III. C System Interface Library

NAME

Cint - C interface to operating system

FUNCTION

C programs operating in user mode under any operating system may assume the existence of several functions which implement program entry/exit and low-level I/O. This section documents these functions, plus several critical presumptions that can be made about the environment supplied, in the most portable of terms. Details of actual implementations may be found in the various C Interface Manuals; but these are best ignored if portability is considered a virtue.

Each C program must provide a function main(), detailed on a separate manual page, that has access to the command line used to invoke the program. Returning from main, or calling exit(), terminates program execution and reports at most one bit of status, success or failure, to the invoker.

C programs may assume the existence of three open text files: STDIN (file descriptor 0), STDOUT (file descriptor 1), and STDERR (file descriptor 2). The first may be used with read() and close(); the latter two may be used with write() and close().

The "standard input" STDIN and "standard output" STDOUT may be redirected on the command line (transparently to the program); the "standard error" file STDERR is a reliable destination for error messages. The following conventions apply to I/O:

A filename - is a string, hence a NUL terminated array of characters, hence a pointer to char when used as an argument. For maximum portability, a filename should consist of letters, of one case only, and digits. The first character should be a letter and there should be no more than six characters, optionally followed by a '.' and no more than two more letters.

A file descriptor - is a short integer (type FILE in the standard header std.h) that is guaranteed to be non-negative. Its value should be otherwise assumed to be magic.

A mode - is a short integer that specifies reading (mode == 0), writing (mode == 1), or updating (mode == 2). No other values are defined.

A binary file - looks to a C program like a sequence of characters, period. There is no record structure and all character codes are allowed. Trailing NULs may be provided, free of charge, by some operating systems.

A text file - is much like a binary file, except it is assumed to contain printable text that may be mapped between internal and external forms. Most programs deal with such files of printable text, where a line structure is imposed (internally) by the presence of a newline (ASCII linefeed) character at the end of each line. Lines can be assumed never to be longer than 512 characters, counting the terminating newline, nor should a text file ever be produced whose last line has no newline at the end.

Space is reserved for each program to grow a stack, or LIFO list of function call argument lists and automatic storage frames, and a heap, or unstructured data area. Heap is purchased in (not necessarily contiguous) chunks by calls on sbreak(), and is never given back during program execution. Stack and heap often must contend for the same (limited) space, so an otherwise correct C program may terminate early, or (sadly) misbehave, because insufficient space was allotted.

Note that all objects in C are presumed to have non-NULL addresses; the system is obliged never to bind an external identifier to the value zero. The system interface ensures that address zero never occurs on the stack or heap, as well. In fact, the addresses -1 and +1 are also discouraged, since some functions treat these values as codes for discredited pointers, much like NULL (0).

NAME

main - enter a C program

SYNOPSIS

```
BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
```

FUNCTION

main is the function called to initiate a C program; hence every user program must contain a function called main. Its arguments are a sequence of NUL terminated strings, pointed at by the first ac elements of the array av, obtained from the command line used to invoke the programs. By convention, ac is always at least one, av[0] is the name by which the program has been invoked, and av[1], if present, is the first argument string, etc. Program execution is terminated by returning from main, or by an explicit call to exit. In either case, one bit of status is returned to the invoker to signify whether the program ran successfully.

RETURNS

main returns YES (or non-zero) if successful, otherwise NO (zero).

EXAMPLE

```
/*  ECHO ARGUMENTS TO STDOUT
 *  copyright (c) 1980 by Whitesmiths, Ltd.
 */
#include <std.h>

BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    if (1 < ac)
        {
        putstr(STDOUT, *++av, NULL);
        for (--ac, ++av; --ac; ++av)
            putstr(STDOUT, " ", *av, NULL);
        write(STDOUT, "\n", 1);
        }
    return (YES);
    }
```

NAME

_pname - program name

SYNOPSIS

```
TEXT _pname;
```

FUNCTION

_pname is the (NUL terminated) name by which the program was invoked, if that can be determined from the command line, or the name provided by the C programmer, if present, or the name "error", delivered up by a waiting library module. The library definition is used only if no definition of _pname is provided by the C program and/or the compile time name is not overridden at runtime.

It is used primarily for labelling diagnostic printouts.

SEE ALSO

error(II)

NAME
close - close a file

SYNOPSIS
```
FILE close(fd)
    FILE fd;
```

FUNCTION
close closes the file associated with the file descriptor fd, making the fd available for future open or create calls.

RETURNS
close returns the now useless file descriptor, if successful, or a negative number.

EXAMPLE
To copy an arbitrary number of files:

```
while (fd = getfiles(&ac, &av, STDIN, -1))
    {
    while (0 < (n = read(fd, buf, BUFSIZE)))
        write(STDOUT, buf, n);
    close(fd);
    }
```

SEE ALSO
create, open, remove, uname

NAME

create - open an empty instance of a file

SYNOPSIS

```
FILE create(fname, mode, rsize)
    TEXT *fname;
    COUNT mode;
    BYTES rsize;
```

FUNCTION

create makes a new file fname, if it did not previously exist, or truncates the existing file to zero length. If (mode == 0) the file is opened for reading, else if (mode == 1) it is opened for writing, else (mode == 2) of necessity and the file is opened for updating (reading and writing).

If the file is to contain arbitrary binary data, as opposed to printable ASCII text, the record size rsize should be non-zero. Not all systems behave well if a textfile is created for updating.

RETURNS

create returns a file descriptor for the created file or a negative number.

EXAMPLE

```
    if ((fd = create("xeq", WRITE, 1)) < 0)
        write(STDERR, "can't create xeq\n", 17);
```

SEE ALSO

close, open, remove, uname

NAME
exit - terminate program execution

SYNOPSIS
```
VOID exit(success)
    BOOL success;
```

FUNCTION
exit calls all functions registered with onexit, closes all files, and terminates program execution. exit is called with a non-zero (YES) to indicate success, or a zero (NO) to indicate unsuccessful termination; not all systems provide a recipient for this information.

RETURNS
exit will never return to its caller.

EXAMPLE
```
if ((fd = open("file", READ, 0)) < 0)
    {
    write(STDERR, "can't open file\n", 16);
    exit(NO);
    }
```

SEE ALSO
onexit

NAME

lseek - set file read/write pointer

SYNOPSIS

```
    COUNT lseek(fd, offset, sense)
        FILE fd;
        LONG offset;
        COUNT sense;
```

FUNCTION

lseek uses the long offset provided to modify the read/write pointer for the binary file fd, under control of sense. If (sense == 0) the pointer is set to the byte offset, which should be positive. If (sense == 1) the byte offset is algebraically added to the current pointer. Other values of sense are extremely system dependent.

The call lseek(fd, 0L, 1) is guaranteed to leave the file pointer unmodified and, more important, to succeed only if lseek calls are both acceptable and meaningful for the fd specified. Other lseek calls may appear to succeed, but without effect, as when rewinding a terminal.

RETURNS

lseek returns the file descriptor if successful, or a negative number.

EXAMPLE

To read a 512-byte block:

```
        BOOL getblock(buf, blkno)
            TEXT *buf;
            BYTES blkno;
            {

            lseek(STDIN, (LONG)blkno << 9, 0);
            return (read(STDIN, buf, BUFSIZE) != BUFSIZE);
            }
```

NAME
onexit - call function on program exit

SYNOPSIS
```
VOID (*onexit())(pfn)
    VOID (*(*pfn)())();
```

FUNCTION
onexit registers the function pointed at by pfn, to be called on program exit. The function at pfn is obliged to return the pointer returned by the onexit call, so that any previously registered functions can also be called.

RETURNS
onexit returns a pointer to another function; it is guaranteed to be non-NULL.

EXAMPLE
To register the function thisguy:

```
GLOBAL VOID (*(*nextguy)())(), (*thisguy())();

if (!nextguy)
    nextguy = onexit(&thisguy);
```

SEE ALSO
exit

BUGS
The type declarations defy description, and are still wrong.

NAME

onintr - capture interrupts

SYNOPSIS

```
VOID onintr(pfn)
    VOID (*pfn)();
```

FUNCTION

onintr ensures that the function at pfn is called on the occurrence of an interrupt generated from the keyboard of a controlling terminal. (Typing a delete DEL, or sometimes a ctl-C ETX, performs this service on many systems.) Any earlier call to onintr is overriden.

The function is called with one integer argument, whose value is always zero, and must not return; if it does, a message is output to STDERR and an immediate error exit is taken.

If (pfn is NULL) then the interrupt is disabled (turned off), assuming that the system supports such an operation. A disabled interrupt is not, however, turned on by a subsequent call with pfn not NULL. Systems that support nothing resembling a keyboard interrupt behave as if the interrupt were disabled at program startup, i.e., the function at pfn is never called.

RETURNS

Nothing.

EXAMPLE

A common use of onintr is to ensure a graceful exit on early termination:

```
    VOID rmtemp()
        {
        remove(uname());
        }
        ...
        onexit(&rmtemp);
        onintr(&exit);
```

Still another use is to provide a way of terminating long printouts, as with an interactive editor:

```
        while (!enter(docmd, NULL))
            putstr(STDOUT, "?\n", NULL);
        ...
    VOID docmd()
        {
        onintr(&leave);
        ...
```

SEE ALSO

enter(II), leave(II), onexit

NAME

open - open a file

SYNOPSIS

```
FILE open(fname, mode, rsize)
    TEXT *fname;
    COUNT mode;
    BYTES rsize;
```

FUNCTION

open opens a file fname and assigns a file descriptor to it. If (mode == 0) the file is opened for reading, else if (mode == 1) it is opened for writing, else (mode == 2) of necessity and the file is opened for updating (reading and writing).

If the file is to contain arbitrary binary data, as opposed to printable ASCII text, the record size rsize should be non-zero. Not all systems behave well if a text file is opened for updating.

RETURNS

open returns a file descriptor for the opened file, or a negative number, if unsuccessful.

EXAMPLE

```
    if ((fd = open("xeq", WRITE, 1)) < 0)
        write(STDERR, "can´t open xeq\n", 16);
```

SEE ALSO

close, create

NAME

read - read characters from a file

SYNOPSIS

```
COUNT read(fd, buf, size)
    FILE fd;
    TEXT *buf;
    BYTES size;
```

FUNCTION

read reads up to size characters from the file specified by fd into the buffer starting at buf.

RETURNS

If an error occurs, read returns a negative number; if end of file is encountered, read returns zero; otherwise the value returned is between 1 and size, inclusive, which is the number of characters actually read into buf.

EXAMPLE

To copy a file:

```
while (0 < (n = read(STDIN, buf, BUFSIZE)))
    write(STDOUT, buf, n);
```

SEE ALSO

write

NAME
remove - remove a file

SYNOPSIS
```
FILE remove(fname)
    TEXT *fname;
```

FUNCTION
remove removes the file fname; on most systems, this is an irreversible act.

RETURNS
remove returns zero, if successful, or a negative number.

EXAMPLE
```
    if (remove(uname()) < 0)
        putstr(STDERR, "can´t remove temp file\n", NULL);
```

NAME

sbreak - set system break

SYNOPSIS

```
TEXT *sbreak(size)
    ARGINT size;
```

FUNCTION

sbreak moves the system break, at the top of the data area, algebraically up by size bytes, rounded up as necessary to placate memory management hardware. There is no guarantee that successive calls to sbreak will deliver contiguous areas of memory, nor can all systems safely accept a call with negative size.

RETURNS

If successful, sbreak returns a pointer to the start of the added data area; otherwise the value returned is NULL.

EXAMPLE

```
if (!(p = sbreak(nsyms * sizeof (symbol))))
    {
    putstr(STDERR, "not enough room!\n", NULL);
    exit(NO);
    }
```

NAME

uname - create a unique file name

SYNOPSIS

```
TEXT *uname()
```

FUNCTION

uname returns a pointer to the start of a NUL terminated name which is likely not to conflict with normal user filenames. The name may be modified by a letter suffix (but not in place!), so that a family of process-unique files may be dealt with. The name may be used as the first argument to a create, or subsequent open, call, so long as any such files created are removed before program termination. It is considered bad manners to leave scratch files lying about.

RETURNS

uname returns the same pointer on every call during a given program invocation. The pointer will never be NULL.

EXAMPLE

```
    if ((fd = create(uname(), WRITE, 1)) < 0)
        putstr(STDERR, "can't create sort temp\n", NULL);
```

SEE ALSO

close, create, open, remove

NAME

write - write characters to a file

SYNOPSIS

```
COUNT write(fd, buf, size)
    FILE fd;
    TEXT *buf;
    COUNT size;
```

FUNCTION

write writes size characters starting at buf to the file specified by fd.

RETURNS

If an error occurs, writes either returns a negative number or a number other than size; otherwise size is returned.

EXAMPLE

To copy a file:

```
while (0 < (n = read(STDIN, buf, BUFSIZE)))
    if (write(STDOUT, buf, n) != n)
        {
        putstr(STDERR, "write error\n", NULL);
        exit(NO);
        }
```

SEE ALSO

read