III.c. CP/M-68k System Interface Library

NAME

Interface - to CP/M-68k system

FUNCTION

Programs written in C for operation on the MC68000 under CP/M-68k are translated according to the following specifications:

external identifiers - may be written in both upper and lowercase. The first eight letters must be distinct. An underscore '_' is prepended to each identifier.

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

literal data - such as strings, double constants, and switch tables, are normally generated into .text.

initialized data - is normally generated into .data. External data names are published via .globl declarations.

uninitialized declarations - result in an .globl reference, one instance per program file.

function calls - are performed by

1) moving arguments onto the stack, right to left. Character and short data are widened to long integer, float is converted to double.

2) calling via "jsr _func".

3) popping the arguments off the stack.

Except for returned value, the registers d0, d1, d2, d6, d7, a0, a1, a2, and the condition codes are undefined on return from a function call. All other registers are preserved. The returned value is in d7 (char or short widened to long integer, long integer, and pointer to), or in d6-d7 (float widened to double, and double).

stack frames - are maintained by each C function, using a6 as a frame pointer. On entry to a function, the instruction 'link a6,#' will stack a6 leaving a6 pointing to the stacked a6, and will reserve # bytes for automatics. If more than 32,768 bytes of autos are specified, additional code is generated to reserve space on the stack. Any non-volatile registers used (d3, d4, d5, a3, a4, a5) are then stacked and, if the top of stack scratch cell is used, an additional four bytes are reserved (by either stacking d0 along with other registers or by modifying the stack pointer). Arguments are now at 8(a6), 12(a6), etc. and auto storage is on the stack at -1(a6) on down. To return, the nonvolatile registers are restored from the stack, a6 and a7 are retrieved with an "unlk a6" and control is returned via "rts". Note that the stack must be balanced on exit if the nonvolatile registers are to be properly restored.

data representation - short integers are stored as two bytes, more significant byte first. Long integers are stored as four bytes, in descending order of significance. Floating numbers are represented as for the proposed IEEE Floating Point Standard, four bytes for float, eight for double, and are stored in descending order of byte significance. The IEEE representations are: most significant bit is one for negative numbers, else zero; next eleven bits (eight for float) are the characteristic, biased such that the binary exponent of the number is the characteristic minus 1022 (126 for float); remaining bits are the fraction, starting with the 1/4 weighted bit. If the characteristic is zero, the entire number is taken as zero and should be all zero to avoid confusing some routines that take shortcuts. Otherwise there is an assumed 1/2 added to all fractions to put them in the interval [0.5, 1.0). The value of the number is the fraction, times -1 if the sign bit is set, times 2 raised to the exponent.

storage bounds - even byte storage boundaries must be enforced for multibyte data. 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.

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

c - compiling C programs

SYNOPSIS

```
A:submit c sfile
```

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:

```
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
```

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:

```
    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
```

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:

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

SEE ALSO

c, pc

NAME

to68k - convert to CP/M-68k executable format

SYNOPSIS

to68k -[bb## o* r t] <file>

FUNCTION

to68k translates standard MC68000 object files to CP/M-68k ".68k" format or CP/M-68k ".rel" format.

The flags are:

-bb## assume the bss bias is set to ##, instead of just beyond the data segment. This flag, if used, must match the "-bb##" flag given to link when the input file was created, since the standard header has no provision for recording a peculiar bss bias.

-o* write the output to the file *. Default is "xeq.68k".

-r suppress relocation bits in the output module.

-t suppress symbol table in the output module.

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

The output file consists of a 28 or 36 byte header, followed by a text segment, a data segment, the symbol table, and relocation information. The header consists of a short magic number followed by the long number of bytes of object code defined by the text segment, the long number of bytes defined by the data segment, the long number of bytes defined by the bss segment, the long number of bytes in the symbol table, a zero padding long, the long relocation bias of the text segment, and a short that is nonzero if relocation bits are suppressed. To this may be appended the long relocation bias of the data segment and the long relocation bias of the bss segment. All integers in the object image are written most significant byte first, in descending order of significance.

The value of the magic number is determined by the relationship between the text, data, and bss biases in the input file header. If the data segment immediately follows the text, and the bss segment immediately follows the data, the magic number is 0x601a and the shorter header is used. Otherwise the magic number is 0x601b and the longer header is used to specify the additional biases.

Text and data segments each consist of an integral number of shorts.

Each symbol table entry consists of an eight-byte name padded with trailing NULs, a flag short, and a value long. Meaningful flag values are 0x200 for text relative, 0x400 for data relative, and 0x100 for bss relative. To this is added 0x8000 if the symbol is defined, and 0x2000 if the symbol is to be globally known.

One short of relocation information is present for each short of text and data. A relocation short of 0 implies no relocation, 02 is for text relative, 01 for data relative, 03 for bss relative, and 04 for an undefined external symbol whose index into the symbol table is the relocation word right shifted three. A short of 05 flags the upper half of a long to be relocated by the short following, and a short of 06 implies 16-bit PC-relative relocation.

EXAMPLE

To translate a standard object file to MC68000 ".68k" format:

```
% link -tb0x1200 -ed__edata -eb__memory Crtcpm.68k PROG.o libccpm.68k
% to68k -o PROG.68k
```

Note that a ".68k" file can be produced, without relocation bits or symbol table, directly by link, by using the files cout or coutl, which emulate a CP/M-68k executable file header. The following command line would produce a file biased to run at location 0x1200, with a magic number of 0x601a:

```
% link -tb0x11e4 -htr -ed__edata -eb__memory -st__stext \
    -sd__sdata -sb__sbss -o PROG.68k -i
cout.68k
Crtcpm.68k
PROG.o
libccpm.68k
```

To translate a standard object file to MC68000 ".rel" format:

```
% link -ed__edata -eb__memory Crtcpm.68k PROG.o libccpm.68k
% to68k -o PROG.rel
```

This is how to68k should be used in conjunction with the standard compiler driver scripts; the link line shown is equivalent to the one in the scripts. A ".rel" file can be converted to ".68k" format under CP/M-68k.

SEE ALSO

as.68k(II), cout, coutl, link(II)

NAME

chdr - C runtime entry

SYNOPSIS

```
link -tb0x1200 -ed__edata -eb__memory a:chdr.o <files> libc.68k
to68k -o prog.68k
```

FUNCTION

All CP/M-68k programs begin execution at location 0x100 in the program base; chdr.o is the startup routine, linked at this address, that sets the stack to the address stored at location 6 in the program base, clears bss (by zeroing memory from __edata to __memory), then calls _main().

Included in the chdr module is the C callable interface function cpm(), plus several internal functions that isolate 68000 machine dependencies.

SEE ALSO

_main, cpm

NAME

cout - simulate a CP/M-68k executable file header

SYNOPSIS

```
link -eb__memory -ed__edata -st__stext -sd__sdata -sb__sbss -htr \
    -tb-28 cout.68k Crtcpm.68k <files> libucpm.68k libccpm.68k
```

FUNCTION

cout is used in conjunction with the stadard link utility to generate ".68k" files that will execute under CP/M on an MC68000. The text bias (value for -tb) given to link should be the actual runtime location desired minus 28 bytes. cout will cause a ".68k" file with a magic number of 601A to be generated.

EXAMPLE

To produce a ".68k" file, without relocation bits or symbol table, linked to run at location 0x500:

```
% link -eb__memory -ed__edata -st__stext -sd__sdata -sb__sbss -htr \
    -tb0x4e4 cout.68k Crtcpm.68k <files> libucpm.68k libccpm.68k
```

NAME

coutl - simulate a long CP/M-68k executable file header

SYNOPSIS

```
link -eb__memory -ed__edata -st__stext -sd__sdata -sb__sbss -htr \
    -tb-36 coutl.68k Crtcpm.68k <files> libucpm.68k libccpm.68k
```

FUNCTION

coutl is used in conjunction with the standard link utility to generate ".68k" files that will execute under CP/M on an MC68000. The text bias (value for -tb) given to link should be the actual runtime location desired minus 36 bytes. coutl will cause a ".68k" file with a magic number of 601B to be generated.

EXAMPLE

To produce a ".68k" file, without relocation bits or symbol table, linked to run at location 0x500, with a data bias rounded up to the next higher 64-byte boundary:

```
% link -eb__memory -ed__edata -st__stext -sd__sdata -sb__sbss -htr \
    -tb0x4dc -dr6 coutl.68k Crtcpm.68k <files> libucpm.68k libccpm.68k
```

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-68k system

SYNOPSIS

```
COUNT cpm(d0, d1)
    COUNT d0;
    TEXT *d1;
```

FUNCTION

cpm is the C callable function that permits arbitrary calls to be made on CP/M-68k. It loads its arguments into registers d0 and d1, then performs a system call via trap #2. The function to be performed is specified by the less significant word of d0, i.e. d0.w; typically d1 contains a word integer or a pointer.

RETURNS

cpm returns the d0.b register returned by the CP/M-68k call, sign extended to a long integer.

EXAMPLE

To read a line from the console:

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

BUGS

The return value from the system call may be corrupted, if larger than a signed character.

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

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