DCC Compiler System: Library Reference

Edited by Jeff Teunissen

The latest version of DCC may be found at
https://github.com/Deek/CoCoC/

Edition 0.5, December 2022
Not finished yet, input welcome!

This book wouldn't be possible without the amazing work of Dennis M. Ritchie, Ken Thompson, Brian W. Kernighan, Alfred Aho, Rob Pike, Stephen Bourne, and a host of others. The co-creations of Unix and C have had impossible-to-overstate impacts on the development of the modern world. It is upon their shoulders we all stand.

The DCC standard libraries (`clib.l` and `clibt.l`) are based on earlier libraries written and/or compiled by Carl Kreider, possibly with additional code by Simmule Turner (`getopt`); with additional code and bug fixes by Jeff Teunissen.

The information in this book is compiled from the source code to, and original documentation for, the DCC standard library and the OS-9 Level II C graphics library; with additional reference to the OS-9 Level II Technical Manual, the UNIX Programmer's Manual (7th ed., vol. 1), The Linux Documentation Project, and the Linux *man-pages* project.

The information contained in this book is believed to be accurate as of the date of publication; however, the CoCoC project disclaims any responsibility for any damages, including indirect or consequential, from the use of the DCC compiler system or from reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

This book was typeset in Adobe Minion Pro, Adobe Myriad Pro, and DejaVu Sans Mono using LyX and XeLaTeX.

OS-9® is a registered trademark of Microware LP.

UNIX® is a registered trademark of The Open Group.

Contents

1 The C Standard Library 1
abort — Cause abnormal program termination 3
abs — Get the absolute value of an integer 4
asctime, ctime — Format date and time into a string 5
assert — Program verification 6
atof, atoi, atol — Convert ASCII string to a number 7
bsearch — Binary search of a sorted array 8
exp, exp10, log, log10, pow — Exponential and logarithmic functions 9
fabs — Get absolute value of floating-point number 10
fclose — Close a stream 11
feof, ferror, clearerr — Check and reset stream status 12
fflush — Flush a stream buffer 13
fgetc, getc, getchar, ungetc — Character input from streams 14
fgets — Read a string from a stream 16
fopen, freopen — Open a stream 17
fputc, fputs, putc, putchar, puts — Character output to streams 19
fread, fwrite — Binary stream input/output 21
frexp — Split a floating-point number into its components 22
fseek, ftell, rewind — Reposition a stream 23
gets — Read a string from `stdin` (deprecated) 24
is* — Character classification functions 25
localtime 26
malloc, calloc, realloc, free — Allocate and free memory 27
memchr — Scan memory for a character 29
memcpy, memccpy — Copy a memory area 30
memset — Fill memory area with a constant byte 31
printf, fprintf, sprintf — Formatted output conversion 32
qsort — Sort an array 34
scanf, fscanf, sscanf — Input string interpretation 35
sin, cos, tan, asin, acos, atan — Trigonometric functions 38
sinh, cosh, tanh, asinh, acosh, atanh — Hyperbolic functions 40
setbuf — Fix file buffer 41
setjmp, longjmp — Jump to another function 42
sqrt — Square root function 43
strcat, strncat — Concatenate two strings 44
strchr, strrchr — Locate a character in string 45

strcmp, strncmp — Compare two strings 46
strcpy, strncpy — Copy string data 47
strlen — Get the length of a string 48
strpbrk — Search a string for any of a set of bytes 49
strspn, strcspn — Get length of a prefix set 50
strtok — Extract tokens from a string 51
system — Issue a shell command 53
time — Get the current clock time 54
tolower, toupper — Convert uppercase or lowercase 55

2 Standard Library Extensions **57**
fdopen — Open a stream on a file number 58
fileno — Get system path number from stream 59
findstr, findnstr — Substring searches 60
getw, putw — Input and output of words (ints) 61
ibrk — Request internal memory 62
inv, sqr — Get the inverse of or square a number 64
l3tol, ltol3 — convert between 3-byte and long integers 65
min, max, umin, umax — Compare integers 66
mktemp — Create unique temporary filename 67
sleep — Sleep for a specified number of seconds 68
stacksize, freemem — get stack reservation info 69
_strass — byte by byte copy 70
strclr — Clear a string 71
strpfx — Check string prefix 72
strucmp, strnucmp, strucpy, strucat — Uppercase string functions 73
strhcpy — Copy an OS-9 string 74

3 OS-9 System Calls **75**
Stub-implemented system calls 76
access — Check user's permissions for a file 77
brk, sbrk — change data area size 78
chain — load and execute a new program 80
chdir, chxdir — change directory 81
chmod — change access permissions of a file 82
chown — change the ownership of a file 83
close — close a file 84
crc — compute a cyclic redundancy count 85
creat — Create a file 86
Defdrive — get default system drive 87
dup — duplicate an open path number 88
exit, _exit — Task termination 89
getpid — get the task id 90

getstat — get file status 91
getuid — return user id 93
intercept — set function for interrupt processing 94
kill — Send a signal to a process 96
lseek — position in file 97
mkdir — Create a directory 98
modload — return a pointer to a module structure 99
munlink — unlink a module 100
open — Open a file 101
os9fork — create a process 102
pause — halt and wait for interrupt 103
prerr — print error message 104
read, readln — read from a file 105
setpr — set process priority 106
setime, getime — set and get system time 107
setuid — set user id 108
setstat — set file status 109
signal — catch or ignore interrupts 110
tsleep — put process to sleep 112
Unlink — remove directory entry 113
wait — wait for child process to end 114
Write, Writeln — write to a file or device 116
_os9 — system call interface from C programs 117

1
The C Standard Library

The standard library contains functions which fall into two classes: the standard I/O package (often abbreviated as "stdio", after its header file <stdio.h>) and convenience functions.

The standard I/O functions provide facilities that are normally considered part of the definitions of other languages; for example, the **FORMAT** statement in FORTRAN. In addition, automatic buffering of I/O channels improves file throughput because fewer system calls are necessary. The convenience functions include facilities for copying, comparing, and concatenating strings, converting numbers to strings, and doing the extra work in accessing system information such as the time.

In the following pages, the functions available are described in terms of what they do and the parameters they expect. The Synopsis section shows the name of each function and the type (if any) it returns. The declaration of arguments is shown as it would be written in the function definition, to indicate the types expected by the function. The header file in which a function is declared is shown in the Synopsis section by preceding the example function call with **#include** <filename>.

Header files Each function in the standard library is declared in at least one *header file*, which are files containing declarations and definitions meant to be pulled in by the C preprocessor using the **#include** directive. Most header files reside in the DEFS directory on the default system drive: if the file name is enclosed in angle bracket delimiters (<>), the preprocessor searches for it relative to this directory. For example, if /DD is the path name of the default system drive, then **#include** <stdio.h> is equivalent to **#include "/DD/DEFS/stdio.h"**.

Please note that because the C compiler assumes any undeclared function returns **int**, if you don't include the header file that declares a function and that function returns another type, you may have unexpected results. It's almost always better to include the header file, but you may choose to declare used functions in your program yourself before calling them. For example, if you wish to use atof() but don't want to **#include** <stdlib.h>, you should at least declare it by writing **double** atof();. Failure to do so may result in unexpected behavior at runtime.

Standard I/O The standard I/O functions implement an efficient user-level buffering scheme. The functions fgetc() and fputc() are the basic input and output functions. The

higher level routines that read or write data essentially call one of them in a loop. Actual file input/output for these buffered files is performed in batches, when the buffer is detected to be empty (for read mode) or full (for write mode). A file with its associated buffer is called a *stream*, and is declared to be a pointer to a defined type `FILE`. The function `fopen()` opens a file using an OS-9 system call, and associates that file with a stream.

The standard I/O functions should not be confused with the low-level system call functions, many of which have similar names. Nor should *file streams* be confused with *path numbers*, also called *file descriptors* or just *fd*s. Passing a stream to a system call, or a file descriptor to a standard I/O function, is a common mistake made by new or unwary users of the C standard library. When it occurs, the usual result is a program crash or hang. Be careful!

abort — Cause abnormal program termination

Synopsis

```
#include <stdlib.h>

abort()
```

Description

The `abort` function writes the program's memory to the file `core` in the current working directory. After this "core dump" is written, the program exits with a status of 1.

Return Value

This function never returns.

See Also

exit, _exit — Task termination

abs — Get the absolute value of an integer

Synopsis

```
#include <stdlib.h>

int abs(i)
int i;
```

Description

The abs function returns the absolute value (distance from zero on the number line) of its integer argument.

Caveats

Attempting to get the absolute value of the most negative integer returns the most negative integer. That is, `abs(-32768)` gives -32768.

See Also

`fabs()`

asctime, ctime — Format date and time into a string

assert — Program verification

Synopsis

```
#include <assert.h>

assert(expression);
```

Description

`assert` is a macro that indicates that *expression* is expected to be true at this point in the program. If the expression is false (zero), it causes an `abort()` after emitting a diagnostic comment on the standard error output.

If the macro `NDEBUG` is defined when `<assert.h>` was most recently included, `assert` does nothing and produces no code. It is recommended to define `NDEBUG` only when not looking for error conditions.

Diagnostics

```
F:N: Assertion 'expression' failed
```

In this example, `F` is the source file, and `N` the source line number of the call to `assert`.

atof, atoi, atol — Convert ASCII string to a number

Synopsis

```
#include <stdlib.h>

double atof (str)
char *str;

int atoi (str)
char *str;

long atol (str)
char *str;
```

Description

The initial part of the string `str` is converted to floating point, integer, or long integer. Conversion stops when an unrecognized character is encountered.

`atof` recognizes optional leading whitespace, then an optional sign, then a string of digits optionally containing a decimal point, then an optional `E` or `e`, then an optional sign, then a non-empty sequence of decimal digits indicating multiplication by a power of 10.

`atoi` and `atol` recognize optional leading whitespace, then an optional sign, then a string of digits.

Notes

Whitespace is as recognized by `isspace()`.

A sign is one of the characters `+` or `-`.

Bugs

Overflow may cause unpredictable results.

There is no error reporting.

See Also

`scanf()`

bsearch — Binary search of a sorted array

Synopsis

```
#include <stdlib.h>

char *bsearch (key, base, count, size, compar)
char *key, *base;
size_t count, size;
int (*compar) ();
```

Description

`bsearch` searches an array of `count` objects, the first member of which is pointed to by `base`, for a member that matches the object pointed to by `key`. The size of each member of the array is specified by `size`.

The contents of the array should already be in ascending order according to the comparison function `compar`. The `compar` routine is expected to take two pointer arguments, which point to `key` and to an array member, in that order; and should return an integer less than, equal to, or greater than zero if `key` is found to be less than, to match, or to be greater than the array member.

Return Value

On success, `bsearch` returns a pointer to a matching member of the array, or `NULL` if no match is found. If multiple elements match, it is unspecified which of them is returned.

Notes

For string variables, `strcmp()` may be a good choice for `compar`.

See Also

`strcmp()`, `qsort()`

exp[10], log[10], pow — Exponential and logarithmic functions

Synopsis

```
#include <math.h>

double exp (x)
double x;

double exp10 (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

double pow (x, y)
double x, y;
```

Link with `-t`.

Description

The `exp` function returns Euler's number e (the base of the natural logarithm, $e \approx 2.71828\ldots$) raised to the power `x`.

`exp10` returns the value 10 raised to the power `x`. For backward compatibility with earlier versions of the library, `exp10` is also available under the name `antilg`.

`pow` returns the value `x` raised to the power `y`.

`log` returns the natural logarithm of `x`.

`log10` returns the common (base 10) logarithm of `x`.

Notes

While (for example) `exp(x)` and `pow(M_E, x)` are equivalent and should result in the same value, the former is generally preferred as it is likely to execute more quickly.

fabs — Get absolute value of floating-point number

Synopsis

```
#include <math.h>

double fabs(x)
double x;
```

Description

The `fabs` function returns the absolute value of the floating-point number `x`. If the number is negative, the value returned is the same, except positive.

For backward compatibility, this function also has the name `dabs`.

See Also

`abs()`

fclose — Close a stream

Synopsis

```
#include <stdio.h>

int fclose (stream)
FILE *stream;
```

Description

The `fclose` function flushes the stream pointed to by `stream` (writing any buffered output it contains using `fflush()`), and closes the underlying path number.

The behavior of `fclose` is undefined if `stream` is not a valid pointer to an open stream.

`fclose` is automatically performed for each open file by `exit()` and normal program termination.

Return Value

On success, `0` is returned; otherwise, `EOF` is returned and `errno` is set. Regardless of error status, any further access to the stream (including another call to `fclose`) results in undefined behavior.

See Also

`fflush()`, system call close — close a file

feof, ferror, clearerr — Check and reset stream status

Synopsis

```
#include <stdio.h>

clearerr (stream)
FILE *stream;

int feof (stream)
FILE *stream;

int ferror (stream)
FILE *stream;
```

Description

`feof` returns a nonzero value if the file associated with `stream` has reached its end. zero is returned on error.

`ferror` returns a nonzero value if an error condition has occurred during any call using `stream`; zero is returned if no error has occurred. Stream error conditions persist, preventing further access by other standard library functions until the stream is closed or the error is cleared by `clearerr`.

`clearerr` resets the error condition on `stream`. This does **not** fix the error condition itself or prevent the error from occurring again – it merely allows standard library functions to attempt to use the stream again.

Caveats

These functions are implemented as macros defined in `<stdio.h>`, so their names may not be re-declared.

See Also

`fopen()`, `fseek()`, system call `open()`

fflush — Flush a stream buffer

Synopsis

```
#include <stdio.h>

int fflush (stream)
FILE *stream;
```

Description

`fflush` causes any buffered data associated with `stream` to be written out to the associated file or device; because input and output are not buffered simultaneously in this implementation, this has an effect only on streams that have been opened for write or update.

The open status of the stream is unaffected.

Diagnostics

`EOF` is returned if `stream` is a not an output stream, or if there is an error writing to the file.

Notes

It is not normally necessary to call `fflush`, but it can be useful when mixing output between `stdout` (which is normally buffered) and `stderr` (which is not). If `fflush` is not used and `stdout` and `stderr` are both connected to a terminal, it is possible for `stderr` messages to appear before messages written to `stdout`, even when the write to `stdout` was performed first. Calling `fflush(stdout)` before a write to `stderr` may help in this case.

Normal program termination indirectly causes all open streams to be flushed.

See Also

`fopen()`, `setbuf()`

fgetc, getc, getchar, ungetc — Character input from streams

Synopsis

```
#include <stdio.h>

int fgetc (stream)
FILE *stream;

int getc (stream)
FILE *stream;

int getchar()

int ungetc (c, stream)
int c;
FILE *stream;
```

Description

`fgetc` reads the next character from stream and returns its integer value, or `EOF` if end of file or an error is encountered.

`getc` is equivalent to `fgetc`, except that it may be implemented as a macro which evaluates `stream` more than once. At this time, this is not true of this implementation.

`getchar` is a macro, equivalent to `fgetc (stdin)`.

`ungetc` pushes `c` back to `stream`. The input file buffer is altered such that the next call to `fgetc` returns `c`. Up to one character may be pushed back, and at least one character must have been read from `stream` before a call to `ungetc` may be made.

Any subsequent call to `fseek()` will erase any push-back done with `ungetc`.

Notes

Under OS-9 there are two different ways to get input from a file or device. The `read()` system call will read characters up to a specified number in “raw” mode; that is, no editing takes place on the input stream, and characters appear to the program exactly as present in the file. On the other hand, the `readln()` system call honors the various mappings of characters associated with a character device (such as a terminal), allows a user to edit text to be input, and returns a final string to the caller once a carriage return is seen on the input.

In the vast majority of cases, it is preferable to use `readln()` for accessing character devices, and `read()` for any other file input; so on OS-9, `fgetc` uses this strategy and, since most stream input using the standard library functions is routed *through* `fgetc`, so do most other

input functions. The choice is made when the first call to `fgetc` is made after the file has been opened. The system is consulted for the status of the file, and a flag is set in the file structure accordingly. The choice may be forced by the programmer by setting the relevant bit *before* any calls to `fgetc` are made. The flags for doing this are defined in `<stdio.h>` as `_SCF` and `_RBF`, and the method is as follows:

Given a stream `f`, `f->_flag |= _SCF;` will force the use of `readln()` for text input on `f`; and `f->_flag |= _RBF;` will force the use of `read()`. This trick may also be played on the standard input stream `stdin` without the need to call `fopen()`, provided this is done before any input has been requested.

Diagnostics

`EOF` is returned in case of end-of-file or error, and `errno` is set accordingly.

See Also

`fputc()`, `fread()`, `fopen()`, `fgets()`, system calls `read()`, `readln()`

fgets — Read a string from a stream

Synopsis

```
#include <stdio.h>

char *fgets(s, size, stream)
char *s;
int size;
FILE *stream;
```

Description

fgets reads up to (size − 1) characters from stream and stores them into the buffer pointed to by s. Reading stops after encountering end of file or a newline. If a newline is read, it is stored in the buffer. A terminating null character is stored after the last character read.

A line of arbitrary length may be read using a limited-size buffer by checking the last character of the returned string for newline; if the last character read is a newline, or a subsequent call to fgets returns NULL (as occurs when end of file is encountered before successfully reading any characters), then the read is complete.

Return Value

fgets returns s on success, and NULL on error or when end of file occurred after having read no characters.

See Also

fread(), fgetc(), puts(), scanf()

fopen, freopen — Open a stream

Synopsis

```
#include <stdio.h>

FILE *fopen (pathname, mode)
char *pathname, *mode;

FILE *freopen (pathname, mode, stream)
char *pathname, *mode;
FILE *stream;
```

Description

`fopen` opens the file whose name is the string pointed to by `pathname`, and associates it with a stream.

The argument `mode` points to a string beginning with one of the following sequences:

`r` Open for reading. The stream is positioned at the beginning of the file.

`r+` Open for reading and writing. The stream is positioned at the beginning of the file.

`w` Truncate file to zero length or create file for writing. The stream is positioned at the beginning of the file.

`w+` Truncate file to zero length or create file for reading and writing. The stream is positioned at the beginning of the file.

`a` Open for appending (writing at end of file). The file is created if it doesn't exist. The stream is positioned at the end of the file.

`a+` Open for reading and appending. The file is created if it doesn't exist. Output is always written to the end of the file.

`d` Open a directory for reading. The stream is positioned at the beginning of the directory. This is an extension, not necessarily portable to other systems.

As an extension, any mode may have an `x` immediately after the initial letter, indicating that relative paths are relative to the current execution directory, and that the file should have execute permission. For example,

```
f = fopen ("fred", "wx+");
```

Three streams are pre-opened at program start:

`stdin` the standard input, using OS-9 path number 0

`stdout` the standard output, using OS-9 path number 1

`stderr` the standard error output, using OS-9 path number 2

All streams are automatically buffered except `stderr`, unless explicitly made unbuffered by calling `setbuf()`.

`freopen` opens the file whose name is pointed to by `pathname`, and associates it with `stream`. The original stream, if it exists, is closed. The `mode` argument is used just as in the `fopen` function.

The primary use of `freopen` is to change the file associated with a standard text stream (`stdin`, `stdout`, `stderr`).

Caveats

The `mode` passed as an argument must be a string, not a character constant. It is a common error to pass a character literal (which is of type **`int`**) rather than a string containing one character.

Diagnostics

These functions return `NULL` if the open was unsuccessful, and set `errno` accordingly.

Bugs

This implementation does not currently support the `b` modifier from the ANSI standard for `mode`. If used, these functions will fail.

See Also

`fclose()`, `fdopen()`, system call `open()`

fputc, fputs, putc, putchar, puts — Character output to streams

Synopsis

```
#include <stdio.h>

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

int fputs (s, stream)
char *s;
FILE *stream;

int putc (c, stream)
int c;
FILE *stream;

int putchar (c)
int c;

int puts (s)
char *s;
```

Description

`fputc` writes the character `c` to `stream` at the current writing position.

`fputs` writes the string `s` to `stream`, without its terminating null character.

`putc` is equivalent to `fputc`, except it may be implemented as a macro which evaluates its argument multiple times. At this time, this is not true of this implementation.

`putchar` is a macro, equivalent to `fputc(c, stdout)`.

Output via `fputc` is normally buffered except when the buffering is disabled by a call to `setbuf()`, or the stream being written to is `stderr`.

puts writes the string `s`, and a trailing newline, to `stdout`.

Diagnostics

`fputc`, `putc`, and `putchar` return `c` from a successful call, and `EOF` on end of file or error.

`fputs` and `puts` return a non-negative number on success, and `EOF` on end of file or error.

Caveats

The `puts` function appends a newline, while `fputs` does not. This inconsistency is dictated by history and backwards compatibility.

See Also

`fgetc()`, `fwrite()`, `printf()`

fread, fwrite — Binary stream input/output

Synopsis

```
#include <stdio.h>

size_t fread (ptr, size, count, stream)
char *buf;
size_t size, count;
FILE *stream;

size_t fwrite (ptr, size, count, stream)
char *buf;
size_t size, count;
FILE *stream;
```

Description

`fread` reads `count` items of data, each `size` bytes in length, from the stream pointed to by `stream`, and stores them in the buffer pointed at by `ptr`.

`fwrite` writes `count` items of data, each `size` bytes in length, to `stream`; reading from the memory area pointed at by `ptr`.

Return Value

On success, these functions return the number of items actually read or written. The only time that number is equal to the number of bytes transferred is when `size` is 1. If end of file is reached, or if an error occurs, the number of items read/written will be less than requested (or zero). When reading or writing multiple objects, it may be useful to use a construction such as:

```
if (count != fread (ptr, sizeof (int), count, stream)) {
    /* executed on short read */
}
```

Notes

The best way to find the appropriate `size` is usually by use of the **`sizeof`** operator.

`fread` does not distinguish between the end of file and error conditions; use `feof()` or `ferror()` to find out which occurred.

See Also

`feof()`, `ferror()`, system calls `read()`, Write, Writeln — write to a file or device

frexp — Split a floating-point number into its components

Synopsis

Description

See Also

fseek, ftell, rewind — Reposition a stream

Synopsis

```
#include <stdio.h>

int fseek (stream, offset, whence)
FILE *stream;
long offset;
int whence;

long ftell (stream)
FILE *stream;

rewind (stream)
FILE *stream;
```

Description

`fseek` repositions the next character position for the stream pointed to by `stream`. The new position is obtained by adding `offset` bytes to the position specified by `whence`. If `whence` is set to `SEEK_SET`(0), `SEEK_CUR`(1), or `SEEK_END`(2), the offset is relative to the start of the file, the current position, or the file's end, respectively. A successful call to `fseek` clears the end-of-file indicator, and undoes any effects of `ungetc()` on the same stream.

`ftell` retrieves the current file position (measured in bytes from the beginning) for the stream pointed to by `stream`.

`rewind` performs the equivalent of `fseek (stream, 0L, SEEK_SET);` to return the file position to the beginning of the file. In addition, the error indicator for stream is also cleared (see `clearerr()`).

Return Value

On success, `fseek` returns `0`, and `ftell` returns the current offset; otherwise `EOF` is returned and `errno` is set to indicate the error.

The `rewind` function returns no value.

See Also

System call lseek — position in file

gets — Read a string from `stdin` (deprecated)

Synopsis

```
#include <stdio.h>

char *gets(s)
char *s;
```

Description

`gets` reads a line from `stdin` into the buffer pointed to by `s`, reading until either a terminating newline or `EOF` is encountered. Once reading is complete, a terminating null byte is stored in the buffer.

Return Value

`gets` returns `s` on success, and `NULL` on error or when end of file is encountered before any characters are read.

Bugs

`gets` is broken by design. It is impossible to tell, without knowing the data in advance, how many characters `gets` will read. Because the function has no built-in limiting functionality, it is able to overwrite any possible buffer you can allocate and is perfectly capable of destroying the program.

Never use this function.

See Also

`fgets()`, `fputc()`, `fgetc()`, `fread()`, `puts()`, `scanf()`

is* — Character classification functions

Synopsis

```
#include <ctype.h>
isclass(c);
```

Description

These functions use a look-up table to classify characters according to their character value. The header file defines them as macros, which means that they are implemented as inline code rather than a function call.

Each results in non-zero for true, or zero for false.

The value is guaranteed to be correct for any integer values in ASCII, but the result is unpredictable outside the range $[\![-1..127]\!]$.

The truth tested by each function is as follows:

`isalpha` c is a letter

`isdigit` c is a digit

`isupper` c is an uppercase letter A-Z

`islower` c is a lower case letter a-z

`isalnum` c is a letter or a digit

`isspace` c is a white space character (horizontal or vertical tab, newline, carriage return, line-feed, form-feed)

`iscntrl` c is a control character ($[\![0..32, 127]\!]$)

`ispunct` c is neither a control character, nor alphanumeric

`isprint` c is printable $[\![32..126]\!]$

`isascii` c is in the range $[\![-1..127]\!]$ (-1 is a special case)

localtime

malloc, calloc, realloc, free — Allocate and free memory

Synopsis

```
#include <stdlib.h>

char *malloc (size)
size_t size;

free (ptr)
char *ptr;

char *calloc (nmemb, size)
size_t nmemb, size;

char *realloc (ptr, size)
char *ptr;
size_t size;
```

Description

The `malloc` family of functions dynamically allocate memory for the program's use, by requesting new blocks of memory no smaller than 1 page (256 bytes) from OS-9 and subdividing those blocks into memory regions returned. There is a small amount of overhead involved in that the allocator must maintain some accounting information used to mark "allocated" vs. "free" sections.

The `malloc` function allocates `size` bytes and returns a pointer to the allocated memory. No attempt is made to initialize the memory allocated to any value or pattern.

`calloc` allocates space for an array of `nmemb` members, each of which is `size` bytes wide, and returns a pointer to the memory allocated. `calloc` initializes the allocated space to zero.

`realloc` changes the size of the memory block pointed to by `ptr`, to `size` bytes. The contents will be unchanged in the range from the start of the region, up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If `ptr` is `NULL`, the call is equivalent to `malloc(size)`. If `size` is zero, the call is equivalent to `free(ptr)`.

`free` releases memory pointed to by `ptr`, which must have been returned by a previous call to `malloc`, `calloc`, or `realloc`. Otherwise, or if `free(ptr)` has already been called before, the result is not defined. If `ptr` is `NULL`, nothing is done.

Return Value

`malloc` and `calloc` return `NULL` if no free memory can be added, or if there was an error. `NULL` may also be returned by `malloc` with a `size` of zero, or by `calloc` with `nmemb` or `size` equal to zero.

`free` returns no value.

The pointer returned by `realloc` *may* be the same as `ptr` if the allocation was able to add bytes to the previously allocated memory block, or it may be different if it was necessary to move the allocation to a new address. If `realloc` fails the original block is left unmodified, neither freed nor moved.

Notes

`calloc` does not check that the multiplication `nmemb * size` will not result in integer overflow, but due to address space limitations of the 6x09 attempting to allocate more than 65535 bytes will not succeed.

See Also

System calls ibrk, `brk()`, unbrk

memchr — Scan memory for a character

Synopsis

```
#include <string.h>

char *memchr (s, c, n)
char *s;
int c;
size_t n;
```

Description

The `memchr` function scans the initial `n` bytes of the memory area pointed to by `s` for the first instance of `c`. Both `c` and the bytes of the memory area are interpreted as **`unsigned char`**.

Return Value

The `memchr` function returns a pointer to the matching byte, or `NULL` if the character is not found in the given memory area.

See Also

`strchr()`

memcpy, memccpy — Copy a memory area

Synopsis

```
#include <string.h>

char *memcpy (dest, src, n)
char *dest, *src;
size_t n;

char *memccpy (dest, src, c, n)
char *dest, *src;
int c;
size_t n;
```

Description

The `memcpy` function copies `n` bytes from memory area `src` to memory area `dest`.

`memccpy` copies up to `n` bytes from memory area `src` to memory area `dest`, stopping when the character `c` is encountered.

For both functions, the two memory areas must not overlap.

Return Value

`memcpy` returns a pointer to `dest`.

`memccpy` returns a pointer to the next byte in `dest` after `c`, or `NULL` if `c` was not found in the first `n` bytes of `src`.

See Also

`strcpy()`

memset — Fill memory area with a constant byte

Synopsis

```
#include <string.h>

char *memset (s, c, n)
char *s;
int c;
size_t n;
```

Description

The `memset` function writes the byte value of `c` to the first `n` bytes of the memory area pointed to by `s`.

Return Value

The `memset` function returns a pointer to the memory area `s`.

See Also

`calloc()`

printf, fprintf, sprintf — Formatted output conversion

Synopsis

```
#include <stdio.h>

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

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

int sprintf (str, format, ...)
char *str, *format;
```

Description

The functions in the `printf` family produce output described by a format string as described below. `printf` sends its output to `stdout`, `fprintf` outputs to `stream`, and `sprintf` formats into a buffer pointed at by `str`.

The `format` string determines the form, type, and number of the arguments expected by the function. If the `format` does not match the arguments correctly, the results are unpredictable.

The `format` may contain characters to be copied directly to the output and/or format specifications. Each format specification causes the function to take the next successive argument for the next output conversion.

Each conversion specification begins with a `%` character. Following the `%`, there may be

- An optional minus sign (`-`), denoting the use of left justification in the field.
- An optional sequence of decimal digits indicating a *field width*. The field will be at least this wide, and may be wider if the conversion requires it. The field will be padded on the left unless the above minus sign is present, in which case it will be padded on the right. The padding character is by default a space, but if the field width starts with a zero (`0`), it will be zero-padded instead.
- An optional period (`.`) and a sequence of decimal digits, the *precision*, which for floating point arguments indicates the number of digits to follow the decimal point on conversion, and for strings, the maximum number of characters from the string argument that are to be accepted from the input.
- An optional character `l` indicates that the following `d`, `o`, `x`, or `X` specifies a **`long int`** argument instead of **`int`**.

- A conversion character which shows the type of the argument and the desired conversion. The recognized conversion characters are:

`%`	Does not consume an argument; inserts a literal `%` character.
`c`	The **`int`** argument is interpreted as an ASCII character.
`s`	Characters from the **`char`** `*` argument are copied up to the next null character, or until the number of characters indicated by the field precision have been copied. If the precision is zero or missing, characters are not counted.
`d`, `o`, `x`, `X`	The **`int`** argument is formatted as a decimal, octal, or hexadecimal string. `X` prints in upper case.
`u`	The **`int`** argument is formatted as a decimal number in the range 0..65535.
`f`	The **`double`** argument is formatted as `[-]nnn.nnn`, where the digits after the decimal point are specified as above. If not specified, the precision defaults to six digits. If the precision is 0, no decimal point or following digits are printed.
`e`, `E`	The **`double`** argument is formatted using scientific E notation as `[-]n.nnne±nn`; with one digit before the decimal point, and the precision controlling the number of following digits. Using `E` as a format specifier results in upper case.
`g`, `G`	The **`double`** argument is formatted in either `f` format or `e` format, whichever produces the shortest string.

Notes

In order to save program space, by default `printf` does not contain code to format long integers or floating point numbers. If you need to format long integers, your program must contain a reference to the function `pflinit()` to force the inclusion of the necessary code. Likewise, if you need to format floating point numbers, your program must reference the function `pffinit()` somewhere. Even if these functions are never called, the reference is sufficient to cause the code to be included.

In **`double`** conversions, the final digit is rounded.

When using `sprintf`, it is the caller's responsibility to ensure that the buffer `str` is large enough to hold all output.

See Also

`fputc()`, `scanf()`

qsort — Sort an array

Synopsis

```
#include <stdlib.h>

qsort (base, nmemb, size, compar)
char *base;
size_t nmemb, size;
int (*compar) ();
```

Description

`qsort` implements the quick-sort algorithm for sorting an arbitrary array of items.

`base` is the address of an array of `nmemb` elements, each of which is `size` bytes in width. A comparison routine, `compar`, is supplied by the caller. The `compar` routine is called with two arguments, which are pointers to the elements being compared. The routine is expected to return an integer less than, equal to, or greater than zero if the first argument is to be considered less than, equal to, or greater than, the second argument. In this manner, any arbitrary type of data may be sorted as long as the comparison function knows what it is.

See Also

`strcmp()`

scanf, fscanf, sscanf — Input string interpretation

Synopsis

```
#include <stdio.h>

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

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

int sscanf (char *str, char *format, ...)
char *str, *format;
```

Description

The `scanf` family of functions performs the complement of the `printf()` family – interpreting formatted input and converting it into usable forms. The functions scan input according to format, as described below. The format may contain conversion specifications – the results from such conversions, if any, are stored in the locations pointed to by the pointer arguments following `format`. Each pointer argument must be typed appropriately by the corresponding conversion specifier.

If the number of conversion specifications in `format` is greater than the number of pointer arguments following `format`, the results are undefined. If the number of pointer arguments is greater than the number of conversion specifications, any excess pointer arguments are ignored.

The `scanf` function performs conversions from the standard input, `fscanf` converts from `stream`, and `sscanf` from the string pointed to by `str`.

Each function expects a format string containing conversion specifications, and zero or more pointers to objects into which the converted values may be stored.

The `format` string consists of a series of directives describing how the function is to process the input. If at any time a directive fails, no further input is interpreted and the function returns.

A directive is one of the following:

- One or more whitespace characters (as recognized by `isspace()`), which matches zero or more whitespace characters in the input.
- An ordinary character (anything other than whitespace or `%`). This must exactly match the next input character.
- A conversion specification, beginning with `%` followed by:

- an optional * indicates suppression of assignment: the function reads input as directed, but discards the input, does not use a pointer argument, and does not count as a successful assignment.
- an optional decimal integer specifying the *maximum field width*. Reading stops when this maximum is reached, or when any non-matching character is encountered. Most conversions discard initial whitespace characters, which don't count toward the field width maximum. The terminating null byte used for string conversions also does not count towards the field width.
- A conversion specifier, one of the following:

`%`	Matches a literal `%`. The conversion matches a single input `%` character. No conversion is done, and no assignment takes place.
`[`	Matches against a character set, defined between the initial `[` and a terminating `]` character. Whitespace skipping is suppressed. The next pointer argument must be a pointer to the first character of an array long enough for the maximum field width plus a terminating null byte. Characters are copied from the input as long as they exist within the character set, unless the first character is a caret (^), in which case characters are copied until a character in the set is encountered.
`c`	Matches a sequence of characters with a length specified by the maximum field width (default 1); the next pointer must be a pointer to **`char`**, and the pointed-to buffer must have space for all characters. No terminating null byte is written after the end. Whitespace skipping is suppressed — if you wish to skip spaces, insert a leading space before the conversion specification.
`d`	Matches an optionally signed decimal integer string; the next pointer must be a pointer to **`int`**.
`e`, `f`	Matches an optionally signed floating-point number; the next pointer must be a pointer to **`float`**.
`o`	Matches an unsigned octal integer string; the next pointer must be a pointer to **`unsigned int`**.
`s`	Matches a sequence of non-whitespace characters; the next pointer must be a pointer to the first element of a character array long enough to hold the input sequence and the terminating null byte. The input string ends at the first whitespace character or the maximum field width, whichever comes first.
`x`	Matches an unsigned hexadecimal integer string; the next pointer must be a pointer to **`unsigned int`**.
`D`	Just like `d`, except the next pointer must be a pointer to **`long int`**.
`E`, `F`	Just like `e` and `f`, except the next pointer must be a pointer to **`double`**.
`O`, `X`	Just like `o` and `x`, except the next pointer must be a pointer to **`unsigned long int`**.

Each of the functions returns a count of the number of fields successfully matched and assigned.

Caveats

The returned count of matches/assignments does not include character matches or assignments suppressed by `*`.

The arguments *must* all be pointers. It is a common error to call `scanf` with the value of an item rather than a pointer to it.

Diagnostics

These functions return `EOF` on end of input or error and a count which is shorter than expected for unexpected or unmatched items.

See Also

`atof()`, `fgetc()`, `printf()`

sin, cos, tan, asin, acos, atan — Trigonometric functions

Synopsis

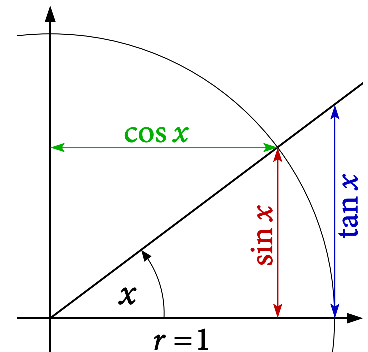
```
#include <math.h>

double sin(x) double x;
double cos(x) double x;
double tan(x) double x;
double asin(x) double x;
double acos(x) double x;
double atan(x) double x;

deg();
rad();
```

Link with `-t`.

Description



The `sin` function computes the sine corresponding to an angle of `x`. In trigonometry, the sine function gives the ratio between the opposite (vertical) side and the hypotenuse of a right triangle with the given angle.

`cos` computes the cosine of the angle `x`. The cosine function gives the ratio between the adjacent (horizontal) side and the hypotenuse of a right triangle with the given angle.

`tan` computes the tangent of the angle `x`, where tangent is defined as $\tan x = \frac{\sin x}{\cos x} = \frac{opposite}{adjacent}$.

`asin` returns the inverse sine ($\sin^{-1} x$, also known as the arcsine) of `x` , where `x` is a number representing the ratio between the opposite side of a right triangle and its hypotenuse (that is, the sine). Given the sine, the `asin` function is used to compute a matching angle.

`acos` returns the inverse cosine ($\cos^{-1} x$, the arc cosine) of `x`. Given the cosine, the `acos` function is used to compute the original angle.

`atan` returns the inverse tangent ($\tan^{-1} x$, the arc tangent) of `x`. Given the tangent, the `atan` function is used to compute the original angle.

Notes

To maximize speed, it is often better to save the results of a calculation for later reuse rather than to call the same function with the same argument a second time, or to pre-calculate a number of values that will be used at runtime.

As an extension to the C standards, this implementation has the ability to perform trigonometric calculations using either radians or degrees. At program start, the trig functions are configured to use radians; calling the `deg` function causes subsequent calls to return and accept angles in degrees, and calling the `rad` function restores the default behavior. It is possible to detect which mode the functions are in, using the global `_deg` variable. When the value of `_deg` is nonzero, trigonometric functions operate using degrees as measurement; otherwise, they use radians.

To avoid messing up user code, library functions that use trigonometry should always save and restore the mode of these functions before returning to the program.

Caveats

These functions generate an `EILLARG` (Illegal Argument) error when passed values outside the domain of the function. Because the tangent is defined as the slope of a line, the `tan` function is not defined for any *odd* multiple of $\frac{\pi}{2}$ radians, or 90 degrees — that is, $(\pm\frac{\pi}{2}, \frac{3\pi}{2}, \frac{5\pi}{2} \cdots)$ or $(\pm 90°, 270°, 540° \cdots)$.

The `asin` and `acos` functions are defined only over the interval $[-1, 1]$.

See Also

`sinh()`

sinh, cosh, tanh, asinh, acosh, atanh — Hyperbolic functions

Synopsis

```
#include <math.h>

double sinh (x) double x;
double cosh (x) double x;
double tanh (x) double x;
double asinh (x) double x;
double acosh (x) double x;
double atanh (x) double x;
```

Link with `-t`.

Description

The `sinh` function implements the hyperbolic *equivalent* of the trigonometric sine function. Unlike the trigonometric functions, the hyperbolic functions are not periodic; the hyperbolic sine is defined as $\sinh x = \frac{e^{2x}-1}{2e^x}$, where e is the base of the natural logarithm.

`cosh` implements the hyperbolic cosine function, defined as $\cosh x = \frac{e^{2x}+1}{2e^x}$.

`tanh` returns the hyperbolic tangent, defined as $\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^{2x}-1}{e^{2x}+1}$.

`asinh` is the inverse of the `sinh` function, returning the original hyperbolic angle. It is defined as $\sinh^{-1} x = \ln{(x + \sqrt{x^2+1})}$.

`acosh` is the inverse of the `cosh` function, returning the original hyperbolic angle. It is defined as $\cosh^{-1} x = \ln{(x + \sqrt{x^2-1})}$.

`atanh` is the inverse of the `tanh` function, and is defined as $\tanh^{-1} x = \frac{1}{2} \ln{(\frac{1+x}{1-x})}$

Caveats

These functions generate an `EILLARG` (Illegal Argument) error when passed values outside the domain of the function. Because logarithms are not defined for real numbers less than *or equal to* zero, `acosh` is valid only over the interval $[1, \infty)$; and `atanh` is valid only over the interval $(-1, 1)$.

See Also

`log()`, `pow()`, `sqrt()`

setbuf — Fix file buffer

Synopsis

```
#include <stdio.h>

setbuf (stream, buf)
FILE *stream;
char *buf;
```

Description

Immediately before the first character is read from or written to a newly opened stream, a buffer is obtained from the system and assigned to it. If you want to control or disable the buffering `setbuf` may be used to forestall this by assigning a user buffer to the file.

`setbuf` must be used after the file has been opened and before any I/O has taken place.

The buffer must be of sufficient size and a value for a manifest constant, `BUFSIZ`, is defined in the header file for use in declarations.

If `buf` is `NULL`, the stream becomes unbuffered and characters are read or written singly.

Notes

The standard error output is unbuffered by default, and the standard output is buffered by default. This can mean that error output may actually be printed to the screen before normal output. For this reason, it is sometimes useful to call `fflush()` on `stdout` before writing to `stderr`.

See Also

`fopen()`, `fflush()`, `fgetc()`, `fputc()`

setjmp, longjmp — Jump to another function

Synopsis

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

longjmp (env, val)
jmp_buf env;
int val;
```

Description

These functions allow the return of program control directly to a higher level function. They are most useful when dealing with errors and interrupts encountered in a low level routine.

In C, the **goto** keyword has scope only in the function in which it is used; that is, a label *must* reside in the same function as any **goto** targeting it. Control may be transferred elsewhere only by means of a function call, which returns to the same location it was called from. In certain unusual situations, it is preferable to be able to start some section of code again, but this would mean returning up a ladder of function calls with error indications all the way.

The functions described here are used for performing such "non-local gotos": transferring execution from one function, to a predetermined location in another function. The `setjmp` function sets the target, and `longjmp` actually performs the jump.

The `setjmp` function is used to save various information about the calling environment (the stack pointer, the program counter, and possibly other registers) in the buffer `env` for later use by a call to `longjmp` using that same buffer. When called, `setjmp` returns 0.

`longjmp` uses the information contained within the buffer `env` to transfer control back to the point where `setjmp` was called; and to restore, or "rewind", the stack to its state when `setjmp` had been called, but before it had returned.

Following a successful `longjmp`, execution continues as if `setjmp` had returned a second time. This "fake" return can be distinguished from the return value — `setjmp` always returns 0, while the return from a `longjmp` call is *never* zero (supplied in `val`, but if `val` is 0 it's replaced by 1).

Notes

The function from which `setjmp` is called must not have returned when `longjmp` is called; if it has returned, the contents of the stack (including any **auto** variables declared within the function) will be unpredictable.

sqrt — Square root function

Synopsis

```
#include <math.h>

double sqrt (x)
double x;
```

Link with `-t`.

Description

On success, the `sqrt` function returns the non-negative square root of `x`.

If `x` is zero, zero is returned.

Caveats

Since this implementation does not support complex numbers, it is not possible to take the square root of a negative number.

strcat, strncat — Concatenate two strings

Synopsis

```
#include <string.h>

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

char *strncat (dest, src, n)
char *dest, *src;
size_t n;
```

Description

The `strcat` function appends the `src` string onto the end of the `dest` string, starting at the original terminating null byte, copying the bytes of `src`, and then appending a new terminating null byte. The strings may not overlap, and `dest` must be large enough to contain the new combined string.

`strncat` is similar, except that it uses at most n bytes from `src`; and if the length of `src` is at least `n` bytes, it does not need to be null-terminated. Just as with `strcat`, the resulting combined string is always null-terminated. As a consequence, the actual size of `dest` must be at least `strlen(dest)+n+1` bytes to avoid a buffer overrun.

Return Value

The `strcat` and `strncat` functions return a pointer to the resulting string `dest`.

Caveats

These functions have no means of checking that the space provided is large enough to hold the combined string. If `dest` is *not* large enough, the program's behavior becomes unpredictable — buffer overruns are an excellent way to attack programs that are meant to be secure.

See Also

`strchr()`, `strcmp()`, `strcpy()`

strchr, strrchr — Locate a character in string

Synopsis

```
#include <string.h>

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

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

Description

The `strchr` function returns a pointer to the first occurrence of the character `c` in the string `s`.

`strrchr` returns a pointer to the *last* occurrence of the character `c` in the string `s`.

`index` and `rindex` are alternate names for `strchr` and `strrchr`, respectively. In this implementation, they use the same code and may be used interchangeably without increasing program size.

Return Value

The `strchr` and `strrchr` functions return a pointer to the matched character, or `NULL` if the character was not found. The terminating null byte is considered part of the string, so if `c` is null (`'\0'`), these functions will return a pointer to the terminator.

Caveats

In these functions, “character” means “byte”; the argument `c` is cast to **`unsigned char`** before doing any comparisons.

See Also

`findstr()`.

strcmp, strncmp — Compare two strings

Synopsis

```
#include <string.h>

int strcmp (s1, s2)
char *s1, s2;

int strncmp (s1, s2, n)
char *s1, *s2;
size_t n;
```

Description

The `strcmp` function compares two strings, `s1` and `s2`, for lexicographic order.

`strncmp` does the same, but instead of potentially continuing to check forever, compares at most `n` characters.

Return Value

These functions return an integer less than, equal to, or greater than 0 depending on whether `s1` is less than, equal to, or greater than `s2`.

See Also

`findstr()`

strcpy, strncpy — Copy string data

Synopsis

```
#include <string.h>

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

char *strncpy (dest, src, n)
char *dest, *src;
size_t n;
```

Description

The `strcpy` function copies the string pointed to by `src`, including its terminating null byte, to the buffer pointed to by `dest`. The destination must be large enough to contain the copy, and the two memory areas may not overlap.

`strncpy` is similar, except at most n bytes are copied from `src`. If there is no null byte present in the first `n` bytes of `src`, the string placed in `dest` will not be null-terminated. If the length of `src` is fewer than `n` bytes, `strncpy` writes additional null bytes to ensure that exactly `n` bytes are written.

Return Value

Both functions return a pointer to the destination string.

Caveats

`strcpy` has no means of checking that the space provided is large enough. It is the user's responsibility to ensure that string space does not overflow.

See Also

`findstr()`, `memcpy()`, `strcat()`

strlen — Get the length of a string

Synopsis

```
#include <string.h>

size_t strlen (s)
char *s;
```

Description

The `strlen` function finds the length of the string pointed to by `s`, excluding its terminating null byte (`'\0'`).

See Also

`findstr()`, `strchr()`, `strcmp()`, `strcpy()`

strpbrk — Search a string for any of a set of bytes

Synopsis

```
#include <string.h>

char *strpbrk (s, accept)
char *s, *accept;
```

Description

The `strpbrk` function locates the first occurrence within the string `s`, of any character found in the string `accept`.

Return Value

This function returns a pointer to a byte in `s` that is equal to one of the bytes in `accept`, or `NULL` if no matching byte is present.

See Also

`memchr()`, `strchr()`, `strspn()`, `strtok()`

strspn, strcspn — Get length of a prefix set

Synopsis

```
#include <string.h>

size_t strspn (s, accept)
char *s, *accept;

size_t strcspn (s, reject)
char *s, *reject;
```

Description

The `strspn` function returns the length of the initial segment of `s` which consists only of characters present in the string `accept`.

`strcspn` is similar, except it returns the length of the initial segment of `s` which consists only of characters *not* present in the string `reject`.

The second argument of these functions is treated as a set of characters, not as a substring. Each function

strtok — Extract tokens from a string

Synopsis

```
#include <string.h>

char *strtok (str, delim)
char *str, *delim;
```

Description

The `strtok` function breaks a string into a sequence of zero or more non-empty tokens. On the first call to `strtok`, the string to be parsed is specified in `str`. For each subsequent call parsing the same string, `str` must be `NULL`. The `delim` argument specifies a set of byte values that are used to delimit the tokens in the parsed string. You may specify different delimiter strings for different calls parsing the same string.

Each call to `strtok` returns a pointer to a null-terminated string containing the next token. This string does not include any delimiting bytes. If no more tokens are found, `strtok` returns `NULL`.

Intermediate state A sequence of calls to `strtok` operating on the same string maintains an internal pointer determining where to start searching for the next token. The first call to `strtok` initially sets this pointer to the first byte of `str`. The start of the next token is found by scanning forward for the next non-delimiter byte from the internal string pointer. If such a byte exists, it is recognized as the start of the next token. If no such byte exists, there are no more tokens and `NULL` is returned. In this way, a string that is empty or that contains only bytes found in `delim`, will return `NULL` the first time it is called.

The end of each token is found by scanning forward in the string until encountering either the next delimiter or a terminating null byte (`'\0'`). If a delimiter is found, it is overwritten with a null byte to end the current token, and `strtok` updates the internal pointer to the next byte for locating the next token. A pointer to the start of the token is then returned.

Under typical use, this means that a sequence of two or more adjacent delimiter bytes is considered to be a single delimiter, and delimiters at the start and end of a string are ignored (with the caveat that using different sets of delimiters can change this in various ways). That is, `strtok` does not return empty strings.

Return Value

The `strtok` function returns a pointer to the next token, or `NULL` if there are no more tokens.

Caveats

This function modifies its first argument. For this reason, it should be noted that:

- You should not pass a pointer to a constant string as the first argument — that would cause `strtok` to modify the in-memory copy of the program, and on many systems it will actually succeed in corrupting it. This is bad.
- Since the delimiting byte ending a token is rewritten with a null byte on return, there is no way to tell the identity of the terminating delimiter.

See Also

`memchr()`, `strchr()`, `strpbrk()`, strspn

system — Issue a shell command

Synopsis

```
#include <stdlib.h>

int system (command)
char *command;
```

Description

The `system` function uses `os9fork()` to create a child process that executes the OS-9 shell command `command`, and waits for it to exit. The exit status of the shell is returned.

Notes

The `system` function is simple and convenient, handling the details of creating a child process and waiting for it to exit; and by calling the OS-9 shell to run the program, it enables you to use I/O redirection for `command`. However, while convenience is gained, flexibility and efficiency are lost. If you don't need to

Caveats

The maximum length of string for the command line is 80 characters. If a longer string is needed, call `os9fork()` directly.

See Also

System calls `os9fork()`, wait — wait for child process to end.

time — get the current clock time

tolower, toupper — Convert uppercase or lowercase

Synopsis

```
#include <ctype.h>

int tolower (c)
int c;

int toupper (c)
int c;
```

Description

These functions convert uppercase letters to lowercase, and vice versa.

If the integer `c` has the value of an uppercase letter, `tolower` returns its lowercase equivalent; otherwise, it returns `c`.

`toupper` does the opposite: if `c` has the value of a lowercase letter, `toupper` returns its uppercase equivalent; otherwise, returning `c`.

For both functions, `c` must be within the range of the type **`unsigned char`**, or have the value `EOF`. For any other values, the result is undefined.

Notes

The header file `<ctype.h>` also defines the macros `_tolower` and `_toupper` as an extension to the C library. These macros are similar to the functions, except they don't check their argument. `_tolower` always modifies its argument as if it were converting an uppercase letter to lowercase, and `_toupper` always modifies its argument as if converting from lowercase to uppercase. *Use these macros only when you are **certain** of the input.*

Bugs

The set of recognized letters is currently limited to the range of ASCII; extended characters are not considered.

2 Standard Library Extensions

The DCC standard library contains a number of functions not present in C standards. Many of these extensions are from versions of the UNIX C libraries, and others are for compatibility with other versions of OS-9; but none of them are specified by C.

This chapter introduces a new optional heading, **Conforming To**. In this and later chapters, Conforming To describes the origins of the function(s) being described — where they come from, and where else they may be used.

A few names/acronyms that may not be familiar to everyone:

BSD The Berkeley Software Distribution, a version of Unix released by the University of California and forming the basis for many modern Unix-like systems. BSD was the start of one of the two major groups that took part in the "Unix wars" of the 1990s.

OSK An abbreviation of Microware's version of OS-9 for Motorola 68000-derived processors, marketed variously as "Professional OS-9", "OS-9 for 68000", and "OS9/68K" at various times.

OS-9000 Microware's flagship modern OS-9 product, a relatively portable re-implementation of the OS-9 operating system in the C language. Available for many different CPU types.

POSIX.1 The Portable Operating System Interface (formally, IEEE 1003) is a set of standard operating system interfaces based on the Unix operating system. POSIX is the entire set of standards, but "POSIX.1" defines the basic set of common C programming interfaces implemented by most modern Unix-like systems, whether it be Linux, FreeBSD, Mac OS, etc. POSIX was, effectively, the peace treaty that mostly brought the competing Unix vendors back together.

SVID The System V Interface Definition, the specifications of which describe AT&T UNIX System V. System V is one of the attempts AT&T made to commercialize Unix, and the start of the *other* major group that took part in the "Unix wars".

fdopen — Open a stream on a system path number

Synopsis

```
#include <stdio.h>

FILE *fdopen (fd, mode)
int fd;
char *mode;
```

Description

fdopen associates a stream with an existing path number fd. The mode of the stream (one of the values "r", "r+", "w", "w+", "a", "a+", "d"; see fopen() for details) must be compatible with the mode the path number was opened with. The file position indicator of the new stream is set to that belonging to fd, and the error and end-of-file indicators are cleared. The path number is not duplicated, and will be closed when the stream is closed.

Caveats

The mode passed as an argument must be a string, not a character constant. It is a common error to pass a character literal (which is of type **int**) rather than a string containing one character.

Diagnostics

This function may fail, returning NULL if unsuccessful, and sets errno accordingly.

Bugs

This implementation does not currently support the b modifier from the ANSI standard for mode. If used, fdopen will fail.

Conforming To

The function fdopen also exists in POSIX.1, OSK, and OS-9000.

See Also

fopen(), fclose(), system call open()

fileno — Get system path number from stream

Synopsis

```
#include <stdio.h>

int fileno (stream)
FILE *stream;
```

Description

`fileno` returns the OS-9 path number used to implement `stream`. The path number is still owned by `stream`, and will be closed when `fclose()` is called on it. If it's necessary for the path number to survive the closing of `stream`, or if passing the path number to code that may potentially close it, duplicate the path number first with dup — duplicate an open path number.

Caveats

This function is implemented as a macro in `<stdio.h>`, so its name may not be redeclared.

Conforming To

The function `fileno` also exists in POSIX.1, OSK, and OS-9000.

See Also

System call `open()`

findstr, findnstr — Substring searches

Synopsis

```
#include <strings.h>

int findstr (pos, haystack, needle)
int pos;
char *haystack, *needle;

int findnstr (pos, haystack, needle, count)
int pos, count;
char *haystack, *needle;
```

Description

These functions search the string pointed to by `haystack` for the first instance of the string pointed to by `needle`, starting at position `pos`.

The returned value is the position, within `haystack`, of the first matched character in needle, or zero if a match is not found. `findstr` stops searching when a null byte is encountered in `haystack`.

`findnstr` is similar, except it only stops searching at position `pos + count`, so it may continue past null bytes.

Notes

Because the returned value is suitable for use as an argument, these functions may be repeated to find the same substring in multiple locations within a string, or different strings without backtracking. When `findnstr` is used in this manner, `count` should be adjusted to prevent seeking past the end of the buffer.

Caveats

The first position is 1, not zero.

The current implementation does not use the most efficient algorithm for pattern matching so that use on very long strings is likely to be somewhat slower than it might be.

See Also

`strchr()`, `strpfx()`, `strrchr()`

getw, putw — Input and output of words (ints)

Synopsis

```
#include <stdio.h>

int getw (stream)
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

Description

`getw` reads a word from `stream`. That is, it reads the next two bytes as an **`int`**, assuming they are in the correct order for the host.

`putw` writes the integer value `w` to `stream` in the host's natural byte order.

Return Value

On success, `getw` returns the value read, and `putw` returns 0. On error, both return `EOF`.

Bugs

The value returned for an error condition is also a legitimate data value. It's possible to use `ferror()` to distinguish between a correct read of the value of `EOF` and an error.

Conforming To

These functions also exist in the SVID, OSK, and OS-9000.

See Also

`ferror()`, `fread()`, `fwrite()`, `getc()`, `fputc()`

ibrk — Request internal memory

Synopsis

```
#include <stdlib.h>

char *ibrk (size)
size_t size;
```

Description

This function requests a pointer from within the program's initial memory reservation. At program start, this *internal memory* is fixed in size by the system based on the program module's header as well as by the calling program (usually the Shell). Within this region the stack pointer starts at the top, and called functions temporarily allocate their local variables in the stack space. In this way, the stack grows down.

Similarly, the program's global variables are located at the bottom of the internal memory, and limit how much of it may be used by the stack.

ibrk returns a pointer to a block of memory of `size` bytes. The memory from which `ibrk` grants requests is the region between the end of the current maximum data allocation, and the current location of the stack.

If the requested size would cause the data allocation to cross the stack pointer, the request fails. Use `freemem()` to find the current amount of unused internal memory.

Return Value

On success, `ibrk` returns a pointer to a buffer of `size` bytes, or `(char *)-1` if not enough internal memory is available.

Notes

See the Memory Management section of the C compiler manual for a more complete explanation of how memory is laid out in OS-9 C programs.

`malloc()` does not use internal memory.

Caveats

Be careful to leave space for the stack when using `ibrk` calls. When the program is compiled with stack checking enabled, the program aborts if calling a function would cause the stack to overwrite variables, with the error message: `*** Stack Overflow ***`. Without stack checking, there is no protection from such a "stack smash," but stack checks are often disabled for performance reasons.

Conforming To

The `ibrk` function also exists in OSK and OS-9000.

See Also

`malloc()`, system calls `brk()`, `sbrk()`

inv, sqr — Get the inverse of or square a number

Synopsis

```
#include <math.h>

double inv (x)
double x;

double sqr (x)
double x;
```

Link with `-t`.

Description

The `inv` function divides 1.0 by the number `x`, returning the result.

`sqr` multiplies the number `x` by itself, returning the result.

Because these functions use assembly algorithms that are faster (but larger) than the code the compiler emits for the equivalent expressions, in situations where speed is more important than size using `inv` or `sqr` may be a good choice.

See Also

`sqrt()`

l3tol, ltol3 — convert between 3-byte and long integers

Synopsis

```
#include <stdlib.h>

l3tol (lp, cp, n)
long *lp;
char *cp;

ltol3 (cp, lp, n)
char *cp;
long *lp;
```

Description

`l3tol` converts a list of `n` three-byte integers packed into a byte string pointed to by `cp`, into a list of long integers pointed to by `lp`.

`ltol3` does the opposite, converting from long integers (`lp`) to three-byte integers (`cp`).

These functions are especially useful for native file system access under OS-9, where each sector is specified using a 3 byte logical sector number. Converting a 3-byte disk address to a **`long int`** allows regular C arithmetic to be performed on it.

Conforming To

These functions also exist on the PDP-11 versions of Unix.

min, max, umin, umax — Compare integers

Synopsis

```
#include <stdlib.h>

int min (i1, i2)
int i1, i2;

int max (i1, i2)
int i1, i2;

unsigned umin (i1, i2)
unsigned u1, u2;

unsigned umax (i1, i2)
unsigned u1, u2;
```

Description

The `min` and `umin` functions return either `i1` or `i2`, whichever has the lower value. The `max` and `umax` functions are similar, except they return the larger value. The two sets of functions differ only in the signedness of the comparison — `min` and `max` compare within the range ⟦−32768..32767⟧, while `umin` and `umax` compare within the range ⟦0..65535⟧.

mktemp — Create unique temporary filename

Synopsis

```
#include <stdlib.h>

char *mktemp (template)
char *template;
```

Description

The mktemp function generates a unique temporary filename from a buffer template containing a string. The last five (or more) characters of template must be XXXXX; these are replaced with a string that makes the filename (trivially) unique.

Notes

Since it will be modified, template must *not* be a string constant, and should be declared as a character array.

Bugs

mktemp is not a secure method of creating a temporary file.

The current method of generating a name is to replace the trailing XXXXX with the program's task ID number, which is easily predictable. For example, if the contents of template start out as foo.XXXXX, and the program's task ID is 351, then template will be rewritten to contain foo.351.

Because mktemp does not attempt to create or open the file, there exists a race condition between testing whether the temporary filename actually exists and the creation of the file.

Conforming To

The function mktemp also exists in BSD Unix, POSIX.1, OSK, and OS-9000.

See Also

System call getpid — get the task id

sleep — Sleep for a specified number of seconds

Synopsis

```
#include <unistd.h>
```

OR

```
#include <signal.h>

unsigned sleep (seconds)
unsigned seconds;
```

Description

`sleep` causes the calling process to sleep either until the number of `seconds` has elapsed, or until a signal arrives which is not currently being ignored.

If `seconds` is zero, the task will sleep for one tick of the system clock, which may differ between systems.

Return Value

`sleep` returns zero if the requested time has elapsed. If the requested time has not elapsed, the remaining number of system clock ticks is returned.

Notes

This function uses the system call `tsleep()`, and does not alter the return value. Because of this, the number returned is not in seconds (as it is on Unix), but in system ticks.

- On most machines running (Nitr)OS-9 Level One, there are 10 ticks each second.
- On most machines running (Nitr)OS-9 Level Two, there are 100 ticks per second.
- A big exception to this rule of thumb is the Tandy Color Computer, on which there are either 50 or 60 ticks per second (depending on the hardware mode).

Conforming To

The function `sleep` also exists in POSIX.1, OSK, and OS-9000.

See Also

System call `tsleep()`

stacksize, freemem — get stack reservation info

Synopsis

```
size_t stacksize();
size_t freemem();
```

Description

For a description of the meaning and use of this call, the user is referred to the Memory Management section of the C compiler manual.

If the stack checking code is enabled, a call to `stacksize` returns the maximum number of bytes of stack used at the time of the call. This call can be used to determine the stack size required by a program.

`freemem` returns the number of bytes of the stack that has not been used.

See Also

`ibrk()`, System call `sbrk()`, Global variable `memend` and constant `end`.

_strass — byte by byte copy

Synopsis

_strass(char *s1, char *s2, int count);

Description

Until such time as the compiler can deal with structure assignment, this function is useful for copying one structure to another.

”Count” bytes are copied from memory location at ”s2” to memory as ”s1” regardless of the contents.

strclr — Clear a string

Synopsis

```
#include <strings.h>

char *strclr (s, n)
char *s;
int n;
```

Description

The `strclr` function writes `n` null bytes (`'\0'`) into the buffer pointed to by `s`. No attempt is made to verify the buffer is large enough to hold `n` bytes.

See Also

`memcpy()`

strpfx — Check string prefix

Synopsis

```
#include <strings.h>

int strpfx (s, prefix)
char *s, *prefix;
```

Description

The `strpfx` function returns nonzero if the string `prefix` is a prefix of `s`, zero otherwise.

Notes

The empty string, `""`, is a prefix of every string (including itself).

strucmp, strnucmp, strucpy, strucat — Uppercase string functions

strhcpy — Copy an OS-9 string

Synopsis

```
#include <strings.h>

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

Description

The `strhcpy` function copies characters from `src` into the buffer pointed at by `dest`, until encountering a character that has the high bit set. That character is copied without its high bit, and a terminating null byte (`'\0'`) is appended.

This function is useful for dealing with strings returned by OS-9 system calls, the last characters of which are ORed with the value 128 to make them negative.

See Also

`strcpy()`

3
OS-9 System Calls

This section of the C compiler manual is a guide to the system calls available from C programs.

It is not intended as a definitive description of OS-9 service requests as these are described in the OS-9 System Programmer's Manual. However, for most calls, enough information is available here to enable the programmer to write systems calls into programs without looking further.

The names used for the system calls are chosen so that programs transported from other machines or operating systems should compile and run with as little modification as possible. However, care should be taken as the parameters and returned values of some calls may not be compatible with those on other systems. Programmers that are already familiar with OS-9 names and values should take particular care. Some calls do not share the same names as the OS-9 assembly language equivalents. The assembly language equivalent call is shown, where there is one, on the relevant page of the C call description, and a cross-reference list is provided for those already familiar with OS-9 calls.

The normal error signal on return from a system call is a returned value of -1. The relevant error will be found in the predefined `int errno`. `errno` always contains the error from the last erroneous system call. Definitions for the errors for inclusion in the programs are in `<errno.h>`.

In the "See Also" sections on the following pages, unless otherwise stated, the references are to other system calls.

Where header files are shown, it is not mandatory to include them, but it might be convenient to use the manifest constants defined in them rather than integers; it certainly makes for more readable programs.

Stub-implemented system calls

For compatibility reasons, it is sometimes useful to provide "system calls" that do nothing. These are "stub" implementations.

lock On Unix, the `lock` system call prevents the current process from being swapped out (evicted from memory while asleep). OS-9 does not swap processes out, so this function is not needed.

sync On Unix, the `sync` system call causes all uncommitted (cached) writes to be committed to disk, synchronizing the state of the file system. In order to maintain predictable real-time behavior, OS-9 does not cache disk writes, so this function has no use.

access — Check user's permissions for a file

Synopsis

```
#include <modes.h>
```

OR

```
#include <unistd.h>

int access (pathname, mode)
char *pathname;
int mode;
```

Description

The `access` function checks whether the calling process can access `pathname` using file access mode `mode`. The `mode` specifies the accessibility check(s) to be performed. Its value may be any OS-9 file mode as used by `open()` or `creat()`, or it may be zero (which tests for mere existence).

The OS-9 header `<modes.h>` defines the following symbolic constants for `mode` which may be combined using the bitwise OR operator:

```
#define FAM_READ        (0001)  /* may be opened for reading */
#define FAM_WRITE       (0002)  /* may be opened for writing */
#define FAM_EXEC        (0004)  /* executable (or searchable, if dir) */
#define FAM_NONSHARE    (0100)  /* open exclusively */
#define FAM_DIR         (0200)  /* directory mode */
```

When using the UNIX header `<unistd.h>`, the following symbolic constants (which may *also* be combined) are defined:

```
#define F_OK    (0)     /* Path exists */
#define R_OK    (1)     /* Path is readable */
#define W_OK    (2)     /* Path is writable */
#define X_OK    (4)     /* Path is executable / searchable */
```

Return Value

This function returns 0 on success, and -1 on failure. In case of failure, `errno` is set.

Caveats

The set of numeric values for `mode` are not directly compatible with other systems. As such, be sure to use the appropriate symbolic constants for code intended to be portable, and closely examine calls to `access` made by foreign code.

brk, sbrk — change data area size

Synopsis

```
#include <unistd.h>

int brk (addr)
char *addr;

char *sbrk (increment)
intptr_t increment;
```

Assembler Equivalent

```
os9 F$Mem
```

Description

These system calls request changes in the location of the *program break*, which defines the end of the program's data area. Increasing the program break allocates memory to the program, while decreasing the break releases memory back to the operating system. OS-9 does not allow a program to have a split data area, so if it's not possible to add memory after the current program break, then it is not possible to increase the size of the data area at all.

These system calls are quite low level; in most cases, it is better to use the `malloc()` family of library functions, since they are portable, easy to use, and in most cases safer.

brk requests that the highest page of the program's data area should include the address `addr`, allocating or freeing memory as necessary.

sbrk increases the program's data area by `increment` bytes.

Return Value

On success, `brk` returns zero, and `sbrk` returns the previous value of the program break, which becomes a pointer the base of a buffer `increment` bytes in size.

On error, both functions return `(char *)-1` and `errno` is set.

Notes

Calling `sbrk` with an `increment` of zero returns the current location of the program break; however, the global variable `memend` (though not specified in Unix) contains this same value.

On Level One systems, these calls may fail even when the system has free memory. Another program may have requested memory in a location adjacent to the program, or that prevents the program's data area from expanding enough to satisfy the request. Level Two systems do not have this restriction, since each process is granted its own independent address space.

See the Memory Management section of the C compiler manual for a more complete explanation of how memory is laid out in OS-9 C programs.

Conforming To

The `brk` and `sbrk` system calls also exist in Unix-like operating systems.

See Also

`ibrk()`, `malloc()`

chain — load and execute a new program

Synopsis

```
#include <process.h>

chain (name, argsz, args, type, lang, datsz)
char *name, *args;
size_t argsz, datsz;
int type, lang;
```

Assembler Equivalent

```
os9 F$Chain
```

Description

The action of `F$Chain` is described fully in the OS-9 documentation. The `chain` function implements the service request as described, but with one important exception: `chain` will **never** return to the caller. If there is an error, the process will abort and return to its parent process. It might be wise, therefore, for the program to check the existence and access permissions of the module before calling `chain`. Permissions may be checked by using `modlink()` or modload — return a pointer to a module structure followed by a call to munlink — unlink a module.

`name` should point to a string containing the name of the desired program module. `argsz` is the length of the parameter string `args`, which is terminated by a newline (`'\n'`). The module's type `type` is as found in the module header (normally 1, for a program module), and the language `lang` should match the language nybble in the module header (C programs have 1 for "6809 Machine Code" here). The data size `datsz` may be zero, or it may contain the number of 256 byte pages to give to the new process as its initial memory segment.

Caveats

This function never returns, and has no diagnostic capability.

See Also

`os9fork()`

chdir, chxdir — change directory

Synopsis

chdir(char *dirname);
chxdir(char *dirname);

Assembler Equivalent

```
os9 F$ChgDir
```

Description

These calls change the current data directory and the current execution directory, respectively, for the running task. "Dirname" is a pointer to a string that gives a pathname for a directory.

Diagnostics

Each call returns 0 after a successful call, or -1 if "dirname" is not a directory path name, or it is not searchable.

See Also

OS-9 Shell commands `Chd` and `Chx`.

chmod — change access permissions of a file

Synopsis

```
#include <modes.h>
```

chmod(char *fname, int perm);

Description

Chmod changes the permission bits associated with a file. ”Fname” must be a pointer to a file name, and ”perm” should contain the desired bit pattern,

The allowable bit patterns are defined in the include file as follows:

```
/* permissions */
#define  S_IREAD   0x01     /* owner read */
#define  S_IWRITE  0x02     /* owner write */
#define  S_IEXEC   0x04     /* owner execute */
#define  S_IOREAD  0x08     /* public read */
#define  S_IOWRITE 0x10     /* public write */
#define  S_IOEXEC  0x20     /* public execute */
#define  S_ISHARE  0x40     /* sharable */
#define  S_IFDIR   0x80     /* directory */
```

Only the owner or the super user may change the permissions of a file.

Diagnostics

A successful call returns 0. A -1 is returned if the caller is not entitled to change permissions of ”fname” cannot be found.

See Also

OS-9 command `Attr`

chown — change the ownership of a file

Synopsis

chown(char *fname, int ownerid);

Description

This call is available only to the super user. ”Fname” is a pointer to a file name, and ”ownerid” is the new user-id.

Diagnostics

Zero is returned from a successful call. -1 is returned from on error.

close — close a file

Synopsis

close(int pn);

Assembler Equivalent

```
os9 F$Close
```

Description

Close takes a path number, ”pn”, as returned from system calls ”open()”, ”creat()”, or ”dup()”, and closes the associated file.

Termination of a task always closes all open files automatically, but it is necessary to close files where multiple files are opened by the task, and it is desired to re-use path numbers to avoid going over the system or process path number limit.

See Also

`creat()`, `open()`, dup — duplicate an open path number

crc — compute a cyclic redundancy count

Synopsis

crc(char *start, int count, char accum[3]);

Assembler Equivalent

```
os9 F$CRC
```

Description

This call accumulates a crc into a three byte array at "accum" for "count" bytes starting at "start". All three bytes of "accum" should be initialized to 0xff before the first call to "crc()". However, repeated calls can be subsequently made to cover an entire module. If the result is to be used as an OS-9 module crc, it should have its bytes complemented before insertion at the end of the module.

creat — Create a file

Synopsis

```
#include <modes.h>

int creat (pathname, mode)
char *pathname;
mode_t mode;
```

Assembler Equivalent

```
os9 F$Create
```

Description

Creat returns a path number to a new file available for writing, giving it the permissions specified in "perm" and making the task user the owner. If, however, "fname" is the name of an existing file, the file is truncated to zero length, and the ownership and permissions remain unchanged. NOTE, that unlike the OS-9 assembler service request, creat does not return an error if the file already exists. "Access()" should be used to establish the existence of a file if it is important that a file should not be overwritten.

It is unnecessary to specify writing permissions in "perm" in order to write to the file in the current task.

The permissions allowed are defined in the include file as follows:

```
#define  S_IPRM    0xff     /* mask for permission bits */
#define  S_IREAD   0x01     /* owner read */
#define  S_IWRITE  0x02     /* owner write */
#define  S_IEXEC   0x04     /* owner execute */
#define  S_IOREAD  0x08     /* public read */
#define  S_IOWRITE 0x10     /* public write */
#define  S_IOEXEC  0x20     /* public execute */
#define  S_ISHARE  0x40     /* sharable */
```

Directories may not be created with this call; use "mknod()" instead.

Diagnostics

This call returns -1 if there are too many files open. If the pathname cannot be searched, if permission to write is denied, or if the file exists and is a directory.

See Also

Write, Writeln — write to a file or device, close — close a file, chmod — change access permissions of a file

Defdrive — get default system drive

Synopsis

char *defdrive(void);

Description

A call to defdrive returns a pointer to a string containing the name of the default system drive. The method used is to consult the ”Init” module for the default directory name. The name is copied to a static data area and a pointer to it is returned.

Diagnostics

-1 is returned if the ”Init” module cannot be linked to.

dup — duplicate an open path number

Synopsis

dup(int pn);

Assembler Equivalent

```
os9 I$Dup
```

Description

Dup takes the path number, ”pn”, as returned from ”open()” or ”creat()” and returns another path number associated with the same file.

Diagnostics

A -1 is returned is the call fails because there are too many files open or the path nmber is invalid.

See Also

open(), creat(), close — close a file

exit, _exit — Task termination

Synopsis

exit(int status);
_exit(int status);

Assembler Equivalent

```
os9 F$Exit
```

Description

Exit is the normal means of terminating a task. Exit does any cleaning up operations required before terminating, such as flushing out any file buffers (see Standard i/o), but _exit does not.

A task finishing normally, that is returning from ”main()”, is equivalent to a call - ”exit(0)”.

The status passed to exit is available to the parent task if it is executing a ”wait”.

Caveats

In older versions of this implementation, normal termination of a C program with no explicit call to exit, _exit — Task termination unconditionally called `exit(0)` after returning from the `main` function. The current version uses the value returned from `main` as the exit status if no call to exit, _exit — Task termination is made. Note that the C language requires a value to be returned from `main`.

See Also

wait — wait for child process to end

getpid — get the task id

Synopsis

getpid(void);

Assembler Equivalent

```
os9 F$ID
```

Description

A number unique to the current running task is often useful in creating names for temporary files. This call returns the task's system id (as returned to its parent by ”os9fork”).

Description

`os9fork()`, standard library function `mktemp()`.

getstat — get file status

Synopsis

```
#include <sgstat.h>
/* code 0 */
```

getstat(int code, int filenum, char *buffer);

```
/* codes 1 and 6 */
```

getstat(int code, int filenum);

```
/* code 2 */
```

getstat(int code, int filenum, long *size);

```
/* code 5 */
```

getstat(int code, int filenum, long *pos);

Assembler Equivalent

```
os9 I$GetStt
```

Description

A full description of getstat can be found in the OS-9 System Programmer's Manual.

"Code" must be the value of one of the standard codes for the getstat service request. "Filenum" must be the path number of an open file.

The form of the call depends on the value of "code".

Code 0:	"Buffer" must be the address of a 32 byte buffer into which the relevant status packet is copied. The header file has the definitions of the various file and device structures for use by the program.

Code 1:	Code 1 only applies to SCF devices and to test for data available. The return value is zero if there is data available. -1 is returned if there is no data.
Code 2:	”Size” should be the address of a long integer into which the current file size is placed. The return value of the function is -1 on error and 0 on success.
Code 5:	”Pos” should be the address of a long integer into which the current file position is placed. The return value of the function is -1 on error and 0 on success.
Code 6:	Returns -1 on EOF and error and 0 on success.

NOTE that when one of the previous calls returns -1, then actual error is returned in errno.

getuid — return user id

Synopsis

```
uid_t getuid ();
```

Assembler Equivalent

```
os9 F$ID
```

Description

Getuid returns the real user id of the current task (as maintained in the password file).

intercept — set function for interrupt processing

Synopsis

intercept(int (* func) (int));

Assembler Equivalent

```
os9 F$Icpt
```

Description

Intercept instructs OS-9 to pass control to the function "func" when an interrupt (signal) is received by the current process.

If the interrupt processing function has an argument, it will contain the value of the signal received. On return from "func", the process resumes at the point in the program where it was interrupted by the signal. "Interrupt()" is an alternative to the use of "signal()" to process interrupts.

As an example, suppose we wish to ensure that a partially completed output file is deleted if an interrupt is received. The body of the program might include:

```
char *temp_file = "temp"; /* name of temporary file */
int pn=0;                 /* path number */
int intrupt();            /* predeclaration */

...

intercept(intrupt);       /* route interrupt processing */
pn = creat(temp_file,3);  /* make a new file */

...

write(pn,string,count);   /* write string to temp file */

...

close(pn);
pn=0;

...
```

The interrupt routine might be coded:

```
intrupt(sig);
{
     if (pn){ /* only done if pn refers to an open file */
          close(pn);
```

```
            unlink(temp_file); /* delete */
        }
  exit(sig);
  }
```

Caveats

”Intercept()” and ”signal()” are mutually incompatible so that calls to both must not appear in the same program. The linker guards against this by giving an ”entry name clash - _sigint” error if it is attempted.

See Also

signal — catch or ignore interrupts

kill — Send a signal to a process

Synopsis

```
#include <signal.h>

int kill (pid, sig)
pid_t pid;
int sig;
```

Description

Kill sends the interrupt type ”interrupt” to the task with id ”tid”.

Both tasks, sender and receiver, must have the same user id unless the user is the super user.

The include file contains definitions of the defined signals as follows:

```
#define SIGKILL     0   /* system abort (cannot be caught or ignored) */
#define SIGWAKE     1   /* wake up */
#define SIGQUIT     2   /* keyboard abort */
#define SIGINT      3   /* keyboard interrupt */
#define SIGWINCH    4   /* window change */
#define SIGHUP      4   /* hang up */
#define SIGALRM     5   /* alarm signal */
```

Other user-defined signals may, of course, be sent.

Diagnostics

Kill returns 0 from a successful call and -1 if the task does not exist, the effective user ids do not match, or the user is not the system manager.

See Also

intercept — set function for interrupt processing, signal — catch or ignore interrupts, OS-9 Shell command `kill`

lseek — position in file

Synopsis

long lseek(int pn, long position, int type);

Assembler Equivalent

```
os9 I$Seek
```

Description

The read or write pointer for the open file with the path number, ”pn”, is positioned by lseek to the specified place in the file. The ”type” indicates from where ”position” is to be measured: if 0, from the beginning of the file, if 1, from the current location, or if 2, from the end of the file.

Seeking to a location beyond the end of a file open for writing and then writing to it, creates a ”hole” in the file which appears to be filled with zeros from the previous end to the position sought.

The returned value is the resulting position in the file unless there is an error, so to find out the current position use

lseek(pn,0l,1);

Caveats

The argument ”position” must be a long integer. Constants should be explicitly made long by appending an ”l”, as above, and other types should be converted using a cast:

e.g. lseek(pn,(long)pos,1);

Notice also, that the return value from lseek is itself a long integer.

Diagnostics

-1 is returned if ”pn” is a bad path number, or attempting to seek to a position before the beginning of a file.

See Also

`creat()`, `open()`, standard library function `fseek()`

mkdir — Create a directory

Synopsis

#include <modes.h>

mknod(char *fname, int desc);

Assembler Equivalent

```
os9 F$MakDir
```

Description

This call may be used to create a new directory. ”Fname” should point to a string containing the desired name of the directory. ”Desc” is a descriptor specifying the desired mode (file type) and permissions of the new file.

The include file defines the possible values for ”desc” as follows:

```
#define  S_IREAD   0x01     /* owner read */
#define  S_IWRITE  0x02     /* owner write */
#define  S_IEXEC   0x04     /* owner execute */
#define  S_IOREAD  0x08     /* public read */
#define  S_IOWRITE 0x10     /* public write */
#define  S_IOEXEC  0x20     /* public execute */
#define  S_ISHARE  0x40     /* sharable */
```

Diagnostics

Zero is returned if the directory has been successfully made; -1 if the file already exists.

See Also

OS-9 command `MakDir`

modload — return a pointer to a module structure

Synopsis

```
#include <module.h>
```

mod_exec *modlink(char *modname, int type, int language);
mod_exec *modload(char *modname, int type, int language);

Assembler Equivalent

os9 F$LINK

os9 F$LOAD

Description

Each of these calls return a pointer to an OS-9 memory module.

modlink will search the module directory for a module with the same name as ”modname” and, if found, increment its link count.

Modload will open the file which has the path list specified by ”filename” and loads modules from the file adding them to the module directory. The returned value is a pointer to the first module loaded.

Above, each is shown as returning a pointer to an executable module, but it will return a pointer to whatever type of module is found.

Diagnostics

-1 is returned on error.

See Also

munlink — unlink a module

munlink — unlink a module

Synopsis

```
#include <module.h>
```

munlink(mod_exec *mod);

Assembler Equivalent

os9 F$UNLINK

Description

This call informs the system that the module pointed to by ”mod” is no longer required by the current process. Its link count is decremented, and the module is removed from the module directory if the link count reaches zero.

See Also

`modlink()`, modload — return a pointer to a module structure

open — Open a file

Synopsis

```
int open (pathname, flags)
char *pathname;
int flags;
```

Assembler Equivalent

```
os9 I$Open
```

Description

This call opens an existing file for reading if ”mode” is 1, writing if ”mode” is 2, or reading and writing if ”mode” is 3. NOTE that these values are OS-9 specific and not compatible with other systems. ”Fname” should point to a string representing the pathname of the file.

Open returns an integer as ”path number” which should be used by i/o system calls referring to the file.

The position where reads or writes start is at the beginning of the file.

Diagnostics

-1 is returned if the file does not exist, if the pathname cannot be searched, if too many files are already open, or if the file permissions deny the requested mode.

See Also

close — close a file, `creat()`, dup — duplicate an open path number, `read()`, Write, Writeln — write to a file or device

os9fork — create a process

Synopsis

```
int os9fork (name, argsz, args, type, lang, datsz)
char *name, *args;
size_t argsz, datsz;
int type, lang;
```

Assembler Equivalent

```
os9 F$Fork
```

Description

The `os9fork` function creates a child process to run concurrently with the calling process. When this “forked” process terminates, it returns to the calling process.

`name` should point to a string containing name of the desired program module. `argsz` is the length of the parameter string `args`, which is terminated by a newline (`'\n'`). The module’s type `type` is as found in the module header (normally 1, for a program module), and the language `lang` should match the language nybble in the module header (C programs have 1 for 6809 machine code here). The data size `datsz` may be zero, or it may contain the number of 256 byte pages to give to the new process as its initial memory segment.

Notes

Despite its name, `os9fork` does not work in the same way that the Unix `fork()` system call does; it is more similar to the `spawn` function from Microsoft Windows.

The action of `F$Fork` is described fully in the OS-9 System Programmer’s Manual.

Diagnostics

On success, the task ID number of the child process is returned; otherwise, `EOF` is returned and `errno` is set accordingly.

pause — halt and wait for interrupt

Synopsis

pause(void);

Assembler Equivalent

os9 I$SLEEP (with a value of 0)

Description

Pause may be used to halt a task until an interrupt is received from ”kill”.

Pause always returns -1.

See Also

`kill()`, signal — catch or ignore interrupts, OS-9 shell command `kill`

prerr — print error message

Synopsis

prerr(int filnum, int errcode);

Assembler Equivalent

os9 F$PERR

Description

PRERR prints an error message on the output path as specified by ”filnum” which must be the path number of an open file. The message depends on ”errcode” which will normally be a standard OS-9 error code.

read, readln — read from a file

Synopsis

read(int pn, char *buffer, int count);
readln(int pn, char *buffer, int count);

Assembler Equivalent

```
os9 I$Read
os9 I$ReadLn
```

Description

The path number, "pn" is an integer which is one of the standard path numbers 0, 1, or 2, or the path number should have been returned by a successful call to "open", "creat", or "dup". "Buffer" is a pointer to space with at least "count" bytes of memory into which read will put the data from the file.

It is guaranteed that at most "count" bytes will be read, but often less will be, either because, for readln, the file represents a terminal and input stops at the end of a line, or for both, end-of-file has been reached.

readln causes "line-editing" such as echoing to take place and returns once the first "\n" is encountered in the input or the number of bytes requested has been read. Readln is the preferred call for reading from the user's terminal.

Read does not cause any such editing. See the OS-9 manual for a fuller description of the actions of these calls.

Diagnostics

Read and readln return the number of bytes actually read (0 at end-of-file) or -1 for physical i/o errors, a bad path number, or a ridicolous "count".

NOTE that end-of-file is not considered an error, and no error indication is returned. Zero is returned on EOF.

See Also

`open()`, `creat()`, dup — duplicate an open path number

setpr — set process priority

Synopsis
setpr(int pid, int priority);

Assembler Equivalent
os9 F$SPRIOR

Description
SETPR sets the process identified by ”pid” (process id) to have a priority of ”priority”. The lowest level is 0 and the highest is 255.

Diagnostics
The call will return -1 if the process does not have the same user id as the caller.

setime, getime — set and get system time

Synopsis

```
#include <time.h>
```

setime(struct sgtbuf *buffer);
getime(struct sgtbuf *buffer);

Assembler Equivalent

os9 F$STIME

os9 F$GTIME

Description

GETIME returns system time in buffer. SETIME sets system time from buffer.

setuid — set user id

Synopsis

setuid(int uid);

Assembler Equivalent

os9 F$SUSER

Description

This call may be used to set the user id for the current task. Setuid only works if the caller is the super user (user id 0).

Diagnostics

Zero is returned from a successful call, and -1 is returned on error.

See Also

getuid — return user id

setstat — set file status

Synopsis

```
#include <sgstat.h>
/* code 0 */
```

setstat(int code, int filenum, char *buffer);

```
/* code 2 */
```

setstat(int code, int filenum, long size);

Assembler Equivalent

os9 F$SETSTT

Description

For a detailed explanation of this call, see the OS-9 System Programmer's Manual.

"Filenum" must be the path number of a currently open file. The only values for code at this time are 0 and 2. When "code" is 0, "buffer" should be the address of a 32 byte structure which is written to the option section of the path descriptor of the file. The header file contains definitions of various structures maintained by OS-9 for use by the programmer. When code is 2, "size" should be a long integer specifying the new file size.

signal — catch or ignore interrupts

Synopsis

```
#include <signal.h>

typedef int (*sighandler_t)(int);
```

sighandler_t signal(int interrupt, sighandler_t address);

Description

This call is a comprehensive method of catching or ignoring signals sent to the current process. Notice that ”kill()” does the sending of signals, and ”signal()” does the catching.

Normally, a signal sent to a process causes it to terminate with the status of the signal. If, in advance of the anticipated signal, this system call is used, the program has the choice of ignoring the signal or designating a function to be executed when it is received. Different functions may be designated for different signals.

The values for ”address” have the following meanings:

0	reset to the default i.e. abort when received.
1	ignore; this will apply until reset to another value.
Otherwise	taken to be the address of a C function which is to be executed on receipt of the signal.

If the latter case is chosen, when the signal is received by the process the ”address” is reset to 0, the default, before the function is executed. This means that if the next signal received should be caught then another call to ”signal()” should be made immediately. This is normally the first action taken by the ”interrupt” function. The function may access the signal number which caused its execution by looking at its argument. On completion of this function the program resumes at the point at which is was ”interrupted” by the signal.

An example should help to clarify all this. Suppose a program needs to create a temporary file which should be deleted before exiting. The body of the program might contain fragments like this:

```
pn  =  creat("temp",3);          /*  create  a temporary file */
signal(2,intrupt);               /*   ensure   tidying   up   */
signal(3,intrupt);
write(pn,string,count);
close(pn);                       /* finished writing */
unlink("temp");                  /* delete it */
exit(0);                         /* normal exit */
```

The call to ”signal()” will ensure that if a keyboard or quit signal is received then the function ”intrupt()” will be executed and this might be written:

```
intrupt(sig)
{
close(pn);                  /* close it if open */
unlink("temp");             /* delete it */
exit(sig);                  /* received signal er exit status */
}
```

In this case, as the function will be exiting before another signal is received, it is unnecessary to call ”signal()” again to reset its pointer. Note that either the function ”intrupt()” should appear in the source code before the call to ”signal()”, or it should be pre-declared.

The signals used by OS-9 are defined in the header file as follows:

```
/* OS-9 signals */
#define         SIGKILL 0  /* system abort (cannot be caught or ignored)
    */
#define         SIGWAKE 1   /* wake up */
#define         SIGQUIT 2   /* keyboard abort */
#define         SIGINT  3   /* keyboard interrupt */

/* special addresses */
#define         SIG_DFL 0   /* reset to default */
#define         SIG_IGN 1   /* ignore */
```

Please note that there is another method of trapping signals, namely ”intercept()” (q.v.). However, since ”signal()” and ”intercept()” are mutually incompatible, calls to both of them must not appear in the same program. The link-loader will preven the creation of an executable program in which both are called by aborting with an ”entry name clash” error for ”_sigint”.

See Also

intercept — set function for interrupt processing, `kill()`, OS-9 Shell command `kill`

tsleep — put process to sleep

Synopsis

```
#include <signal.h>

unsigned tsleep (ticks)
unsigned ticks;
```

Assembler Equivalent

```
os9 F$Sleep
```

Description

The `tsleep` system call deactivates the calling process until the number of system ticks requested has elapsed, or indefinitely if `ticks` is zero. If `ticks` is 1, the effect will be to yield the rest of the current time slice. Because the program will be awakened if a signal is received, sleeping indefinitely is an excellent way to wait for a signal or interrupt without wasting CPU time.

The length of a system clock tick is system-dependent, but is usually between 10 and 100 milliseconds:

- On most Level One systems, a clock tick is 100 milliseconds (1/10 of a second).
- On most Level Two systems, a clock tick is 10 milliseconds (1/100 of a second).
- On Tandy Color Computers, a clock tick is either 16.66… or 20 milliseconds.

Because it is not known when the call to `tsleep` was made, this call cannot be used for precise timekeeping.

A sleep of one tick is effectively a "yield remaining time slice" request; the process will be placed in the active process list (the "run queue"), and will resume when it reaches the front of the queue. A sleep of $N = 2$ or more ticks delays the insertion until $N - 1$ ticks have elapsed, and then the program resumes when it reaches the front of the run queue.

Caveats

The system clock must be running to perform a timed sleep; it is not required to perform an indefinite sleep, or to give up a time slice.

See Also

`kill()`, wait — wait for child process to end, library function `sleep()`

Unlink — remove directory entry

Synopsis

unlink(char *fname);

Assembler Equivalent

os9 I$DELETE

Description

Unlink deletes the directory entry whose name is pointed to by ”fname”. If the entry was the last link to the file, the file itself is deleted and the disc space occupied made available for re-use. If, however the file is open, in any active task, the deletion of the actual file is delayed until the file is closed.

Diagnostics

Zero is returned from a successful call, -1 if the file does not exist, if its directory is write-protected, or cannot be searched, if the file is a non-empty directory or a device.

See Also

OS-9 command `Del`

wait — wait for child process to end

Synopsis

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

OR

```
#include <process.h>

pid_t wait (wstatus)
int *wstatus;
```

Assembler Equivalent

```
os9 F$Wait
```

Description

The `wait` system call halts the calling task until a child task has terminated.

If a child process has already terminated, `wait` returns immediately. Otherwise, it blocks until a process terminates, or a signal handler interrupts the call. If `wstatus` is not `NULL`, `wait` stores status information in the **`int`** to which it points.

After `wait` returns, the integer pointed to by `wstatus` contains the child process's exit status. This is usually the argument of the exit, _exit — Task termination system call, or the signal number it received if it was killed by a signal.

Return Value

`wait` returns the process ID of the task that exited, or -1 if the current process has no children.

Notes

On POSIX systems, `wstatus` may be inspected using several macros to determine the way the child terminated; however, OS-9 does not supply information enabling the C library to distinguish an exit due to signal from a normal exit. So, for compatibility with Unix, several macros are defined in `<sys/wait.h>`:

`WIFEXITED(wstatus)` always returns true.

`WIFSIGNALED(wstatus)` always returns false.

`WIFSTOPPED(wstatus)` always returns false.

`WEXITSTATUS(wstatus)` returns `wstatus`, the exit status of the child.

`WSTOPSIG(wstatus)` returns `wstatus`, but since `WIFSTOPPED()` always returns false, should never be called.

`WTERMSIG(wstatus)` returns `wstatus`, but since `WIFSIGNALED()` always returns false, should never be called.

Caveats

A `wait` call must be made for each child task the program spawns; until a terminated child process's exit status has been collected with `wait`, it remains in the process table as a "zombie" (a process in the `DEAD` state). Too many of these "unreaped" child processes may even prevent new processes from being created.

The values of OS-9 status codes are not necessarily compatible with those of other systems.

See Also

exit, _exit — Task termination, `os9fork()`, intercept — set function for interrupt processing, signal — catch or ignore interrupts

Write, Writeln — write to a file or device

Synopsis

write(int pn, char *buffer, int count);
writeln(int pn, char *buffer, int count);

Assembler Equivalent

os9 I$WRITE

os9 I$WRITLN

Description

"Pn" must be a value returned by "open", "creat" or "dup" or should be a 0(stdin), 1(stdout), or 2(stderr).

"Buffer should point to an area of memory from which "count" bytes are to be written. Write returns the actual number of bytes written, and if this is different from "count", an error has occurred.

Writes in multiples of 256 bytes to file offset boundaries of 256 bytes are the most efficient.

Write causes no "line-editing" to occur on output. Writeln causes line-editing and only writes up to the first "\n" in the buffer if this is found before "count" is exhausted. For a full description of the actions of these calls the reader is referred to the OS-9 documentation.

Diagnostics

-1 is returned if "pn" is a bad path number, of "count" is ridiculous or on physical i/o error.

See Also

`creat()`, `open()`

_os9 — system call interface from C programs

Synopsis

```
#include <os9.h>

int _os9 (code, regs)
char code;
struct registers *regs;
```

Description

The `_os9` function calls the OS-9 system call `code` directly, without using assembly language or a helper function. `code` may be any value defined for the current system; the header file `<funcs.h>` (which is included by `<os9.h>`) contains a list of all codes known at this time[1].

The input registers (`regs`) for system calls are accessed using the following structure, defined in <os9.h>:

```
struct registers {
    char        rg_cc,  /* condition codes */
                rg_a,   /* 6809 accumulator A */
                rg_b,   /* 6809 accumulator B */
                rg_dp;  /* 6809 direct page register */
    unsigned    rg_x,   /* 6809 index register X */
                rg_y,   /* 6809 index register Y */
                rg_u;   /* 6809 combo register U */
};
```

Diagnostics

-1 is returned if the OS-9 call failed. 0 is returned on success.

Program Example

```
#include <os9.h>
#include <modes.h>

/* this program does an I$GetStt call to get file size */
int
main (argc, argv)
int argc;
char **argv;
{
    struct registers r;
    int path;
```

[1] Your machine may not implement all system calls, or may support calls not known to the compiler

```
    /* tell linker we need to print longs */
    pflinit();

    /* low level open (file name is first command line param) */
    path = open (*++argv, S_IREAD);

    /* set up regs for call to OS-9 */
    r.rg_a = path;
    r.rg_b = SS_SIZE;

    if (_os9 (I_GETSTT, &r) == 0) { /* success */
        printf ("filesize = %lx\n", (long)(r.rg_x << 16) + r.rg_u);
    } else {     /* failed */
        printf ("OS9 error #%d\n", r.rg_b & 0xFF);
    }

    dumpregs (&r);  /* take a look at the registers */
}

dumpregs (r)
register struct registers *r;
{
    printf ("cc=%02x\n", r->rg_cc & 0xff);
    printf (" a=%02x\n", r->rg_a & 0xff);
    printf (" b=%02x\n", r->rg_b & 0xff);
    printf ("dp=%02x\n", r->rg_dp & 0xff);
    printf (" x=%02x\n", r->rg_x);
    printf (" y=%02x\n", r->rg_u);
    printf (" u=%02x\n", r->rg_y);
}
```