	
0	The file-modification time was changed	
-1	Time was unchanged and the function sets **errno** to one of these values:	
	EACCES	Pathname specifies directory or read-only file.
	EINVAL	Invalid argument; the times argument is invalid.
	EMFILE	Too many open files (the file must be opened to change its modification time).
	ENOENT	Filename or pathname not found.

va_arg, va_end, va_start

Access variable-argument lists.

Syntax

```
#include <stdarg.h>
      #include <stdio.h>
      type va_arg (va_list arg_ptr, type);
      void va_end (va_list arg_ptr);
      void va_start (va_list arg_ptr, prev_param);
```

Parameters

`arg_ptr`
Pointer to variable-argument list.

`prev_param`
Parameter preceding first optional argument.

`type` Type of argument to be retrieved.

Additional Information

These macros provide a portable way to access a function's arguments when the function takes a variable number of arguments. Use the **va_start( )** macro before using **va_arg( )** for the first time. The macros behave as follows:

Macro	**Description**
va_arg( )	Retrieves type parameter from the location given by `arg_ptr`. Increments `arg_ptr` to point to the next argument in the list, using the size of type parameter to determine where the next argument starts. Use this macro multiple times to retrieve all arguments from the list.
va_end( )	After all arguments have been retrieved, resets `arg_ptr` to a null pointer.
va_start( )	Sets `arg_ptr` to the first optional argument in the variable-argument list. The `arg_ptr` argument must be of the `va_list` type. The argument `prev_param` is the name of the required parameter immediately preceding the first optional argument in the argument list. If `prev_param` is declared with the register storage class, the macro's behavior is undefined.

The macros assume that the function takes a fixed number of required arguments, followed by a variable-argument list.

See also: <*stdarg.h*>, vfprintf( ), vprintf( ), vsprintf( )

Returns

Va_arg( ) returns the current argument.

Va_start( ) and **va_end( )** do not return values.

vfprintf, vprintf, vsprintf

Vfprintf( ) formats and sends data to the file specified by `stream`, **vprintf( )** sends data to standard output, and **vsprintf( )** sends data to the memory pointed to by `buffer`.

Syntax

```
#include <stdio.h>
      #include <stdarg.h>
      int vfprintf (FILE *stream, const char *format,
                    va_list argptr);
      int vprintf (const char *format, va_list argptr);
      int vsprintf (char *buffer, const char *format,
                    va_list argptr);
```

Parameters

`stream`	Pointer to `FILE` structure.
`format`	Formatted string.
`argptr`	Pointer to list of arguments.
`buffer`	Storage location for output.

Additional Information

These functions are similar to their counterparts **fprintf( )**, **printf( )**, and **sprintf( )**, but each accepts a pointer to a variable-argument list instead of additional arguments.

The `format` argument has the same form and function as for the **printf( )** function.

The `argptr` parameter has type `va_list`. The `argptr` parameter points to a list of arguments that are converted and output according to the corresponding format specifications in the `format` argument.

See also: printf( ) for a description of `format`, fprintf( ), sprintf( ), va_arg( ), va_end( ), va_start( )

Returns

The number of characters written, not counting the terminating null character.

A negative value if an output error occurs.

vfscanf, vscanf, vsscanf (ANSI, stdio)

Reads and formats character data into the specified locations.

- **Vfscanf** is analogous to vfprintf and reads input from the current position of a stream using a variable argument list of pointers (see **stdarg**).
- **Vscanf** scans a variable argument list from the standard input (stdin) and **vsscanf** scans it from a string. These are analogous to the **vprintf** and **vsprintf** functions, respectively.

Syntax

```
#include <stdio.h>
   #include <stdarg.h>
   int vfscanf (FILE *stream, const char *format, va_list argptr);
   int vscanf (const char *format, va_list argptr);
   int vsscanf (const char *buffer, const char *format, va_list
   argptr);
```

Parameters

`stream`	Pointer to FILE structure.
`format`	Formatted string. This parameter has the same form and function as in printf.
`argptr`	Pointer to list of arguments. This parameter has type va_list and points to a list of arguments that are converted and output according to the corresponding format specifications in format.
`buffer`	Storage location for input.

Additional Information

These functions are similar to their counterparts fscanf, scanf, and sscanf, but each accepts a pointer to a variable-argument list instead of additional arguments.

Returns

Success	The number of fields successfully converted and assigned, which may be less than the number requested. Does not include fields read but not assigned.
Failure	A negative value if an output error occurs. EOF if end-of-file is encountered on the first attempt to read a character.

See also: va_arg, va_end, va_start

wcstombs

Converts a sequence of wide characters to a corresponding sequence of multibyte characters.

Syntax

```
#include <stdlib.h>
      size_t wcstombs (char *mbstr, const wchar_t *wcstr,
                       size_t count);
```

Parameters

`mbstr`	The address of a sequence of multibyte characters which have been converted.
`wcstr`	The address of a sequence of wide characters to convert.
`count`	The number of bytes to convert.

Additional Information

If **wcstombs( )** encounters the wide-character null, either before or when `count` occurs, it converts it to the multibyte null character (a 16-bit 0) and stops. Thus, the multibyte character string at `mbstr` is null-terminated only if **wcstombs( )** encounters a wide-character null character during conversion. If the sequences pointed to by `wcstr` and `mbstr` overlap, the behavior of **wcstombs( )** is undefined.

See also: mblen( ), mbstowcs( ), mbtowc( ), wctomb( )

Returns

The number of converted multibyte characters, excluding the wide-character null character.

-1 cast to type `size_t` if a wide character cannot be converted to a multibyte character.

wctomb

Converts a wide character to the corresponding multibyte character and stores it in a specified location.

Syntax

```
#include <stdlib.h>
      int wctomb (char *mbchar, wchar_t wchar);
```

Parameters

`mbchar`	The address of a converted multibyte character.
`wchar`	A wide character to convert.

See also: mblen( ), mbstowcs( ), wcstombs( )

Returns

The number of bytes, never greater than MB_CUR_MAX, in the wide character.

0 if `wchar` is the wide-character null.

-1 if the conversion is not possible in the current locale.

write

Writes data from a buffer to a file.

Syntax

```
#include <io.h>
      int write (int handle, const char *buffer, unsigned int
      count);
```

Parameters

`handle`	Descriptor referring to an open file.
`buffer`	Data to be written.
`count`	Number of bytes.

Additional Information

Writing begins at the current file pointer position. If the file is open for appending, the operation begins at the end-of-file. After writing, the file pointer increases by the number of bytes actually written.

When writing more than 2 gigabytes to a file, the return value must be of type unsigned integer. However, the maximum number of bytes that can be written to a file at one time is 4 gigabytes -2, since 4 gigabytes -1 (or 0xFFFFFFF) is indistinguishable from -1 and would return an error.

When **write( )** is received, the file descriptor is checked for text or binary mode.

If the file was opened in text mode, the output buffer is written up to each <LF> character, then a <CR><LF> pair is written separately. If multiple tasks are writing to the same output, scrambling will occur in text mode; use binary mode. When writing to files opened in text mode, the **write( )** function treats a <Ctrl-Z> character as the logical end-of-file. When writing to a device, **write( )** treats a <Ctrl-Z> in the buffer as an output terminator.

See also: fwrite( ), open( ), read( )

Returns

The number of bytes actually written, not including <CR><LF> pairs. May be less than `count`, as when disk space is filled before `count` bytes are written.

-1 on error, and the function sets **errno** to one of these values:

EBADF	Invalid file descriptor or file not opened for writing.
ENOSPC	No space left on device.

❑❑❑

Index