III.b. P/OS System Interface Library

III.b - 1	**Interface**	to P/OS system
III.b - 3	**Conventions**	RMS system subroutines
III.b - 4	**c350**	compiling C programs
III.b - 5	**pc350**	compiling Pascal programs
III.b - 6	**ctkb**	task building C programs
III.b - 8	**_main**	setup for main call
III.b - 9	**_pname**	program name
III.b - 10	**close**	close a file
III.b - 11	**create**	open an empty instance of a file
III.b - 12	**emt**	make a system call
III.b - 13	**entrap**	enter function on system trap
III.b - 15	**exit**	terminate program execution
III.b - 16	**fcall**	call a Professional library routine
III.b - 18	**lseek**	set file read/write pointer
III.b - 19	**onexit**	call function on program exit
III.b - 20	**onintr**	capture interrupts
III.b - 21	**open**	open a file
III.b - 22	**read**	read from a file
III.b - 23	**remove**	remove a file
III.b - 24	**sbreak**	set system break
III.b - 25	**uname**	create a unique file name
III.b - 26	**write**	write to a file

NAME

Interface - to P/OS system

FUNCTION

Programs written in C for operation on the PDP-11 under P/OS are translated according to the following specifications:

external identifiers - may be written in both upper and lowercase, but only one case is significant. The first six letters must be distinct. Any underscore is changed to a '.' The identifier "_" must be avoided.

function text - is normally generated into the c$text section 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 the c$data section.

initialized data - is normally generated into the c$data section. External data names are published via .globl declarations.

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

function calls - are performed by

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

2) calling via 'jsr pc,func'.

3) popping the arguments off the stack.

Except for returned value, the registers r0, r1, fr0, fr1, and the condition code are undefined on return from a function call. All other registers are preserved. The returned value is in r0 (char sign-extended to integer, integer, pointer to), or in r0-r1 (long), or in fr0 (float widened to double, double). If the floating point processor is not to be used, the double storage location c$fac is the analog of fr0, c$fac+010 is the analog of fr1. If the FPP is used, its mode must be double/integer on entry and exit of a C function.

stack frames - are maintained by each C function, using r5 as a frame pointer. On entry to a function, the call 'jsr r5,c$sav' will stack r5, r4, r3, and r2 and leave r5 pointing at the stacked r5. Arguments are now at 4(r5), 6(r5), etc. and auto storage may be reserved on the stack at -7(r5) on down. To return, the jump 'jmp c$ret' will use r5 to restore r2, r3, r4, r5 and sp then return via 'rts pc'. The previous sp is ignored, so the stack need not be balanced on exit, and none of the potential return registers are used. The alternate jump 'jmp c$rets' will return without restoring r2, r3, r4.

data representation - integer is the same as short, two bytes stored less significant byte first. Long integers are stored as two two-byte integers, more significant integer first. Floating numbers are represented as for the PDP-11 Floating Point Processor, four bytes for float, eight for double, and are stored as two or four short integers, in descending order of significance.

storage bounds - Even byte storage boundaries must be enforced for multi-byte data. The compiler may generate incorrect code for passing long or double arguments if a boundary stronger than even is requested.

module name - is taken as the first defined external encountered, unless an explicit module name is given to the code generator. The module name is published via a .title declaration.

SEE ALSO

c~ret(IV), c~rets(IV), c~sav(IV), c~savs(IV)

NAME

Conventions - RMS system subroutines

SYNOPSIS

```
#include <pos.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, <pos.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:

```
FAIL - -1, the standard failure return
NFILES - 10, maximum number of open files
NOBLOCK - 010000000000, seek address representing no real block
WCR - 0020, strip carriage returns
WDIRT - 0100, block must be written back
WFIX - 0040, binary I/O
WREAD - 0001, reads permitted
WWRITE - 0002, writes permitted
WOPEN - 0004, file is open
WTTY - 0010, file is a terminal
```

SEE ALSO

RMS I/O manuals

NAME

c350 - compiling C programs

SYNOPSIS

```
>@lb:[1,1]c350
*FILE[S]: sfile
```

FUNCTION

c350 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 MACRO-11 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 response to the prompt from c350. The object file output from MACRO-11 is put at sfile.obj.

EXAMPLE

To compile test.c:

```
>@lb:[1,1]c350
*FILE[S]: test
```

FILES

ctmp1.tmp, ctmp2.tmp, ctmp3.tmp

BUGS

There should be some way to pass the -m flag to rp1, or some of the myriad flags acceptable to rpp and rp2, other than by modifying the command file proper.

NAME

pc350 - compiling Pascal programs

SYNOPSIS

```
>@lb:[1,1]pc350
*FILE[S]: sfile
```

FUNCTION

pc350 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 MACRO-11 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 response to the prompt from pc350. The object file output from MACRO-11 is put at sfile.obj.

EXAMPLE

To compile test.p:

```
>@lb:[1,1]pc350
*FILE[S]: test
```

SEE ALSO

c350

FILES

ctmp1.tmp, ctmp2.tmp, ctmp3.tmp

BUGS

There should be some way to pass the -m flag to rp1, or some of the myriad flags acceptable to ptc, rpp and rp2, other than by modifying the command file proper.

NAME

ctkb - task building C programs

SYNOPSIS

Under RSX:
```
>PAB
PAB>@FILE
```

Under VMS:
```
$ RUN SYS$SYSTEM:PROTKB
PAB>@FILE
```

FUNCTION

Programs written in C must be linked with certain object modules that implement the standard C runtime environment. The normal Toolkit PAB program is used, but with several important constraints. The above example produces an executable file "file.tsk" from one or more object files that presumably were produced by the C compiler, using an indirect file "file.cmd", which might look like:

```
file/cp=<files>/mp
clstr=rmres,posres:ro
units=12
stack=3000
gbldef=tt$efn:1
gbldef=tt$lun:5
gbldef=mn$lun:6
gbldef=hl$lun:7
gbldef=ms$lun:10
gbldef=wc$lun:11
```

The object files and libraries to link are specified in "file.odl", which might look like:

```
        .root ctest-rmsrot,rmsall
ctest:  fctr pchdr-<files>-poslib/lb
@lb:[1,5]rmsrlx
        .end
```

The indirect file above shows the constraints needed in running PAB. Since C is a stack intensive language, it is almost always necessary to provide a larger stack than that provided by default. The STACK=3000 option provides a stack that seems to be sufficient for most C programs. It may have to be changed, however, to suit individual programs.

The C runtime has provision for up to ten files to be open simultaneously, counting STDIN, STDOUT, and STDERR. Logical unit number 5 is reserved for console I/O; addition LUNs are specified for special P/OS services.

pchdr.obj is the module that gets control when file.tsk is run. It calls the function _main(), described later in this section, which prompts for 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 circum-

vented if <files> contains a defining instance of _main().

<files> can be one or more object modules produced by the C compiler, or produced from MACRO-11 source that satisfies the interface requirements of C, as described in the Interface page earlier in this section. 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.

poslib.olb is a library containing all of the portable C library functions, plus those required for the P/OS interface. It should be scanned last.

EXAMPLE

To compile and build the program echo.c:

```
>@C350
*FILE[S]: ECHO
>PAB
PAB>@ECHO
```

where "echo.cmd" and "echo.odl" are available as described above. The file "echo.tsk" is now ready for downloading to P/OS.

SEE ALSO

Interface, _main, c350

NAME

_main - setup for main call

SYNOPSIS

BOOL _main()

FUNCTION

_main is the function called whenever a C program is started. It opens STDIN, STDOUT, and STDERR to the console, obtains a command line by prompting at the bottom of the screen, parses it into argument strings, sets _pname to argument zero, 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 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 variable length records and used as the standard output, STDOUT. All other strings are taken as argument strings to be passed to main.

RETURNS

_main returns the boolean value obtained from the main call, which is then discarded by the current startup code.

EXAMPLE

To avoid the loading of _main and some of the RMS code it calls on, one can provide a substitute _main, instead of the normal main:

```
COUNT _main()
    {
    if (open("TI:", READ, 0) != STDIN
        || create("TI:", WRITE, 0) != STDOUT
        || create("TI:", WRITE, 0) != STDERR)
        exit(NO);
    <program body>
    }
```

SEE ALSO

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.

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

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 version of a file of the specified name. 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, writing, and extending).

If (rsize == 0), the file has variable length records with an explicit carriage return, linefeed sequence at the end of each line; lines may be up to 510 characters long and are each terminated by a linefeed (newline) when output to the file. The newline is taken to represent the carriage return, linefeed sequence. Otherwise if (rsize != 0) characters are input and output unchanged and the file is taken as a sequence of unstructured 512-byte logical blocks.

RETURNS

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

EXAMPLE

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

SEE ALSO

close, open, remove, uname

NAME

emt - make a system call

SYNOPSIS

```
COUNT emt(code, arg1, ...)
    COUNT code;
    ...
or
COUNT emt(pdpb)
    struct dpb *pdpb;
```

FUNCTION

emt performs an 'EMT 0377' system call to P/OS, using the arguments arg1, ... to the call as the Directive Parameter Block. No checking whatsoever is made for reasonable values of code, which must contain a valid Directive Identification Code in its lower byte and a Block size in its upper byte; nor are the arguments checked for number, type, or validity. Thus it is possible to perform an arbitrary system call from C.

It should be emphasized that C functions cannot be connected directly to Asynchronous System Traps, since the calling sequences are incompatible; the interface routine entrap should be used as an intermediary.

RETURNS

emt returns the Directive Status Word returned by P/OS on each system call. By fairly uniform system convention, the value is negative if and only if something went wrong.

SEE ALSO

entrap, fcall

NAME

entrap - enter function on system trap

SYNOPSIS

```
struct ast {
    COUNT jsr_r1, (*entrap)(), code, (*fn)();
    } ast {
    04137, &entrap, code, &fn};
```

FUNCTION

entrap is a general interface routine for entering any Synchronous or Asynchronous System Trap. It expects to be called, when a trap occurs, by the sequence:

```
jsr     r1,#entrap
.word   code, fn
```

where code is an integer specifying the number of words to be removed from the stack before returning from the trap, and fn is the address of the C function to be called when the trap occurs; code is negative if an SST occurs (-5 for memory protect, -3 for EMT, etc.), positive for an AST (0 for power recovery, 2 for floating exception, etc.).

fn is called with the arguments:

```
fn(code, r0, r1, ...)
```

where code is the same as for the entrap call, r0 and r1 are the register contents when the trap occurs, and ... is whatever else P/OS puts on the stack for the particular trap. This call is unusual in that the arguments are used on return from the function fn() to restore the interrupted environment, so they should not be altered unless by careful design.

The "calling sequence" described above under SYNOPSIS is intended for use with an emt call that specifies an ast entry point. Using &ast for the entry point will cause entrap to be called when the trap occurs, and hence the C function. This permits entrap to be used with an arbitrary number of system traps.

RETURNS

entrap restores r0 and r1 from the arguments sent to fn, uses code to pop the stack as described above, and returns to the interrupted program. Return is via an RTI instruction for SSTs, or via an ASTX system call for ASTs. If the ASTX is rejected, entrap exits to the system.

EXAMPLE

To specify a receive-by-reference AST:

```
if (emt(287, &ast) < 0)
    putstr(STDERR, "can't SRRA!\n", NULL);
```

SEE ALSO

emt, fcall

BUGS

No attempt is made to save volatile floating registers.

None of this stuff works right if I-space and D-space are separate.

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. The success code is currently ignored.

RETURNS

exit will never return to the caller.

EXAMPLE

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

SEE ALSO

onexit

NAME

fcall - call a Professional library routine

SYNOPSIS

```
TYPE fcall(fn, nargs, arg1, ...)
    TYPE (*fn)();
    COUNT nargs;
    ...
```

FUNCTION

fcall is a generic interface function for calling Professional library routines. fn is the Professional library function or subroutine to call, nargs is the number of arguments, and arg1, ... are the arguments to be passed to fn on the call. TYPE is whatever type of value fn returns, if any.

Note that Professional library functions expect to be passed the addresses of their arguments, not simple rvalues.

If a C function is called from a Professional library routine, the arguments can be picked up by the following ruse:

```
cfn(p)
    struct {
        int nargs;
        int *arg[1];
        } *p;
/* (&p)[-2]->nargs gives the number of arguments
 * (&p)[-2]->arg[n] gives pointer to arg n
 */
```

Integer values may be returned in the normal manner.

RETURNS

fcall returns whatever fn returns.

EXAMPLE

To call the library subroutine proc with the single argument x:

```
IMPORT VOID proc();
    ...
fcall(&proc, 1, &x);
```

Or equivalently, one could use the lines

```
#define proc(arg1)  fcall(&proc, 1, &(arg1))
IMPORT VOID proc();
```

so as to be able to write

```
proc(x);
```

BUGS

In passing back the return value of fn, fcall returns REALs as C longs, and INTEGER * 4, REAL * 8, or COMPLEX cannot be returned properly.

NAME

lseek - set file read/write pointer

SYNOPSIS

```
FILE 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. RMS uses all 32 bits of the offset.

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. A file must be created or opened with a non-zero rsize for lseek calls to be acceptable.

RETURNS

lseek returns the file descriptor 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 (fread(STDIN, buf, 512) != 512);
    }
```

BUGS

It doesn't check for illegal values of offset.

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

```
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 is supposed to ensure 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.)

On this system, onintr is currently a dummy, so pfn is never called.

RETURNS

Nothing.

NAME

open - open a file

SYNOPSIS

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

FUNCTION

open opens a file of specified name 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 (rsize != 0) characters are input and output unchanged from 512-byte logical blocks. Otherwise, only sequential access is permitted, carriage returns, NULs, and formfeeds are discarded on input, and newline is taken as end of record on output.

RETURNS

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

EXAMPLE

```
if ((fd = open("xeq", WRITE)) < 0)
    putstr(STDERR, "can't open xeq\n", NULL);
```

SEE ALSO

close, create

NAME

read - read 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 opened or created with (rsize != 0) characters are input and output unchanged from 512-byte logical blocks. Otherwise carriage returns, NULs, and formfeeds are deleted on input.

RETURNS

If an error occurs, read returns -1; if end-of-file is initially encountered, read returns zero; otherwise the value returned is between 1 and size, inclusive. When reading from a disk file, size bytes are read whenever possible.

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 deletes the file fname from the file system.

RETURNS

remove returns zero, if successful, or -1.

EXAMPLE

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

NAME

sbreak - set system break

SYNOPSIS

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

FUNCTION

sbreak moves the system break, at the top of the task, algebraically up by size bytes, rounded up to a multiple of 64 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))))
    {
    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 null 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 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, 1)) < 0)
    putstr(STDERR, "can't create sort temp\n", NULL);
```

SEE ALSO

close, create, open, remove

NAME

write - write to a file

SYNOPSIS

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

FUNCTION

write writes size characters starting at buf to the file specified by fd. If the file was opened or created with (rsize != 0) characters are output unchanged to 512-byte logical blocks. Otherwise each newline (linefeed) is expanded to a carriage return, linefeed sequence and is taken as an end of record indicator.

RETURNS

If an error occurs, write returns -1; otherwise the value returned should be size.

EXAMPLE

To copy a file:

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

SEE ALSO

read