III.b. CP/M-86 - DOS System Interface Library

NAME

Interface - to CP/M-86 or DOS system

FUNCTION

Programs written in C for operation on the 8086 family under CP/M-86 or DOS are translated into assembler according to the following specifications:

external identifiers - may be written in both upper and lower case; both cases are significant. The first eight letters must be distinct. Any underscore is left alone. An underscore is prepended to each identifier.

function text - is normally generated into the .text segment and is not to be altered or read as data. External function names are published via .public declarations.

literal data - such as strings, are normally generated into the .data segment. Switch tables are generated into the .text segment.

initialized data - are normally generated into the .data segment. External data names are published via .extern declarations.

uninitialized declarations - result in an .extern declaration as above, one instance per program file. The SMALL model of computation is used, i.e., it is presumed that the segment registers ds and ss contain the same value. The segment register es is never used, nor are segment overrides ever specified by generated code.

function calls - are performed by

1) moving arguments on the stack, right to left. Character data is sign-extended to integer, float is converted to double.

2) calling via "call _func", where _func is assumed by default to be in the same segment,

3) popping the arguments off the stack.

Except for returned value, the registers ax, cx, and dx, the flags, and the floating accumulators are undefined on return from a function call. All other registers are preserved. The returned value is in ax (char sign-extended to integer, integer, pointer to), or in dx/ax (long), or in the primary floating accumulator (float widened to double, double). Note that an intersegment call is generated to any function named by a "-far" flag to p2.86.

floating accumulators - are allocated in the sixteen-byte data area beginning at c_fac, if the 8087 processor is not used; the primary accumulator is at c_fac, the secondary is at c_fac+8. If the 8087 floating point processor is used ("-f" flag to p2), the primary floating accumulator is fr0 and the secondary is fr7; no other accumulators are modified, and the stack is left balanced at every call and return. The control word is never modified by generated code or

runtime support functions; hence it must be set properly at program startup, and may be modified during program operation. Its value at initialization is probably best for use with the math libraries; it is certainly unwise to specify a precision of less than 53 bits.

stack frames - are maintained by each C function, using bp as a frame pointer. On entry to a function, the call "call c_sav" will stack bp and leave it pointing at the stacked bp, then stack si, di, and bx. Arguments are at 4(bp), 6(bp), etc. and auto storage may be reserved on the stack at -7(bp) on down. To return, the jump "jmp c_ret" will use bp to restore bp, sp, bx, si, and di, then return. The jump "jmp c_fret" will restore the registers in the same way, then do an inter-segment return (as from a function named by a "-far" flag to p2.86). In either case, the previous sp is ignored, so the stack need not be balanced on exit, and none of the potential return registers are used. If within a "near" function it is not necessary to save or restore si, di, and bx, then the sequence "push bp/mov bp,sp" is used on entry and the jump "jmp c_rets" is used to return. If stack checking is requested ("-ck" flag to p2), the call "call c_savs" will behave as does c_sav, and in addition ensure that adding the quantity in ax to sp will not cause the stack to wrap around nor go below the value in the C variable _stop (__stop in assembler).

data representation - integer is the same as short, two bytes stored less significant byte first. Long integers are stored as two short integers, more significant short first; when copied to dx/ax, the more significant half is in dx. All signed integers are twos complement. Floating numbers are represented as for the 8087 processor, following the IEEE Standard, four bytes for float, eight for double, and are stored in ascending order of significance.

storage bounds - no storage bounds need to be enforced. Two-byte boundaries may be enforced, inside structures and among automatic variables, to significantly enhance performance and to ensure data structure compatibility with machines such as the PDP-11, but boundaries stronger than this are not fully supported by the stacking logic; the compiler may generate incorrect code for passing long or double arguments if a boundary stronger than even is requested.

module name - is not used.

SEE ALSO

c_fac(IV), c_ret(IV), c_rets(IV), c_sav(IV), c_savs(IV)

NAME

Conventions - CP/M system subroutines

SYNOPSIS

```
#include <cpm.h>
```

FUNCTION

All standard system library functions callable from C follow a set of uniform conventions, many of which are supported at compile time by including a system header file, <cpm.h>, at the top of each program. Note that this header is used in addition to the standard header <std.h>. The system header defines various system parameters and a useful macro or two.

Herewith the principal definitions:

```
CTRLZ - 032, ctl-Z for text end of file
EOF - 1, end of file from CP/M read
FAIL - -1, standard failure return code
MCREATE - 0, copen modes
MOPEN - 1
MREMOVE - 2
MWRITE - 4
SYSBUF - 0x80, location of CP/M buffer
CRESET - 0, CP/M system call codes
CRDCON - 1
CWRCON - 2
CRDRDR - 3
CWRPUN - 4
CWRLST - 5
CDCIO - 6
CGIOST - 7
CSIOST - 8
CPRBUF - 9
CRDBUF - 10
CICRDY - 11
CLFTHD - 12
CINIT - 13
CLOGIN - 14
COPEN - 15
CCLOSE - 16
CSRCH - 17
CSRCHN - 18
CDEL - 19
CREAD - 20
CWRITE - 21
CMAKE - 22
CRENAME - 23
CILOGIN - 24
CIDRNO - 25
CSETAD - 26
CIALLOC - 27
CWPROT - 28
CGETVEC - 29
CSETFA - 30
```

CGETPAR - 31
CGSUSER - 32
CRREAD - 33
CRWRITE - 34
CFSIZE - 35
CSETRR - 36
CDRESET - 37
CRZWRIT - 40
WOPEN - 1, the WCB flags
WDIRT - 2
WSTD - 4
WCR - 010
WUSED - 020
LST - -4, the device codes
PUN - -3
RDR - -2
CON - -1

SEE ALSO

CP/M or CDOS Manual

NAME

Conventions - DOS system subroutines

SYNOPSIS

```
#include <dos.h>
```

FUNCTION

All standard system library functions callable from C follow a set of uniform conventions, many of which are supported at compile time by including a system header file, <dos.h>, at the top of each program. Note that this header is used in addition to the standard header <std.h>. The system header defines various system parameters and a useful macro or two.

Herewith the principal definitions:

```
CTRLZ - 032, ctl-Z for text end of file
EOF - 1, end of file from DOS read
FAIL - -1, standard failure return code
MCREATE - 0, copen modes
MOPEN - 1
MREMOVE - 2
MWRITE - 4
SYSBUF - 0x80, location of DOS buffer
CRDCON - 1, DOS system call codes
CWRCON - 2
CRDRDR - 3
CWRPUN - 4
CWRLST - 5
CDCIN - 7
CDCOUT - 8
CPRBUF - 9
CRDBUF - 10
CICRDY - 11
CKBIN - 12
CINIT - 13
CLOGIN - 14
COPEN - 15
CCLOSE - 16
CSRCH - 17
CSRCHN - 18
CDEL - 19
CREAD - 20
CWRITE - 21
CMAKE - 22
CRENAME - 23
CIDRNO - 25
CSETAD - 26
CRREAD - 33
CRWRITE - 34
CFSIZE - 35
CSETRR - 36
CSETVEC - 37
CCRSEG - 38
CBREAD - 39
```

CBWRITE - 40
CGDATE - 42
CSDATE - 43
CGTIME - 44
CSTIME - 45
WOPEN - 1, the WCB flags
WDIRT - 2
WSTD - 4
WCR - 010
WXDIRT - 020
WXOPEN - 040
LST - -4, the device codes
PUN - -3
RDR - -2
CON - -1

SEE ALSO
DOS Manual

NAME

c - compiling C programs

SYNOPSIS

A:submit c sfile (CP/M)
A:c sfile (DOS)

FUNCTION

c is an indirect command file that causes a C source file to be compiled and assembled. It does so by invoking the compiler passes and the assembler in the proper order, then deleting the intermediate files.

The C source file is at sfile.c, where sfile is the name typed in the submit line. The relocatable image from the assembler is put at sfile.o.

EXAMPLE

To compile test.c under CP/M:

```
A:submit c test
```

SEE ALSO

ld, pc

FILES

sfile.tm?

BUGS

There should be some way to pass the -m flag to p1, or some of the myriad flags acceptable to pp, other than by modifying the command file proper.

NAME

pc - compiling Pascal programs

SYNOPSIS

A:submit pc sfile (CP/M)
A:pc sfile (DOS)

FUNCTION

pc is an indirect command file that causes a Pascal source file to be compiled and assembled. It does so by invoking the compiler passes and the assembler in the proper order, then deleting the intermediate files.

The Pascal source file is at sfile.p, where sfile is the name typed in the submit line. The relocatable image from the assembler is put at sfile.o.

EXAMPLE

To compile test.p under CP/M:

```
A:submit pc test
```

SEE ALSO

c, ld

FILES

sfile.tm?

BUGS

There should be some way to pass flags to ptc, other than by modifying the command file proper.

NAME

ld - linking a C program

SYNOPSIS

A:submit ld ofile (CP/M)
A:ld ofile (DOS)

FUNCTION

Programs written in C must be linked with certain object modules that implement the standard C runtime environment. The ld command script invokes the link program, including the standard object header file, the object file ofile, and any libraries in the correct order, and produces the executable image xeq.com.

The standard object header gets control when xeq.com is run. It calls the function _main(), described in the system library, which reads in a command line, redirects the standard input and output as necessary, calls the user provided main(ac, av) with the command arguments, and passes the value returned by main on to exit(). All of this malarkey can be circumvented if file contains a defining instance of _main().

ofile must be a standard object file produced by the assembler, or by an earlier (partial binding) invocation of link. The assembler accepts either the output of the C code generator p2 or assembler source that satisfies the interface requirements of C, as described earlier under Interface. The important thing is that the function main(), or _main(), be defined somewhere among these files, along with any other modules needed and not provided by the standard C library.

EXAMPLE

To compile and run the program echo.c under CP/M:

```
A:submit c echo
A:submit ld echo
A:xeq hello world!
hello world!
```

SEE ALSO

c, pc

NAME

to86 - convert to CP/M-86 or DOS executable format

SYNOPSIS

to86 -[exe b# o* s zb zh] <file>

FUNCTION

to86 translates standard 8086 object files to CP/M-86 ".cmd" or DOS ".exe" format files.

The flags are:

-exe produce a DOS ".exe" format file. Default is CP/M-86 ".cmd" format.

-b# override the stack plus heap size in the standard object file and set it to #. to86 takes this value as the minimum number of bytes to reserve beyond the end of the data group for stack and data area growth. If the value is odd, the space reserved is actually a 64K byte group.

-o* write the output to the file *. If -exe is specified, default is "xeq.exe"; otherwise the default is "xeq.cmd".

-s make code group shareable, if small model is used.

-zb generate zeros for the bss segment. Default is to reserve space beyond the end of the data group image for bss, as well as for stack and data area growth. The standard startup routines zero the bss segment before _main is called, so this option is needed only for alternate startup sequences.

-zh generate zeros for the stack plus heap, as well as the bss segment. Once again, this should not be necessary, given the standard startup routines.

If <file> is present, it is used as the input rather than the default "xeq".

An output ".cmd" file consists of a 128-byte header, followed by one or two group images. If the data bias in the object file lies below the end of the text segment, then the "small" model is used and both code and data groups are generated. Otherwise, the "8080" model is used and only a combined code/data group is generated. In either case, the group into which data is generated must have no input segment biased below 0x100, to leave room for the base page.

For each group generated, a nine-byte group descriptor is written into the header, commencing with byte zero; the unused portion of the header is filled with zeros. A group descriptor consists of a format byte, followed by four two-byte unsigned paragraph lengths, written less significant byte first. The format byte is 0x01 for a nonsharable code group, 0x02 for a data group, and 0x09 for a sharable code group. The first paragraph length is the size of the corresponding group image in the file, in multiples of sixteen bytes, naturally. The second provides an absolute load

address if nonzero; to86 always sets this to zero. The third paragraph length is the minimum space to reserve for the group, typically zero unless larger than the size of the group image. And the last length provides a maximum space to reserve; it too is always zero.

The group images that follow are padded as necessary to be a multiple of sixteen bytes in length.

An output ".exe" file consists of a 32-byte header, followed by the group images as described above. The header consists of sixteen two-byte words, each written less significant byte first. Word zero is the magic number 0x5a4d; word two is the size of the file in 512-byte pages; word four is the size of the header in 16-byte paragraph units, always 2; word five is the size of the bss plus stack and heap to reserve beyond the end of the image, also in paragraph units; word six is the high/low loader switch, always -1 for low memory loading; word seven is the stack pointer; word eight is the stack segment register, relative to the start of the program image; word nine is the checksum of the entire file, taken as an array of two-byte words; and word ten is the program counter. All other words in the header are zero.

In the small model, the stack segment register points at the start of the combined data/stack group, which follows the code group. In all cases, the code segment register points at the start of the Program Segment Prefix; hence the text segment must be biased at 0x100 for either the 8080 or the small model.

EXAMPLE

To generate a CP/M executable file for the 8080 model:

```
% link -tb0x100 -ed__edata -eb__memory chdr.o PROG.o libc.86
% to86 -b1 -o PROG.cmd
```

And for the small model:

```
% link -db0x100 -ed__edata -eb__memory chdr.o PROG.o libc.86
% to86 -b0x4000 -o PROG.cmd
```

In either case, the standard object file is left at "xeq".

To make a DOS executable file for the 8080 model:

```
% link -tb0x100 -ed__edata -eb__memory chdr.o PROG.o libc.86
% to86 -exe -b0x4000 -o PROG.exe
```

And for the small model:

```
% link -tb0x100 -db0 -ed__edata -eb__memory chdr.o PROG.o libc.86
% to86 -exe -b0x4000 -o PROG.exe
```

Note that a DOS ".com" file can be produced, for the 8080 model, directly by link:

```
% link -htr -tb0x100 -ed__edata -eb__memory -o PROG.com -i
chdr.o
PROG.o
libc.86
```

SEE ALSO

as.86(II), link(II)

NAME

chdr - C runtime entry for CP/M-86

SYNOPSIS

link -tb0x100 -ed__edata -eb__memory [fpphdr.o] chdr.o <files>

FUNCTION

All CP/M-86 programs begin execution at the start of chdr.o, which is typically linked first when constructing an executable file. fpphdr.o precedes chdr.o if the 8087 floating point processor must be initialized at the start of the program. This startup code sets the stack to the top of the space reserved for the data segment, clears the bss area, then calls _main. If the stack top fence _stop has not been otherwise initialized, the startup code also sets _stop to point just beyond the bss area.

Included in the chdr module are the C callable interface functions cpm() and cpmx(), plus several internal functions that isolate CP/M-86 system dependencies.

SEE ALSO

_main, cpm, cpmx

NAME

doshdr - C runtime entry for DOS

SYNOPSIS

link -htr -tb0x100 -ed__edata -eb__memory [fpphdr.o] doshdr.o <files>

FUNCTION

All DOS programs begin execution at the start of doshdr.o, which is typically linked first when constructing an executable file. fpphdr.o precedes doshdr.o if the 8087 floating point processor must be initialized at the start of the program. For a ".com" file, these headers must be loaded first. This startup code, on a ".exe" file, copies the command line at 0x80 in the program segment prefix to the top of the stack area, then sets ds to match ss. For any file, it also clears the bss area, then calls _main. If the stack top fence _stop has not been otherwise initialized, the startup code also sets _stop to point just beyond the bss area.

Included in the doshdr module are the C callable interface functions dos(), dosx(), and dosy(), plus several internal functions that isolate DOS system dependencies.

SEE ALSO

DOS2, _main, dos, dosx, dosy

NAME

DOS2 - enhancements for DOS Version 2.0

SYNOPSIS

link -htr -tb0x100 -ed__edata -eb__memory doshdr.o dmain.o <files>

FUNCTION

DOS Version 2.0 contains a number of enhancements over earlier releases, including command line I/O redirection and hierarchical filesystems. The standard interface provided for DOS and CP/M-86 is intentionally restricted to those features that are implemented uniformly across multiple versions, to assure maximum portability; hence command line redirection is performed by the startup function _main(), and no provision is made for reaching files other than in the current directory of each disk.

dmain.o contains a replacement for _main(), plus several other interface functions, to make use of the added features. At the cost of incompatibility with earlier releases of DOS, using dmain.o offers the following advantages:

arbitrary pathnames - such as "c:\lib\hdrs\std.h" may be used wherever a filename is expected. Note that DOS accepts the alternate form "c:/lib/hdrs/std.h" from user programs, if not always on the command line of standard utilities.

smaller programs - since command line redirection and I/O buffering is left to the system.

uppercase command line text - since the command line need not be forced to lowercase for portability.

saner type ahead - since the console need not be scanned for a CTL-C to provide early termination. Note that the standard interface function onintr() can be used to handle an interrupt caused by typing CTL-C, with the standard interface, or CTL-BREAK with the standard or dmain.o DOS interface.

proper input redirection - since the new command line redirection seems to misbehave for programs with the older style system calls. Command line redirection under DOS 2.0, using the standard interface, is best achieved by placing the entire argument string within double quotes, as in "pp <infile", to permit _main() to do the job properly instead of trying to use DOS.

standard device names - may be used, rather than the CP/M equivalents required in the standard interface.

proper file size - is better determined, to better than the nearest multiple of 128 bytes.

exit status - is reported back to the invoker for testing. A status of zero means success, one means failure.

In all, the following functions are modified in some form by dmain.o: _main(), close(), create(), exit(), lseek(), open(), read(), remove, and write().

NAME

_main - setup for main call

SYNOPSIS

BOOL _main()

FUNCTION

_main is the function called whenever a C program is started. It parses the command line at 0x80 (in the program base) into argument strings, redirects STDIN and STDOUT as specified in the command line, then calls main.

The command line is interpreted as a series of strings separated by spaces. If a string begins with a '<', the remainder of the string is taken as the name of a file to be opened for reading text and used as the standard input, STDIN. If a string begins with a '>', the remainder of the string is taken as the name of a file to be created for writing text and used as the standard output, STDOUT. All other strings are taken as argument strings to be passed to main. The command name, av[0], is taken from _pname.

Note that the argument strings remain in the default record buffer beginning at 0x80 in the program base, which is never used by other C interface routines.

EXAMPLE

To avoid the loading of _main and all the the file I/O code it calls on, one can provide a substitute _main instead:

```
BOOL _main()
    {
    <program body>
    }
```

RETURNS

_main returns the boolean value obtained from the main call, which is then passed to exit.

SEE ALSO

_pname, cpm, exit

NAME

_pname - program name

SYNOPSIS

```
TEXT *_pname;
```

FUNCTION

_pname is the (NUL terminated) name by which the program was invoked, at least as anticipated at compile time. If the user program provides no definition for _pname, a library routine supplies the name "error", since it is used primarily for labelling diagnostic printouts.

Argument zero of the command line is set equal to _pname.

SEE ALSO

_main

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. If the file was written to or created, close ensures that the last record is written out and the directory entry properly closed.

RETURNS

close returns the now useless file descriptor, if successful, or -1.

EXAMPLE

To copy an arbitrary number of files:

```
while (0 <= (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

cpm - call CP/M-86 system

SYNOPSIS

```
COUNT cpm(cx, dx)
    COUNT cx;
    TEXT *dx;
```

FUNCTION

cpm is the C callable function that permits arbitrary calls to be made on CP/M-86. It loads its arguments into registers cx and dx, then performs a system call via interrupt 224. The function to be performed is specified by the less significant byte of cx, i.e., the cl register; typically dx contains a word integer or a pointer.

RETURNS

cpm returns the al register returned by the CP/M-86 call, sign extended to a word integer.

EXAMPLE

To read a line from the console:

```
buf[0] = 125;
cpm(CRDBUF, buf);
cpm(CWRCON, '\n');
```

SEE ALSO

cpmx

NAME

cpmx - call CP/M-86 system with doubleword result

SYNOPSIS

```
LONG cpmx(cx, dx)
    COUNT cx;
    TEXT *dx;
```

FUNCTION

cpmx is the C callable function that permits arbitrary calls to be made on CP/M-86, when a doubleword result is expected. It loads its arguments into registers cx and dx, then performs a system call via interrupt 224. The function to be performed is specified by the less significant byte of cx, i.e., the cl register; typically dx contains a word integer or a pointer.

RETURNS

cpmx returns the bx and es registers returned by the CP/M-86 call, packed as a long integer; bx is the less significant sixteen bits.

EXAMPLE

To obtain a disk allocation vector:

```
IMPORT LONG cpmx();
union {
    LONG lo;
    struct {
        UCOUNT bx, es;
            } x;
        } rval;

rval.lo = cpmx(CIALLOC);
```

SEE ALSO

cmp

NAME

create - open an empty instance of a file

SYNOPSIS

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

FUNCTION

create makes a new file of specified name, 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). This mode information is largely ignored, however.

If (rsize is zero), carriage returns and NULs are deleted on input, a ctl-Z is treated as an end of file, a carriage return is injected before each newline on output, and a ctl-Z is appended to the data in a partially filled last record. If (rsize != 0) data is transmitted unaltered.

Filenames take the general form x:name.typ, where x is the disk designator, name is the eight-character filename, and typ its three-character type. If x: is omitted, it is taken as the disk logged in on the first call to create, open, or remove; a missing name or typ is taken as all blanks. Letters are forced uppercase.

The four physical devices con:, rdr:, pun:, and lst: are also accepted as filenames. Reading from pun: or lst: gives an instant end of file, while writing to rdr: sends all bytes to hell with no complaint.

RETURNS

create returns a file descriptor for the created file or -1.

EXAMPLE

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

SEE ALSO

close, open, remove, uname

NAME

dos - call DOS system

SYNOPSIS

```
COUNT dos(axrev, dx, cx)
    COUNT axrev;
    TEXT *dx, *cx;
```

FUNCTION

dos is the C callable function that permits arbitrary calls to be made on DOS. It loads its arguments into registers ax (axrev with bytes reversed), dx, and cx, then performs a system call via interrupt 21. The function to be performed is specified by the less significant byte of axrev, i.e., the ah register; typically dx contains a word integer or a pointer; cx contains an optional extra argument.

RETURNS

dos returns the al register returned by the DOS call, sign extended to a word integer.

EXAMPLE

To read a line from the console:

```
buf[0] = 125;
dos(CRDBUF, buf);
dos(CWRCON, '\n');
```

SEE ALSO

dosx, dosy

NAME

dosx - call DOS system with doubleword result

SYNOPSIS

```
LONG dosx(axrev, dx, cx)
    COUNT axrev;
    TEXT *dx, *cx;
```

FUNCTION

dosx is the C callable function that permits arbitrary calls to be made on DOS, when a doubleword result is expected. It loads its arguments into registers ax (axrev with bytes reversed), dx, and cx, then performs a system call via interrupt 21. The function to be performed is specified by the less significant byte of axrev, i.e., the ah register; typically dx contains a word integer or a pointer; cx contains an optional extra argument.

RETURNS

dosx returns the dx and cx registers returned by the DOS call, packed as a long integer; dx is the less significant sixteen bits.

EXAMPLE

To get the date:

```
IMPORT LONG dosx();
union {
    LONG lo;
    struct {
        TINY day, month;
        COUNT year;
        } x;
    } rval;

rval.lo = dosx(CGDATE);
```

SEE ALSO

dos, dosy

NAME

dosy - call DOS 2.0 system

SYNOPSIS

```
COUNT dosy(axrev, dx, cx, bx)
    COUNT axrev;
    TEXT *dx;
    BYTES cx, bx;
```

FUNCTION

dosy is the C callable function that permits most of the new calls to be made on DOS 2.0. It loads its arguments into registers ax (axrev with bytes reversed), dx, cx, and bx, then performs a system call via interrupt 21. The function to be performed is specified by the less significant byte of axrev, i.e., the ah register; typically dx contains a word integer or a pointer; cx contains a count; and bx contains a "handle" or file descriptor.

RETURNS

If there is no error, dosy returns the ax (or dx/ax) register returned by the DOS call; otherwise dx/ax contains the DOS error return code, negated.

EXAMPLE

To copy input to output:

```
while (0 <= (n = dosy(SYREAD, buf, sizeof (buf), STDIN)))
    if (dosy(SYWRITE, buf, n, STDOUT) != n)
        dosy(SYWRITE, "write error\n", 12, STDERR);
```

SEE ALSO

dos, dosx

NAME

exit - terminate program execution

SYNOPSIS

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

FUNCTION

exit calls all functions registered with onexit, then terminates program execution. success is ignored.

RETURNS

exit will never return to the caller.

EXAMPLE

```
if ((fd = open("file", READ)) < 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 file fd, under control of sense. If (sense == 0) the pointer is set to offset, which should be positive; if (sense == 1) the offset is algebraically added to the current pointer; otherwise (sense == 2) of necessity and the offset is algebraically added to the length of the file in bytes to obtain the new pointer. The system uses only the low order 31 bits of the offset; the sign is ignored.

RETURNS

lseek returns zero if successful, or -1.

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, 512) != 512);
    }
```

BUGS

The length of the file is taken as 128 times the total number of records, even if a text file is terminated early with a ctl-Z.

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 not to be NULL.

EXAMPLE
```
    IMPORT 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 a keyboard interrupt, usually caused by a DEL key typed during terminal input or output. (Under DOS on the 8086, a CTL-BREAK also serves this function.) Any earlier call to onintr is overridden.

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

If (pfn == NULL) then these interrupts are disabled (turned off).

RETURNS

Nothing.

EXAMPLE

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

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

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

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

SEE ALSO

onexit

NAME

open - open an existing file

SYNOPSIS

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

FUNCTION

open associates a file descriptor with an existing file. 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). This mode information is largely ignored, however.

If (rsize == zero), carriage returns and NULs are deleted on input, a ctl-Z is treated as an end of file, a carriage return is injected before each newline on output, and a ctl-Z is appended to the data in a partially filled last record. If (rsize != 0) data is transmitted unaltered.

Filenames take the general form x:name.typ, where x is the disk designator, name is the eight-character filename, and typ its three-character type. If x: is omitted, it is taken as the disk logged in on the first call to create, open, or remove; a missing name or typ is taken as all blanks. Letters are forced uppercase.

The four physical devices con:, rdr:, pun:, and lst: are also accepted as filenames. Reading from pun: or lst: gives an instant end of file, while writing to rdr: sends all bytes to hell with no complaint.

RETURNS

open returns a file descriptor for the file or -1.

EXAMPLE

```
if ((fd = open("xeq", WRITES, 1)) < 0)
    write(STDERR, "can't open xeq\n", 15);
```

SEE ALSO

close, create, remove, uname

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. If the file was created or opened with (rsize == 0) carriage returns and NULs are discarded on input.

Reading from pun: or lst: always gives a count of zero.

If the console is read, DEL at the start of a line causes an interrupt, to be processed as specified by onintr.

RETURNS

If an error occurs, read returns -1; if end of file is encountered, read returns zero; otherwise the value returned is between 1 and size, inclusive.

EXAMPLE

To copy a file:

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

SEE ALSO

onintr, write

NAME

remove - remove a file

SYNOPSIS

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

FUNCTION

remove deletes the file fname from the filesystem.

RETURNS

remove returns zero, if successful, or -1.

EXAMPLE

```
if (remove("temp.c") < 0)
    write(STDERR, "can't remove temp file\n", 23);
```

NAME

sbreak - set system break

SYNOPSIS

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

FUNCTION

sbreak moves the system break, at the top of the data area, algebraically up by size bytes.

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))))
    {
    write(STDERR, "not enough room!\n", 17);
    exit(NO);
    }
```

BUGS

The stack is assumed to lie above the data area, so sbreak will return a NULL if the new system break lies above the current stack pointer; this may not be desirable behavior on all memory layouts.

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, 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, which is currently the string "ctempc.". The pointer will never be NULL.

EXAMPLE

```
if ((fd = create(uname(), WRITE, 0)) < 0)
    write(STDERR, "can't create sort temp\n", 23);
```

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. If the file was created or opened with (rsize == 0), each newline output is preceded by a carriage return. Moreover, a ctl-Z is appended to a file that does not end on a record boundary.

RETURNS

If an error occurs, write returns -1; otherwise the value returned should be size. Writing to rdr: does nothing, but returns size as if it did everything.

Characters typed at the console are inspected during write calls; typing a DEL causes an interrupt, to be processed as specified by onintr.

EXAMPLE

To copy a file:

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

SEE ALSO

onintr, read