

### III.c. RSX-11M System Interface Library

III.c - 1	<b>Interface</b>	to RSX-11M system
III.c - 3	<b>Conventions</b>	RSX-11M system subroutines
III.c - 5	c	compiling C programs
III.c - 6	pc	compiling Pascal programs
III.c - 7	ctkb	task building C programs
III.c - 9	_main	setup for main call
III.c - 10	_pname	program name
III.c - 11	close	close a file
III.c - 12	create	open an empty instance of a file
III.c - 13	emt	make a system call
III.c - 14	entrap	enter function on system trap
III.c - 16	exit	terminate program execution
III.c - 17	fcall	call a Fortran program
III.c - 18	lseek	set file read/write pointer
III.c - 19	onexit	call function on program exit
III.c - 20	onintr	capture interrupts
III.c - 21	open	open a file
III.c - 22	rad50	convert ASCII to rad50
III.c - 23	read	read from a file
III.c - 24	remove	remove a file
III.c - 25	sbreak	set system break
III.c - 26	uname	create a unique file name
III.c - 27	write	write to a file

**NAME**

Interface - to RSX-11M system

**FUNCTION**

Programs written in C for operation on the PDP-11 under RSX-11M 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 - RSX-11M system subroutines

**SYNOPSIS**

```
#include <rsx.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, <rsx.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:

```
EFN - 52, the standard event flag number
FACRE - 010
FAEXT - 04
FANSP - 0100
FARD - 01
FASHR - 040
FATMP - 020
FAWRT - 02
FDB - struct fdb, the standard RSX file descriptor block
FDBLK - 010
FDCLL - 02
FDCLR - 02
FDDIR - 010
FDFTN - 01
FDINS - 010
FDPLC - 04
FDRAN - 02
FDREC - 01
FDRWM - 01
FDSD1 - 020
FDSQD - 040
FDTTY - 04
FNB - struct fnblk, the standard filename block
IE_ADP - -0142
IE_EOF - -10
IE_IEF - -0141
IE_ILU - -0140
IE_SDP - -0143
IE_ULN - -5
IE_UPN - -1
IOS - struct ios, the standard I/O status block
IO_ATT - 01400
IO_DAC - 010000
IO_DET - 02000
IO_KIL - 00012
IO_RLB - 01000
IO_RVB - 010400
IO_WLB - 00400
IO_WVB - 011000
```

**Conventions**

- 2 -

**Conventions**

IS\_SUC - 01  
NBDEV - 0200  
NBDIR - 0100  
NBNAM - 04  
NBSD1 - 0400  
NBSD2 - 01000  
NBSNM - 040  
NBSTP - 020  
NBSVR - 010  
NBTYP - 02  
NBVER - 01  
NFILES - 10  
QIOW - 06003  
RFIX - 1  
RSEQ - 3  
RVAR - 2

TTYLUN - 5, the standard LUN for teletype I/O

**SEE ALSO**

RSX I/O manuals.

**NAME**

c - compiling C programs

**SYNOPSIS**

```
>@lb:[1,1]c  
*FILE[S]: 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 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 c. The object file output from MACRO-11 is put at sfile.obj.

**EXAMPLE**

To compile test.c:

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

**SEE ALSO**

pc

**FILES**

ctmp1.tmp, ctmp2.tmp, ctmp3.tmp

**BUGS**

There should be some way to pass the -m flag to cp1, or some of the myriad flags acceptable to cpp and cp2, other than by modifying the command file proper.

pc

### III.c. RSX-11M System Interface Library

pc

#### NAME

pc - compiling Pascal programs

#### SYNOPSIS

```
>@lb:[1,1]pc  
*FILE[S]: 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 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 pc. The object file output from MACRO-11 is put at sfile.obj.

#### EXAMPLE

To compile test.p:

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

#### SEE ALSO

c

#### FILES

ctmp1.tmp, ctmp2.tmp, ctmp3.tmp

#### BUGS

There should be some way to pass the -m flag to cp1, or some of the myriad flags acceptable to ptc, cpp and cp2, other than by modifying the command file proper.

**NAME**

ctkb - task building C programs

**SYNOPSIS**

```
>TKB
TKB>file/cp=<files>
TKB>LB:[1,1]CHDR,LB:[1,1]CLIB/LB
TKB>/
TKB>UNITS=12
TKB>STACK=3000
TKB>//
```

**FUNCTION**

Programs written in C must be linked with certain object modules that implement the standard C runtime environment. The normal RSX-11M TKB program is used, but with several important constraints. These are shown in the example above, which produces an executable file "file.tsk" from one or more object files in the list <files> that presumably were produced by the C compiler.

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. Since logical unit numbers 5 and 6 tend to be reserved for console I/O, twelve units must be assignable to achieve the full capacity of the runtime.

chdr.obj is the module that gets control when file.tsk is run. It calls the function `_main()`, described in the RSX-11M 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 <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 RSX-11M Interface Manual. 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.

clib is a library containing all of the portable C library functions, plus those required for the RSX-11M interface. It should be scanned last.

Note that the task constructed by TKB must be installed before it is possible to feed a command line to it on execution.

**EXAMPLE**

To compile and run the program echo.c:

ctkb

- 2 -

ctkb

```
>ec  
*FILE[S]: ECHO  
>TKB  
TKB>ECHO/CP=ECHO  
TKB>LB:[1,1]CHDR,LB:[1,1]CLIB/LB  
TKB>/  
TKB>UNITS=12  
TKB>STACK=3000  
TKB>/  
>INSTALL ECHO/TASK=...ECH  
>ECHO HELLO WORLD!  
hello world!
```

SEE ALSO

c

**\_main****III.c. RSX-11M System Interface Library****\_main****NAME**

`_main` - setup for main call

**SYNOPSIS**

`BOOL _main()`

**FUNCTION**

`_main` is the function called whenever a C program is started. It obtains a command line, if possible, translates it into lowercase, 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 all the FCS code it calls on, one can provide a substitute `_main`, instead of the normal `main`:

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

**SEE ALSO**

`exit`

**BUGS**

The task must be installed before it can be fed a command line. Even worse, feeding a command line to a task from another task is an intricate and privileged process.

\_pname

### III.c. RSX-11M System Interface Library

\_pname

#### 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

**close**

**III.c. RSX-11M System Interface Library**

**close**

**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, which is -1 or the RSX error return code.

**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**

**create****III.c. RSX-11M System Interface Library****create****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 assumed carriage return/linefeed at the end of each line; lines may be up to 510 characters long and are each terminated by a linefeed (newline) on output, which is taken to represent the carriage return/linefeed. Otherwise if (rsize != 0) characters are input and output unchanged.

**RETURNS**

create returns a file descriptor for the created file or a negative number, which is -1 or the RSX error return code.

**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 RSX, 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.

Note, however, that GMCR reads the command line straight into the DPB, so the second form should always be used with this call. It should also 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 RSX on each system call. By fairly uniform system convention, the value is negative if and only if something went wrong.

**EXAMPLE**

To read a command line and complain if not there:

```
struct dpb {
    char dic, size, cbuf[80];
} gcmldpb {127, 41};

if ((n = emt(&gcmldpb)) < 0)
{
    emt(QIOW, IO_WVB, TTYLUN, EFN, &iobs, 0,
        "help!\r\n", 7, 0, 0, 0, 0);
    emt(0463); /* i.e., exit */
}
```

**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 RSX 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

entrap

- 2 -

entrap

**BUGS**

No attempt is made to save volatile floating registers.

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

**exit****III.c. RSX-11M System Interface Library****exit****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

**fcall**

**III.c. RSX-11M System Interface Library**

**fcall**

**NAME**

**fcall - call a Fortran program**

**SYNOPSIS**

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

**FUNCTION**

**fcall** is a generic interface function for calling Fortran programs. **fn** is the Fortran 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.

Note that Fortran expects arguments to be addresses, not values.

If a C function is called from Fortran, 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**

```
fcall(&func, 1, &x);
```

**BUGS**

REALs are returned as C longs, and INTEGER \* 4, REAL \* 8, or COMPLEX cannot be returned properly.

**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. RSX uses only the low order 25 bits of the offset; the rest are ignored.

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. In general, a file must have been created with a non-zero rsize for lseek calls to be acceptable.

**RETURNS**

lseek returns the file descriptor if successful, or a negative number, which is -1 or the RSX error return code.

**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**

lseek doesn't check for illegal values of offset.

onexit

### III.c. RSX-11M System Interface Library

onexit

#### 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.

**onintr**

**III.c. RSX-11M System Interface Library**

**onintr**

**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.

**open**

**III.c. RSX-11M System Interface Library**

**open**

**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).

**rsize** is ignored, but should be present for portability.

**RETURNS**

**open** returns a file descriptor for the created file or a negative number, which is -1 or the RSX error return code.

**EXAMPLE**

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

**SEE ALSO**

**close, create**

rad50

III.c. RSX-11M System Interface Library

rad50

**NAME**

rad50 - convert ASCII to rad50

**SYNOPSIS**

```
COUNT rad50(s, n)
    TEXT *s;
    COUNT n;
```

**FUNCTION**

rad50 converts up to three characters of the string starting at s into a packed word containing three "radix 50" characters. If n is less than three, the string is effectively padded on the right by spaces; a zero or negative count n yields a word of all spaces, i.e., zero.

The valid rad50 characters consist of the space ' ', period '.', dollar sign '\$', digits '0' through '9', and letters in either case, 'a' through 'z' or 'A' through 'Z'. All other characters are mapped into the illegal rad50 character.

**EXAMPLE**

To convert an RT-11 style filename:

```
n = scnstr(fn, ':');
if (fn[n])
{
    dblk[0] = rad50(fn, n);
    fn += n + 1;
}
else
    dblk[0] = 0;
n = scnstr(fn, '.');
dblk[1] = rad50(fn, n);
dblk[2] = rad50(fn + 3, n - 3);
dblk[3] = (fn[n] == '.') ? rad50(fn + n + 1, lenstr(fn + n + 1)) : 0;
```

read

### III.c. RSX-11M System Interface Library

read

#### 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 has fixed length records, succeeding records are unpacked with no interpretation. Otherwise if the file is connected to a record device, such as a teletype, carriage returns are converted to newlines ('\n' or linefeed). Otherwise if the file is unstructured (RSTS), carriage returns and nulls are deleted on input. Otherwise if the file has the FDCR attribute, a newline is returned at the end of each record.

#### RETURNS

If an error occurs, read returns a negative number which is -1 or the RSX error return code; if end of file is 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

**remove**

**III.c. RSX-11M System Interface Library**

**remove**

**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 a negative number, which is -1 or the RSX error return code.

**EXAMPLE**

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

**sbreak**

**III.c. RSX-11M System Interface Library**

**sbreak**

**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);
}
```

**BUGS**

A kludge version of **sbreak** must be used with IAS, since that system doesn't support the EXTK directive.

uname

III.c. RSX-11M System Interface Library

uname

**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

**write**

**III.c. RSX-11M System Interface Library**

**write**

**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 has fixed length records, the characters are packed into succeeding records with no interpretation. Otherwise if the file is unstructured (RSTS) each newline is preceded by a carriage return. Otherwise, the newline character ('\n', or linefeed) is taken as a record terminator. If the file is connected to a record device, such as a teletype, each newline is output as a carriage return, linefeed sequence. Otherwise if the file has the FDCR attribute, newlines are discarded by **write**; variable length records more than 510 characters long are folded at 510 characters.

**RETURNS**

If an error occurs, **write** returns a negative number which is -1 or the RSX error return code. 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**