NAME

libsa - support library for standalone executables

SYNOPSIS

#include <stand.h>

DESCRIPTION

The **libsa** library provides a set of supporting functions for standalone applications, mimicking where possible the standard BSD programming environment. The following sections group these functions by kind. Unless specifically described here, see the corresponding section 3 manpages for the given functions.

STRING FUNCTIONS

String functions are available as documented in string(3) and bstring(3).

MEMORY ALLOCATION

*void ** **malloc**(*size_t size*)

Allocate *size* bytes of memory from the heap using a best-fit algorithm.

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

Free the allocated object at *ptr*.

void **setheap**(*void *start, void *limit*)

Initialise the heap. This function must be called before calling **alloc**() for the first time. The region between *start* and *limit* will be used for the heap; attempting to allocate beyond this will result in a panic.

*char ** **sbrk**(*int junk*)

Provides the behaviour of **sbrk**(*0*), i.e., returns the highest point that the heap has reached. This value can be used during testing to determine the actual heap usage. The *junk* argument is ignored.

ENVIRONMENT

A set of functions are provided for manipulating a flat variable space similar to the traditional shell-supported environment. Major enhancements are support for set/unset hook functions.

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

int **setenv**(*const char *name, const char *value, int overwrite*)

int **putenv**(*char *string*)

int **unsetenv**(*const char *name*)

These functions behave similarly to their standard library counterparts.

*struct env_var ** **env_getenv**(*const char *name*)

Looks up a variable in the environment and returns its entire data structure.

int **env_setenv**(*const char *name, int flags, const void *value, ev_sethook_t sethook, ev_unsethook_t unsethook*)

Creates a new or sets an existing environment variable called *name*. If creating a new variable, the *sethook* and *unsethook* arguments may be specified.

The set hook is invoked whenever an attempt is made to set the variable, unless the EV_NOHOOK flag is set. Typically a set hook will validate the *value* argument, and then call **env_setenv**() again with EV_NOHOOK set to actually save the value. The predefined function **env_noset**() may be specified to refuse all attempts to set a variable.

The unset hook is invoked when an attempt is made to unset a variable. If it returns zero, the variable will be unset. The predefined function *env_nounset* may be used to prevent a variable being unset.

STANDARD LIBRARY SUPPORT

int **abs**(*int i*)

int **getopt**(*int argc, char * const *argv, const char *optstring*)

long **strtol**(*const char *nptr, char **endptr, int base*)

long long **strtoll**(*const char *nptr, char **endptr, int base*)

long **strtoul**(*const char *nptr, char **endptr, int base*)

long long **strtoull**(*const char *nptr, char **endptr, int base*)

void **srandom**(*unsigned int seed*)

long **random**(*void*)

*char ** **strerror**(*int error*)

Returns error messages for the subset of errno values supported by **libsa**.

assert(*expression*)

Requires <*assert.h*>.

int **setjmp**(*jmp_buf env*)

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

Defined as **_setjmp**() and **_longjmp**() respectively as there is no signal state to manipulate. Requires <*setjmp.h*>.

CHARACTER I/O

void **gets**(*char *buf*)

Read characters from the console into *buf*. All of the standard cautions apply to this function.

void **ngets**(*char *buf, int size*)

Read at most *size* - 1 characters from the console into *buf*. If *size* is less than 1, the function's behaviour is as for **gets**().

int **fgetstr**(*char *buf, int size, int fd*)

Read a line of at most *size* characters into *buf*. Line terminating characters are stripped, and the buffer is always NUL terminated. Returns the number of characters in *buf* if successful, or -1 if a read error occurs.

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

void **vprintf**(*const char *fmt, va_list ap*)

int **sprintf**(*char *buf, const char *fmt, ...*)

void **vsprintf**(*char *buf, const char *fmt, va_list ap*)

The *printf functions implement a subset of the standard **printf**() family functionality and some extensions. The following standard conversions are supported: c,d,n,o,p,s,u,x. The following modifiers are supported: +,-,#,*,0,field width,precision,l.

The b conversion is provided to decode error registers. Its usage is:

printf("reg=%b\n", regval, "<base><arg>*");

where <base> is the output expressed as a control character, e.g. \10 gives octal, \20 gives hex. Each <arg> is a sequence of characters, the first of which gives the bit number to be inspected (origin 1) and the next characters (up to a character less than 32) give the text to be displayed if the bit is set. Thus

printf("reg=%b\n", 3, "\10\2BITTWO\1BITONE");

would give the output

reg=3<BITTWO,BITONE>

The D conversion provides a hexdump facility, e.g.

printf("%6D", ptr, ":"); gives "XX:XX:XX:XX:XX:XX"

printf("%*D", len, ptr, " "); gives "XX XX XX ..."

CHARACTER TESTS AND CONVERSIONS

int **isupper**(*int c*)

int **islower**(*int c*)

int **isspace**(*int c*)

int **isdigit**(*int c*)

int **isxdigit**(*int c*)

int **isascii**(*int c*)

int **isalpha**(*int c*)

int **isalnum**(*int c*)

int **iscntrl**(*int c*)

int **isgraph**(*int c*)

int **ispunct**(*int c*)

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

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

FILE I/O

int **open**(*const char *path, int flags*)

Similar to the behaviour as specified in open(2), except that file creation is not supported, so the mode parameter is not required. The *flags* argument may be one of O_RDONLY, O_WRONLY and O_RDWR. Only UFS currently supports writing.

int **close**(*int fd*)

void **closeall**(*void*)

Close all open files.

ssize_t **read**(*int fd, void *buf, size_t len*)

ssize_t **write**(*int fd, void *buf, size_t len*)

(No file systems currently support writing.)

off_t **lseek**(*int fd, off_t offset, int whence*)

Files being automatically uncompressed during reading cannot seek backwards from the

current point.

int **stat**(*const char *path, struct stat *sb*)

int **fstat**(*int fd, struct stat *sb*)

The **stat**() and **fstat**() functions only fill out the following fields in the *sb* structure: st_mode,st_nlink,st_uid,st_gid,st_size. The **tftp** file system cannot provide meaningful values for this call, and the **cd9660** file system always reports files having uid/gid of zero.

PAGER

The **libsa** library supplies a simple internal pager to ease reading the output of large commands.

void **pager_open**()

Initialises the pager and tells it that the next line output will be the top of the display. The environment variable LINES is consulted to determine the number of lines to be displayed before pausing.

void **pager_close**(*void*)

Closes the pager.

int **pager_output**(*const char *lines*)

Sends the lines in the NUL-terminated buffer at *lines* to the pager. Newline characters are counted in order to determine the number of lines being output (wrapped lines are not accounted for). The **pager_output**() function will return zero when all of the lines have been output, or nonzero if the display was paused and the user elected to quit.

int **pager_file**(*const char *fname*)

Attempts to open and display the file *fname*. Returns -1 on error, 0 at EOF, or 1 if the user elects to quit while reading.

MISC

char * **devformat**(*struct devdesc* *)

Format the specified device as a string.

int **devparse**(*struct devdesc **dev, const char *devdesc, const char **path*)

Parse the devdesc string of the form ‘device:[/path/to/file]’. The devsw table is used to match the start of the ‘device’ string with *dv_name*. If *dv_parsedev* is non-NULL, then it will be called to parse the rest of the string and allocate the struct devdesc for this path. If NULL, then a default routine will be called that will allocate a simple struct devdesc, parse a unit number and ensure there’s no trailing characters. If path is non-NULL, then a pointer to the remainder of the devdesc string after the device specification is written.

int **devinit**(*void*) Calls all the *dv_init* routines in the devsw array, returning the number of routines that returned an error.

void **twiddle**(*void*)

Successive calls emit the characters in the sequence |,/,-,\ followed by a backspace in order to provide reassurance to the user.

REQUIRED LOW-LEVEL SUPPORT

The following resources are consumed by **libsa** - stack, heap, console and devices.

The stack must be established before **libsa** functions can be invoked. Stack requirements vary depending on the functions and file systems used by the consumer and the support layer functions detailed below.

The heap must be established before calling **alloc**() or **open**() by calling **setheap**(). Heap usage will vary depending on the number of simultaneously open files, as well as client behaviour. Automatic decompression will allocate more than 64K of data per open file.

Console access is performed via the **getchar**(), **putchar**() and **ischar**() functions detailed below.

Device access is initiated via **devopen**() and is performed through the **dv_strategy**(), **dv_ioctl**() and **dv_close**() functions in the device switch structure that **devopen**() returns.

The consumer must provide the following support functions:

int **getchar**(*void*)

Return a character from the console, used by **gets**(), **ngets**() and pager functions.

int **ischar**(*void*)

Returns nonzero if a character is waiting from the console.

void **putchar**(*int*)

Write a character to the console, used by **gets**(), **ngets**(), ***printf**(), **panic**() and **twiddle**() and thus by many other functions for debugging and informational output.

int **devopen**(*struct open_file *of, const char *name, const char **file*)

Open the appropriate device for the file named in *name*, returning in *file* a pointer to the remaining body of *name* which does not refer to the device. The *f_dev* field in *of* will be set to point to the *devsw* structure for the opened device if successful. Device identifiers must always precede the path component, but may otherwise be arbitrarily formatted. Used by **open**() and thus for all device-related I/O.

int **devclose**(*struct open_file *of*)

Close the device allocated for *of*. The device driver itself will already have been called for the close; this call should clean up any allocation made by devopen only.

void **__abort**()

Calls **panic**() with a fixed string.

void **panic**(*const char *msg, ...*)

Signal a fatal and unrecoverable error condition. The *msg* ... arguments are as for **printf**().

INTERNAL FILE SYSTEMS

Internal file systems are enabled by the consumer exporting the array *struct fs_ops *file_system[]*, which should be initialised with pointers to *struct fs_ops* structures. The following file system handlers are supplied by **libsa**, the consumer may supply other file systems of their own:

ufs_fsops The BSD UFS.

ext2fs_fsops Linux ext2fs file system.

tftp_fsops File access via TFTP.

nfs_fsops File access via NFS.

cd9660_fsops ISO 9660 (CD-ROM) file system.

gzipfs_fsops Stacked file system supporting gzipped files. When trying the gzipfs file system, **libsa** appends .gz to the end of the filename, and then tries to locate the file using the other file systems. Placement of this file system in the *file_system[]* array determines whether gzipped files will be opened in preference to non-gzipped files. It is only possible to seek a gzipped file forwards, and **stat**() and **fstat**() on gzipped files will report an invalid length.

bzipfs_fsops The same as *gzipfs_fsops*, but for bzip2(1)-compressed files.

The array of *struct fs_ops* pointers should be terminated with a NULL.

DEVICES

Devices are exported by the supporting code via the array *struct devsw *devsw[]* which is a NULL terminated array of pointers to device switch structures.

DRIVER INTERFACE

The driver needs to provide a common set of entry points that are used by **libsa** to interface with the device.

```
struct devsw {
    const char      dv_name[DEV_NAMLEN];
    int             dv_type;
    int             (*dv_init)(void);
    int             (*dv_strategy)(void *devdata, int rw, daddr_t blk,
                                   size_t size, char *buf, size_t *rsize);
    int             (*dv_open)(struct open_file *f, ...);
    int             (*dv_close)(struct open_file *f);
    int             (*dv_ioctl)(struct open_file *f, u_long cmd, void *data);
    int             (*dv_print)(int verbose);
    void    (*dv_cleanup)(void);
    char *  (*dv_fmtdev)(struct devdesc *);
    int             (*dv_parsedev)(struct devdesc **dev, const char *devpart,
                                   const char **path);
    bool    (*dv_match)(struct devsw *dv, const char *devspec);
};
```

dv_name() The device’s name.

dv_type() Type of device. The supported types are:

DEVT_NONE

DEVT_DISK

DEVT_NET

DEVT_CD

DEVT_ZFS

DEVT_FD

Each type may have its own associated (struct type_devdesc), which has the generic (struct devdesc) as its first member.

dv_init() Driver initialization routine. This routine should probe for available units. Drivers are responsible for maintaining lists of units for later enumeration. No other driver routines may be called before **dv_init**() returns.

dv_open() The driver open routine.

dv_close() The driver close routine.

dv_ioctl() The driver ioctl routine.

dv_print() Prints information about the available devices. Information should be presented with **pager_output**().

dv_cleanup()

Cleans up any memory used by the device before the next stage is run.

dv_fmtdev()

Converts the specified devdesc to the canonical string representation for that device.

dv_parsedev()

Parses the device portion of a file path. The devpart will point to the ‘tail’ of device name, possibly followed by a colon and a path within the device. The ‘tail’ is, by convention, the part of the device specification that follows the *dv_name* part of the string. So when *devparse* is parsing the string "disk3p5:/xxx", devpart will point to the ‘3’ in that string.

The parsing routine is expected to allocate a new struct devdesc or subclass and return it in dev when successful. This routine should set path to point to the portion of the string after device specification, or "/xxx" in the earlier example. Generally, code needing to parse a path will use *devparse* instead of calling this routine directly.

dv_match() NULL to specify that all device paths starting with *dv_name* match. Otherwise, this function returns 0 for a match and a non-zero errno to indicate why it didn't match. This is helpful when you claim the device path after using it to query properties on systems that have uniform naming for different types of devices.

HISTORY

The **libsa** library contains contributions from many sources, including:

- **libsa** from NetBSD
- **libc** and **libkern** from FreeBSD 3.0.
- **zalloc** from Matthew Dillon <*dillon@backplane.com*>

The reorganisation and port to FreeBSD 3.0, the environment functions and this manpage were written by Mike Smith <*msmith@FreeBSD.org*>.

BUGS

The lack of detailed memory usage data is unhelpful.