III.b. OS System Interface Library

Intro	Introduction to the CMS and OS system interface.	III.b - 1
Usage	using the compiler under CMS and OS.	III.b - 2
CC	compiling C or Pascal programs under CMS and OS.	III.b - 10
Files	under CMS and OS	III.b - 16
Errors	CMS and OS system subroutines.	III.b - 22
Upper	case support	III.b - 23
ASCII	character representation	III.b - 24
EBCDIC	the EBCDIC version of the compiler	III.b - 25
Interface	to CMS and OS.	III.b - 27
Echo	documentation and source	III.b - 30
_atoe	translation table from ASCII to EBCDIC	III.b - 32
_cmpswt	set CMS compiler switch.	III.b - 33
_cmsl	issue CMS commands with argument list.	III.b - 34
_cmsv	issue CMS commands with argument vector.	III.b - 35
_etoa	translation table from EBCDIC to ASCII	III.b - 36
_iscms	determine if running under CMS	III.b - 37
_ists	determine if running interactively	III.b - 38
_main	setup for main call.	III.b - 39
_paths	the search path.	III.b - 40
_pname	program name	III.b - 41
_stksiz	size of the stack.	III.b - 42
_svc	call OS.	III.b - 43
_svc202	call CMS	III.b - 44
close	close a file	III.b - 45
create	open an empty instance of a file	III.b - 46
exit	terminate program execution.	III.b - 47
exitrc	terminate program execution with return-code	III.b - 48
fcall	call a Fortran program	III.b - 49
fcalld	call a Fortran program and return DOUBLE	III.b - 50
fcalli	call a Fortran program and return integer.	III.b - 51
lseek	set file read/write pointer.	III.b - 52
mkexec	make file executable	III.b - 53
onexit	call function on program exit.	III.b - 54
onintr	capture interrupts	III.b - 55
open	open a file.	III.b - 56
read	read from a file	III.b - 57
remove	remove a file.	III.b - 58
sbreak	set system break	III.b - 59
uname	create a unique filename	III.b - 60
write	write to a file.	III.b - 61
xecl	execute a file with argument list.	III.b - 62
xecv	execute a file with argument vector.	III.b - 63
xlate	translate a buffer of characters	III.b - 64

NAME

Intro - Introduction to the CMS and OS system interface

FUNCTION

Two classes of operating systems are currently supported with the C/370 compiler: VM/CMS (release 2 and upwards) and OS/VS (ranging from MFT to MVS).. As CMS can simulate most of the functions in OS, the system interface is the same for both classes of operating systems. However, the interface also contains support for operations that are unique to CMS.

This section contains information about the CMS and OS specific details of the C/370 compiler. Its main parts are:

Usage This section describes the basic use of the compiler. It deals with compiling, linking and running a simple C program. It also contains examples of the CMS commands and JCL statements that are required, as well as two examples of C programs that copy files.

CC The section on the compiler driver **CC** describes the syntax for the arguments and their meaning. The **ddnames** used by the compiler are also listed.

Files When you're ready to do I/O on files other than the standard input, output and error files you should read **Files**. This section discusses text and binary filetypes and the so called "I/O drivers" in the interface.

Errors Explains the negative values returned by certain system interface functions.

UPPER A discussion of problems facing users who must write C code in uppercase, and the limited support available to such users via the "JAPAN" option.

ASCII A table showing the positions in EBCDIC of some ASCII characters is presented. The table also shows which EBCDIC characters the compiler expects as input. See also **Usage** for problems with the character representation.

EBCDIC The EBCDIC version of the compiler is discussed here.

Interface A description of what happens with external identifiers, function text, literal and initialized data, and uninitialized declarations, how function calls are performed and how the stack is maintained.

Echo The documentation and source code for the C program **echo**.

Functions The rest of the section consists of manual pages, one for each function in the system interface.

The Whitesmiths' portable specification of the interface to the operating system is described in the C Programmer's Manual, section III.

NAME

Usage - using the compiler under CMS and OS

FUNCTION

The following two paragraphs describe the operating system dependent aspects of the use of the compiler and the compiled programs.

USING THE COMPILER UNDER CMS

This section describes how to compile, link and execute a C program. As an example of a C program we use the program **echo**, which is listed in Section III.b of this manual. The **echo** program simply echoes its argument to its "standard output".

Editing a C program under CMS

When editing a C program you can use any record-format and record-length you like.

If you prefer to use sequence-fields in your programs, you can put them in any column, but you must ask the compiler driver, **CC**, to remove them (by using the SEQFLD parameter) before handing the source to the compiler. This does not apply for **#include**'d files.

As you may already have noticed, the C language uses a lot of characters. You'll probably have difficulties in finding some of them on your terminal, and even if you find them, they may have different positions in the EBCDIC alphabet than the positions that the compiler expects. To cope with this problem you can either

- use an alternate representation of the character as found in the C Programmer's Manual (see "Punctuation" under Syntax in Section I). For example, you can use (< instead of { and (| instead of [.
- let CMS translate the character to another one. This is done with the **SET INPUT c xx** and **SET OUTPUT xx c** commands. For instance, if you can't type in a backslash \ you can change the "commercial at" **@** to the backslash character:

```
set input @ e0
set output e0 @
```

- change the translation tables **_etoa** and **_atoe**, replace the old ones in the C library and relink the compiler.

Compiling a C program under CMS

To compile the echo program (assuming it is stored as 'ECHO C A'), type:

```
cc echo
```

The compiler driver **cc** runs the three compiler passes, the Whitesmiths' assembler, and the program generating the object module. The object module is stored as 'echo text'.

In case compiler diagnostics are produced, they will be written to your terminal unless you have given the PRINT or DISK options.

Loading and generating modules under CMS

Before you can run your program, it must be linked together with other, implicitly- or explicitly-called functions. Under CMS this is done with the **load** command. To identify for the **load** command the library where these functions are stored, use the **global** command in the following manner:

```
global txtlib clib
```

Now you can load the program with:

```
load echo
```

An executable module is created by the **genmod** command:

```
genmod echo
```

Executing a C program under CMS

Now you have a program that can be executed just by giving its name and arguments like this:

```
echo hello world!
```

Compiling C programs to be stored in libraries

When you want to store C object modules in a library (i.e. TXTLIB), you should use the NOSTART and LIBGEN options to **CC**. NOSTART prevents the code-generator from generating the (harmless but unnecesary) branch to C#START, LIBGEN causes TOBJ (the last phase during compilation) to emit NAME and ALIAS statements in the object module.

If you want to change the default value that is used when the stack is allocated, for example, you should compile the C file 'STKSIZ C A':

```
/*  THIS NUMBER IS USED IN START WHEN ALLOCATING THE STACK
 *  copyright (c) 1983, 1984, 1985 by Whitesmiths, Ltd.
 */
#include <std.h>

GLOBAL BYTES _stksiz = 30000; /* modified 01/27/85 by PT */
```

Compile the C file with:

```
cc stksiz ( nostart libgen
```

You must then delete the old object file in the C library with:

txtlib del clib $$stksiz

This is because the TXTLIB command has no provision for replacing members in libraries (even if " NAME MBR(R) " is used). Then add the new C object file to the library with:

txtlib add clib stksiz

Note that the name used in the first command is the member name in the library; the second is the file name.

To create a library out of C object modules compiled with NOSTART and LIBGEN, you can use the COPY command in the following manner to create an input file to the TXTLIB command:

copy * text a myobj text a (append)

(Remember to erase 'MYOBJ TEXT A' before you reissue the **copy** command.)

To create 'MYLIB TXTLIB A':

txtlib gen mylib myobj

Using the C I/O system under CMS

As an example of how to specify filenames to **open** and **create** we give you the following (not very) general file-copy utility:

```
/*  COPY CMS-FILE 'INPUT FILE A' TO 'FILE OUTPUT B' AS TEXT-FILES
 */
#include <std.h>

BOOL main()
    {
    FAST FILE ifd, ofd;
    FAST COUNT n;
    TEXT buf[BUFSIZE];

    if ((ifd = open("input.file", READ, 0)) < 0)
        error("cannot open ", "input file");
    if ((ofd = create("b:output", WRITE, 0)) < 0)
        error("cannot create", "file output b");
    while ((n = read(ifd, buf, sizeof buf)) > 0)
        if (write(ofd, buf, n) != n)
            error("write error", NULL);
    if (n < 0)
        error("read error", NULL);
    return (YES);
    }
```

A somewhat more sophisticated program that takes two flags for specifying the "record-size"s (i.e., to distinguish between text and binary files),

another flag to specify the buffer size and two arguments for the input and output filenames is listed later after the OS section below.

If you enter this program as 'FILECOPY C A', type:

```
cc filecopy
global txtlib clib
load filecopy
genmod filecopy
```

You can, for instance, use this program to copy the source program to your terminal with:

```
filecopy filecopy.c con:
```

or directly down the drain (in verbose mode) with:

```
filecopy -v filecopy.c null:
```

Use this program to find out more about the I/O system.

USING THE COMPILER UNDER OS

This section describes how you compile, link and execute a C program. As an example of a C program we use the program **echo** which is listed in Section III.b in this manual. The **echo** program simply echoes its arguments to its "standard output". We also assume that the program is stored in a cataloged dataset named 'SMITH.TESTC.ECHO.C', the header files are stored in 'SYS1.C370.MACLIB' and the C library is named 'SYS1.C370.CLIB'. The JOB statements must also be modified to conform to your local standards.

Compiling a C program under OS

To compile the echo program, submit the following JCL:

```
//SMITH   JOB  (...),'COMPILE ECHO',REGION=512K
//JOBLIB   DD  DSNAME=SYS1.C370.LINKLIB,DISP=SHR
//COMP   EXEC  PGM=CC
//SYSIN    DD  DSNAME=SMITH.TESTC.ECHO.C,DISP=SHR
//SYSLIB   DD  DSNAME=SYS1.C370.MACLIB,DISP=SHR
//SYSLIN   DD  DCB=(BLKSIZE=800,LRECL=80,RECFM=FB),
//             DISP=(NEW,CATLG),DSNAME=SMITH.TESTC.ECHO.OBJ,
//             SPACE=(TRK,(1,1)),VOL=SER=PUB001,
//             UNIT=3350
//SYSTERM  DD  SYSOUT=*
```

The compiler driver **CC** runs the three compiler passes, the Whitesmiths' assembler and the program to generate the object module. The object module is stored in the dataset 'SMITH.TESTC.ECHO.OBJ'.

Linking a C program under OS

The following JCL will link the **echo** program. The executable program will be stored in the partitioned data set 'SMITH.TESTC.ECHO.LOADMOD'.

```
//SMITH   JOB  (...),'LINK ECHO',REGION=512K
//JOBLIB   DD  DSNAME=SYS1.C370.LINKLIB,DISP=SHR
//LINK   EXEC  PGM=LINKEDIT
//SYSLIB   DD  DSNAME=SYS1.CLIB,DISP=SHR
//SYSLIN   DD  DSNAME=SMITH.TESTC.ECHO.OBJ,DISP=SHR
//SYSLMOD  DD  DISP=(NEW,CATLG),
//             DSNAME=SMITH.TESTC.ECHO.LOADMOD(ECHO),
//             SPACE=(TRK,(1,1,1)),VOL=SER=PUB001,
//             UNIT=3350
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  UNIT=SYSSQ,SPACE=(CYL,(1,1))
```

Executing a C program under OS

Throughout this documentation, a simple notation is used to show how a program is started, and how parameters are passed to it. This notation consists of the program name followed by a series of parameters. Each element on this "command line" is delimited by whitespace:

<program name> <parameters>

On most systems, programs can be run using more or less this kind of notation. Under OS, this notation must be translated to:

// EXEC PGM=<program name>,PARM='<parameters>'

To execute the newly linked **echo** program, use the following JCL:

```
//SMITH   JOB  (...),REGION=512K
//JOBLIB   DD  DSNAME=SMITH.TESTC.ECHO.LOADMOD,DISP=SHR
//       EXEC  PGM=ECHO,PARM='HELLO WORLD!'
//SYSPRINT DD  SYSOUT=*
```

Compiling C programs to be stored in libraries

When you want to store C object modules in a library (i.e. link library), you should use the NOSTART and LIBGEN options to **CC**. NOSTART prevents the code-generator from generating the (harmless but unnecesary) branch to C#START. LIBGEN causes TOBJ (the last phase during compilation) to emit NAME and ALIAS statements in the object module.

If you just have written a C function called **_ists()** that determines whether you are running under TSO or batch, you can use the following JCL to replace the one supplied with the compiler (which is a dummy that returns **_iscms()**) (the C file is stored in the partitioned dataset 'SMITH.CFUNCS.SOURCE' as the member ISTS):

```
//SMITH   JOB  (...),'REP ISTS',REGION=512K
//JOBLIB   DD  DSNAME=SYS1.C370.LINKLIB,DISP=SHR
```

```
//COMP   EXEC  PGM=CC,PARM=(NOSTART,LIBGEN)
//SYSIN    DD  DSNAME=SMITH.CFUNCS.SOURCE(ISTS),DISP=SHR
//SYSLIB   DD  DSNAME=SYS1.C370.MACLIB,DISP=SHR
//SYSLIN   DD  UNIT=SYSSQ,SPACE=(TRK,(1,1)),DISP=(NEW,PASS)
//SYSTERM  DD  SYSOUT=*
//LINK   EXEC  PGM=LINKEDIT,PARM=NCAL,COND=(0,LT,COMP)
//SYSLIN   DD  *.COMP.SYSLIN,DISP=(OLD,DELETE)
//SYSLMOD  DD  DSNAME=SYS1.C370.CLIB,DISP=OLD
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  UNIT=SYSSQ,SPACE=(CYL,(1,1))
```

Using the C I/O system under OS and TSO

As an example of how to specify filenames to **open** and **create** we give you the following (not very) general file copy utility.

```
//SMITH   JOB  (...),'HELLO',REGION=512K
//JOBLIB   DD  DSNAME=SYS1.C370.LINKLIB,DISP=SHR
//COMP   EXEC  PGM=CC
//SYSIN    DD  DATA,DLM='##'
/*  COPY OS-FILE 'INPUT' TO 'WTP:' AS TEXT-FILES
 */
#include <std.h>

BOOL main()
    {
    FAST FILE ifd, ofd;
    FAST COUNT n;
    TEXT buf[BUFSIZE];

    if ((ifd = open("input", READ, 0)) < 0)
        error("cannot open ", "input");
    if ((ofd = create("wtp:", WRITE, 0)) < 0)
        error("cannot create ", "wtp:");
    while ((n = read(ifd, buf, sizeof buf)) > 0)
        if (write(ofd, buf, n) != n)
            error("write error", NULL);
    if (n < 0)
        error("read error", NULL);
    return (YES);
    }
##
//SYSLIB   DD  DSNAME=SYS1.C370.MACLIB,DISP=SHR
//SYSLIN   DD  UNIT=SYSSQ,SPACE=(TRK,(1,1)),DISP=(NEW,PASS)
//SYSTERM  DD  SYSOUT=*
//GO     EXEC  PGM=LOADER,PARM=(NOPRINT,TERM),COND=(0,LT,COMP)
//SYSLIB   DD  DSNAME=SYS1.C370.CLIB,DISP=SHR
//SYSLIN   DD  *.COMP.SYSLIN,DISP=(OLD,DELETE)
//SYSPRINT DD  SYSOUT=*
//SYSTERM  DD  SYSOUT=*
//SYSUT1   DD  UNIT=SYSSQ,SPACE=(CYL,(10,10))
//INPUT    DD  *
Hello world!
/*
```

Use the more sophisticated file copy program listed below to get aquainted with I/O operations under OS and TSO. It is, for instance, an excellent tool for converting fixed length records to variable length ones. Under OS you must (still) use DD-statements when you refer to datasets. If you are logged on as the TSO user SMITH, you can use file copy program (called **filecopy**) to list the header file **<std.h>** with the following commands:

```
allocate file(input) dataset('sys1.c370.maclib(std)')
call cprogs(filecopy) 'input term:'
```

A SOMEWHAT MORE SOPHISTICATED FILE COPY PROGRAM

```
/*  FILE COPY
 */
#include <std.h>

GLOBAL BYTES ib = 0, ob = 0;
GLOBAL BYTES bs = BUFSIZE;
GLOBAL BOOL vflag = NO;

BOOL main(ac, av)
    COUNT ac;
    TEXT **av;
    {
    IMPORT TEXT *_pname;
    FILE ifd, ofd;
    FAST COUNT nr, nw;
    FAST TEXT *buf;

    getflags(&ac, &av, "bs#,ib#,ob#,v:F <infile> <outfile>",
        &bs, &ib, &ob, &vflag);
    if (ac != 2)
        usage("takes two arguments: <infile> <outfile>\n");
    buf = alloc(bs, NULL);
    if ((ifd = open(av[0], READ, ib)) < 0)
        error("cannot open input file ", av[0]);
    if ((ofd = create(av[1], WRITE, ob)) < 0)
        error("cannot create output file ", av[1]);
    while ((nr = read(ifd, buf, bs)) > 0)
        {
        if ((nw = write(ofd, buf, nr)) != nr)
            {
            errfmt("%p: write error, code %i\n", _pname, nw);
            exit(NO);
            }
        if (vflag)
            putfmt("copied %i byte%p...\n", nw, (nw > 1) ? "s" : "");
        }
    if (nr < 0)
        {
        errfmt("%p: read error, code: %i\n", _pname, nr);
        exit(NO);
```

```
            }
        return (YES);
        }
```

SEE ALSO
    Files

NAME

CC - compiling C or Pascal programs under CMS and OS

SYNOPSIS

Under CMS:

```
cc fn [( blank separated list of parameters]
```

Under OS:

```
// EXEC PGM=CC,PARM='comma separated list of parameters'
```

FUNCTION

The program CC is provided to make it easier to use the Whitesmiths' C and Pascal compilers. It accepts parameters in a form similar to that used by IBM compilers.

CC optionally produces a numbered source listing of the source program on SYSTERM or SYSPRINT and then invokes the compiler passes PTC (Pascal only), PP, P1 and P2@370, optionally followed by a call to AS@370 and TOBJ.

CC generally stays quiet if no listings are requested and no diagnostics are produced.

The parameters are:

(NO)AS370 - use the Whitesmiths' assembler and object module conversion program to generate the object module. If NOAS370, the system assembler is invoked. Default is AS370.

(NO)ASM - invoke an assembler to produce an object deck on SYSLIN. When NOASM is specified the assembler text is written to SYSPUNCH (which is filedef'd under CMS, see below), and no object module is generated. See also the AS370 and ASMG options. Default is ASM.

(NO)ASMG - use the program ASMGASM when invoking the assembler. If ASMG is not specified, the standard OS assembler, IFOX00, is invoked instead. If the AS370 option is in effect, this parameter is ignored. Default is NOASMG.

BAseregs=# - specify the number of additional base-registers used in each function. For normal-sized functions, and even quite large ones, a single base-register, BASEREGS=0, is sufficient. Each additional base-register will permit the functions to be 4096 bytes larger. If the assembler complains about addressability errors, you should try this flag. Default is BASEREGS=0. This flag is passed to **p2**.

(NO)CAsedistinctions - ignore case distinctions in testing external identifiers for equality, and map all names to lowercase on output from P1 (which is translated to uppercase by TOBJ). By default, case distinctions don't matter. This flag is passed to **p1**.

(NO)CHeck - enable stack overflow checking. Default is CHECK. This flag is passed to **p2**.

CSect - produce a named CSECT as output file. The name is chosen as follows: If the external symbol **_main** is found in the text section it is used, otherwise the first external symbol is used. The first character in this symbol is replaced by the commercial at character, **@**. This parameter is ignored in case NOAS370 is specified. Default is CSECT. (This flag is passed to TOBJ.)

Define=(#,#...) - where * has the form **name=def**. Define **name** with the definition string **def** to **pp**; if **=def** is omitted, the definition is taken to be "1". Up to ten definitions may be entered in this fashion. This flag is passed to **pp**.

(NO)DIsk [CMS only] - direct the SYSTERM output to "fn clisting" and SYSPRINT output to "fn listing" under CMS. Default is NODISK.

(NO)Ebcdic - compile the program in EBCDIC mode. When this option is used, **CC** passes a flag to **pp**, causing it to read a translation table from the file with the **DDname** ATOE. Under CMS, ATOE is filedef'd to 'ATOE TAB *'. It also passes a flag to P1, causing declarations of type **char** to be **unsigned char**. A program compiled with the EBCDIC option must be linked with library functions compiled in EBCDIC. Default is NOEBCDIC.

(NO)ISO [Pascal only] - disallow pointer types defined using type identifiers from outer blocks, i.e., require the type pointed at to be defined in the same set of type declarations as the pointer type itself. Default is ISO. This flag is passed to **ptc**.

(NO)LADad [CMS only] - disallow the use of the built-in auxilliary directory in **CC**. Useful if the system disk has been changed and **CC** hasn't. The default depends on if **CC** was generated with an auxilliary directory or not.

LC=# - specify the number of lines per page when using the SOURCE or LIST options. Default is 60 lines per page.

(NO)LIBH [OS only]- strip off the trailing ".h" before searching for **#include <file.h>**. Default is LIBH. This affects the arguments to PP.

(NO)LIBgen - emit NAME- and ALIAS-statements for the external symbols defined in the file. This is useful when creating libraries with the Linkage Editor or the CMS TXTLIB command. A maximum of 15 ALIAS statements and one NAME statement are produced. This option has an effect only if the AS370 option (or NOAS370 and ASMG) is used. Default is NOLIBGEN.

(NO)List - ask the assembler to produce a listing. This flag may not be used together with the AS370 option. Default is NOLIST.

LW=# - specify line width for use with SOURCE. Default is 132 if PRINTER and 80 if TERMINAL.

(NO)MAin - output a warning message saying that this flag is obsolete. There is no longer a distinction between compiling a main program or

a subprogram.

(NO)MEmberseparation - treat each **struct/union** as a separate name space, i.e., require **x.m** to name a structure **x** with **m** as one of its members. Default is MEMBERSEPARATION. This flag is passed to **p1**.

(NO)MEMFiles - use files in memory for temporary files. Default is MEMFILES.

(NO)NUMber - use the contents of the sequence field (specified by SEQFLD) as line numbers in listings and error messages. The sequence field must contain a valid integer. Default is NONUM.

(NO)PAScal - compile a Pascal program by invoking the Pascal to C translator as the first pass. Default is NOPASCAL.

(NO)PRinter - use the SYSPRINT dataset for diagnostic output. Overrides the TERMINAL option. If specified under CMS, the filedef's for both SYSTERM and SYSPRINT will be PRINTER. Default is NOPRINTER.

(NO)RCHeck [Pascal only] - emit code to do run time array bounds checks in Pascal programs. Default is RCHECK. This flag is passed to **ptc**.

SDensity=# - use indexed switch table if the density of the table is at least **#** %. Default is 25. This flag is passed to **p2**.

SETSize=# [Pascal only] - make **#** the number of bits in the maximum allowable set size, i.e., the size of all sets whose basetype is integer becomes the specified power of two. Acceptable values are in the range [0 , 32). Default is 8. This flag is passed to **ptc**.

SEQfld=xxy - remove line numbers produced by some text editors in position **xx** and **y** columns onwards. Default is SEQFLD=0, i.e., no line numbers are present.

SNumber=# - use indexed switch table if the number of entries is less or equal to **#**. Default is 257. This flag is passed to **p2**.

(NO)SOurce - produce a numbered source listing of the input file. Files included are not listed. Default is NOSOURCE.

(NO)START - emit code that branches to the C program startup routine "C#START" at the beginning of the assembler file. This allows the program to be the only explicit input to the linker or loader. Default is START. This flag is passed to **p2**.

(NO)TErminal - use the SYSTERM dataset for compiler output. Default is TERMINAL.

(NO)VErbose - emit some information about what is to be done before doing it. This is done on the standard error file. Default is NOVERBOSE.

Xmask=# - map the three virtual sections, for Functions (04), Literals (02) and Variables (01) to the two physical sections Code (bit is

one) and Data (bit is zero). This option has only effect when using the AS370 option. Default is 7. This flag is passed to **p2**.

FILES

The following datasets are required by **CC**:

```
 1  SYSLIN   - Object module, output from the assembler
 4  SYSLIB   - Partitioned data set for #includes
 5  SYSIN    - Source program written in C or Pascal
 6  SYSPRINT - Compiler (and possibly assembler) printed output
 7  SYSPUNCH - Assembler program output from pass two
 8  SYSUT1   - Work file (used if NOAS370 or NOMEMFILES)
 9  SYSUT2   -  -  "   -
10  SYSUT3   -  -  "   -
11  SYSUT4   -  -  "   -
12  SYSTERM  - Compiler printed output
```

If EBCDIC is specified, the dataset ATOE is also required.

When used under OS you must define these with DD-statements. Under CMS, **CC** internally issues the following **filedef** commands for the above mentioned **ddnames** (**fn** is the filename specified as the first argument to **CC**).

```
filedef sysin disk fn c a
filedef syslin disk fn text a
filedef sysprint term ( recfm v lrecl 120 )
filedef systerm term ( recfm v lrecl 120 )
filedef syspunch disk fn s a
filedef sysut1 disk file sysut1
filedef sysut2 disk file sysut2
filedef sysut3 disk file sysut3
filedef sysut4 disk file sysut4
```

If NOAS370 is specified, the filedef for SYSPUNCH will be (SYSPUNCH is only used if NOASM is specifed):

```
filedef syspunch disk fn assemble a
```

If the PRINT option is specified, the filedefs for SYSPRINT and SYSTERM will be:

```
filedef sysprint printer
filedef systerm printer
```

If the DISK option is given, the filedef for SYSPRINT and SYSTERM will be:

```
filedef sysprint disk fn listing
filedef systerm disk fn clisting
```

The compiler datasets may have fixed or variable record-format and may be blocked, standard or spanned. If no defaults exists, fixed records of 80 bytes, collected in 3200 bytes blocks will be used. The actual restrictions are imposed by the system interface, described elsewhere.

The SYSUTx datasets are the workfiles for the system assembler (if used). If NOMEMFILES is specified they are also used as intermediate files for the compiler passes. They are required only if any of the NOAS370 or NOMEMFILES options are specified. Care should be taken if DCB-parameters are given for these files.

Files to be **#include**'d with **<file>** are searched in the partitioned dataset SYSLIB under OS (with the trailing ".h" stripped off), and on all accessed disks under CMS.

When invoked dynamically, **CC** makes use of any DD-name substitution list passed to it. The positions of DD-names are indicated in the above table.

EXAMPLE

To compile, load and run a simple C program under CMS (assume that the program 'hello c a' exists):

```
cc hello
global txtlib clib
load hello
start
```

To compile, load and run a simple C program under OS:

```
//SMITH   JOB  (...),REGION=512K
//COMP    EXEC PGM=CC
//SYSIN    DD  DATA,DLM='##'
/* THE MINIMUM PROGRAM
 * copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

/* put string to STDOUT
 */
BOOL main()
    {
    write(STDOUT, "hello world\n", 12);
    return (YES);
    }
##
//SYSLIB   DD  DSNAME=SYS1.C370.MACLIB,DISP=SHR
//SYSLIN   DD  UNIT=SYSSQ,DISP=(,PASS),SPACE=(TRK,(1,1)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//SYSPRINT DD  SYSOUT=A
//GO      EXEC PGM=LOADER,COND=(0,LT,COMP),PARM=(NOPRINT,TERM)
//SYSLIB   DD  DSNAME=SYS1.C370.CLIB,DISP=SHR
//SYSLIN   DD  DISP=(OLD,DELETE),DSNAME=*.COMP.SYSLIN
//SYSTERM  DD  SYSOUT=A
//SYSPRINT DD  SYSOUT=A
//SYSUT1   DD  UNIT=SYSSQ,SPACE=(CYL,(10,10))
```

WARNINGS

Assigning anything other than a printer or spooling-system to the dataset SYSPRINT may give unpredictable results if you use the PRINT option, since

more than one program will use this dataset.

If too little memory is available to the temporary **mem**-files, The compiler will encounter the error code EIO on **write**. If this should happen, either increase the available memory for CC, or specify NOMEMFILES.

The default filedef commands issued under CMS cannot be overridden.

SEE ALSO
pp, p1, p2.370, as.370, tobj

NAME

Files - under CMS and OS

FUNCTION

The system interface to CMS and OS is fairly complex due to the various ways that I/O can be handled. The interface is designed to be transparent to users that program in "Whitesmiths' portable style", yet powerful enough to give the IBM-oriented user access to various features under CMS and OS.

All I/O is done through files, where a file can be a data set, the user's terminal or a special interface such as "write to programmer". Under CMS you can also access CMS-files, the virtual card-reader, card-punch and printer and the tape-drives.

Files under CMS and OS are said to have "record-structure", either of fixed or variable (with a maximum) length. The system interface converts this structure to and from a "byte-stream".

For a binary file this simply means ignoring the record boundaries. For a text file it also implies translating from EBCDIC to ASCII and separating the records (lines) with a newline character.

With a few exceptions, data of arbitrary length can be read or written. For example, a file containing ten 100-bytes records can be copied with 1000 reads and writes specifying one byte, or it can be copied with one single read/write sequence specifying 1000 bytes.

TEXT AND BINARY FILES

Text and binary files are treated differently. A file is specified as being a text or binary file by the third parameter in the call to **open** or **create**.

Text files

are assumed to contain printable text which is translated to ASCII on **read** and to EBCDIC on **write**. Thus, text is always represented in ASCII inside a program.

A newline character (ASCII linefeed), causes **write** to terminate the current output record. The rest of the record will be filled with spaces if the file has fixed or undefined record-format. The newline itself is never written.

The record will also be output if the (maximum) record-length is exceeded.

A call to **read** returns the number of characters asked for, plus what is left in the current input record, plus one more (the newline character). A newline is inserted when a complete record has been read.

Text files can be accessed only sequentially, i.e., **lseek** is generally not supported on them. However, on CMS-files you can seek to the beginning and the end even on a text file.

Binary files

contain a unstructured sequence of bytes. There is no record structure visible to the program, and all character codes are allowed. In the physical files data is packed in records, with no interpretation at all. When record-format is fixed, the last record written will be zero-padded unless the number of characters written is an integral number of the record length.

It is the programmer's responsibility to perform any translation between EBCDIC and ASCII if a binary file also contains text.

FILENAMES

Filenames, as given to the functions **open**, **create** and **remove**, have the following form:

[drivername:]driverarg

The **driverarg** is decoded by the I/O-drivers and can, for instance, be a CMS filename, a **ddname** (data-definition name), or just an empty string. If the drivername is omitted the following defaults will be used:

	rsize=0	rsize>0	remove
CMS	cms	cms	cms
OS	seq	mem	mem

I/O DRIVERS

In the system interface there are several drivers that support different types of files. The following drivers are currently supported:

cms
cmsb Access a CMS-file. (CMS only).

The argument to the driver takes the following form:

[fm:][fn.]ft[([recfm][lrecl])]

The CMS-file **'fn ft fm'** will be accessed. If the file-mode **fm** is omitted, 'A1' will be used. If it is given as *****, all accessed disks will be searched. If the file-name **fn** is omitted, 'FILE' will be used. The record-format **recfm** and record-length **lrecl** can be specified when creating a file. They are set according to the following hierarchy:

- If the record-format or the record-length is specified within parentheses in the file-name, that specification is used.
- If an old file exists, its record-format or record-length is used.
- A value is assigned, depending on the "record-size", **rsize**, given to **open** or **create**.

	rsize=0	rsize=1	rsize>1
recfm	V	F	F
lrecl	512	512	rsize

Only F and V are recognized as record-formats. Direct access via **lseek** is allowed if the file has fixed record-format and is opened as a binary file. Two special cases of direct access is allowed on text files: **lseek** to the beginning and the end of the file.

For files opened as text files any trailing blanks in a record are deleted if the CMS driver is used. Also, as CMS-files cannot contain empty records, empty lines in a textfile are written as records containing one space.

da Access a data set with BDAM. (OS only)

This allows direct access (**lseek**) on the file. The argument is the **ddname** of the file. The file must already exist and be formatted to its full size, i.e., it cannot be extended.

lib
libh Access a member in a partitioned data set using BPAM.

The argument has the form:

ddname/membername

or

membername

The member **membername** in the partitioned data set **ddname** is accessed sequentially with BPAM. In the second form, 'SYSLIB' will be used as **ddname**. Access is read-only (in the current release). If the file is opened as a text file, trailing blanks in a line will be stripped on **read**.

When the **libh** is used, any trailing ".h" in the membername is removed.

mem Access a "file" residing in primary memory. The argument, **mfname**, is the name of the file.

create on a mem-file will create it in memory. The file can later be accessed by the program (or by its "children", "parents" or other "relatives", if **xecl** or **xecv** is used to start up other programs).

If a mem-file is opened that has not previously been created, the file with **ddname mfname** is read sequentially and copied into memory.

If a mem-file hasn't been removed when the root **main** exits, its contents will be written sequentially to the file with ddname **mfname**.

No translation is performed when reading/writing to mem-files. Translation of text files is performed only when they are transferred between memory and data sets. This speeds up the handling of temporary text files.

Direct access via **lseek** is allowed on mem-files.

mem-files can be used for either temporary files or for direct-access to basically sequential data sets, or for files that need to be extended.

mt0 Access the virtual tape drive at virtual address 181 with RDTAPE and WRTAPE macros. (CMS only).

Data is read or written 512 bytes at a time. This driver is meant to be compatible with the file **/dev/mt0** under IDRIS and UNIX systems. When the file is closed, a tape-mark is written if the file is opened for writing and the tape is rewound.

mt8 Same as **mt0** except that the tape is not rewound. (CMS only).

mt16 Same as **mt0** except that no tape mark is written. (CMS only).

mt24 Same as **mt8** and **mt16**. (CMS only).

null The data-sink. Returns immediate end-of-file on **read**. Data written is thrown away, 512 bytes at a time.

prt Write to the virtual printer with the PRINTL macro-instruction. (CMS only.)

When the file is closed a 'CP CLOSE PRINTER' is issued. Maximum line length is 131 characters.

pun Write to the virtual card punch with the PUNCHC macro-instruction. (CMS only.)

Maximum line-length is 80 characters. A 'CP CLOSE PUNCH' is issued when the file is closed.

rdr Read cards (or lines) from the virtual card-reader with the RDCARD macro-instruction. (CMS only.)

Each RDCARD asks for 512 characters.

rmt0 Access the tape drive at virtual address 181 with RDTAPE and WRTAPE macros. (CMS only.)

If **rsize**, the third argument to **open** and **create**, is equal to 1, the largest block-size that can be read or written is 32760 bytes, otherwise **rsize** is taken as the largest possible block-size. Each call to **read** must specify a length large enough to contain the physical block to be read. The actual block-size is returned by **read**. A call to **write** will result in a block of the given length being written.

This driver is meant to be compatible with the file **/dev/rmt0** under IDRIS and UNIX systems.

When the file is closed, a tape-mark is written if the file is opened for writing and the tape is rewound.

rmt8 Same as **rmt0** except that the tape is not rewound. (CMS only.)

rmt16 Same as **rmt0** except that no tape mark is written. (CMS only.)

rmt24 Same as **rmt8** and **rmt16**. (CMS only.)

seq
seqb Access a data set with QSAM. The argument is the **ddname** of the data set. The file can have fixed, variable or undefined record format and may be blocked, standard and/or spanned. For spanned records you must specify BFTEK=A in the DCB-parameters.

For files opened as text files any trailing blanks in a record are deleted if the **seq**-driver is used.

If a control-character is specified in the DCB-parameters (RECFM=...A) a form-feed produces the control-character **1** which gives a page-eject on printers. The character **\r** (return) causes overprinting with the control-character **+**.

If no record length is given, a length of 80 is assumed and RECFM=F is set. In the absence of block-size, 40*LRECL is used and RECFM=B is set.

term
terml Read/write using the the TGET/TPUT interface. A partial line (i.e., without trailing newline) is output in ASIS mode when the **term**-driver is used, otherwise (when the **terml**-driver is used), only full lines are output (in EDIT-mode). All complete lines are output in EDIT mode. Typing the character-sequence **/*** in the first position of a line when reading from a terminal generates an end-of-file condition. **terml** is useful on systems where TPUT ASIS is not supported (such as CMS).

wtp Output lines via the "write to programmer"-interface. Record length is 70 characters.

DATA-DEFINITION NAMES - DDNAMES

In the cases where OS data sets are accessed they are referenced by their data-definition name, or **ddname**. Under CMS, **ddnames** can also be used when accessing CMS-files, virtual unit-record devices and tapes.

A **ddname** is connected to a file by means of the Job Control Language (JCL) DD-statement under OS, the command ALLOCATE under TSO or the FILEDEF-command under CMS.

Any dot in in the **ddname**-part of a filename is replaced by **@** if the name contains less than 8 character, otherwise it is removed. A **ddname** can contain at most 8 characters.

STANDARD FILEDESCRIPTORS

The files opened for these file-descriptors depend on the environment. The following files are "open" when **main** gets control:

Filedesc	CMS & TS	Batch	Comment
STDIN	term:	seq:sysin	demand opened
STDOUT	term:	seq:sysprint	demand opened
STDERR	term:	wtp:	

This setup depends on the functions **_ists** and **_iscms**. **_ists** is a system-dependent routine that should return YES if the program is running under a time-sharing system (TS).

STDIN and STDOUT are actually "demand-opened", i.e., they will not really be opened until they are used. This makes it possible to omit SYSIN and SYSPRINT DD-statements under OS if the files aren't needed.

SEE ALSO

ASCII, _iscms, _ists, open, create, close, read, write, lseek

NAME

Errors - CMS & OS system subroutines

SYNOPSIS

```
#include <os.h>
```

FUNCTION

All standard 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, **<os.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.

The following are the principle definitions from **<os.h>**:

```
#define ENOERROR    0    /* No error */
#define EBADFD      -1   /* Bad file descriptor */
#define EBADNM      -2   /* Bad filename */
#define EOPEN       -3   /* Open error */
#define ENODRVR     -4   /* No such driver */
#define EBADMODE    -5   /* Bad mode */
#define EIO         -6   /* I/O error */
#define EMFILE      -7   /* Too many open files */
#define ESEEK       -8   /* Bad call to lseek */
#define EBADACC     -9   /* Bad access */
#define EBADFRM     -10  /* Bad file format */
#define EBADMBR     -11  /* No such member */
#define EBADFL      -12  /* No such file */
#define ENOTACC     -13  /* File not accessable */
#define ENOCMS      -14  /* CMS only */
```

SEE ALSO

various IBM manuals.

NAME

UPPER - case support

FUNCTION

C programs are normally written in lowercase. In some instances it is desirable to write in uppercase. This is especially true in Japan, where many terminals have only uppercase letter, the lower case being used for Kata-Kana. The 370 compiler provides some limited support for such users.

The first problem is the use of uppercase in source files. The preprocessor accepts either case for #-commands so there is no problem there. The reserved words in C are handled by the header file **upper.h** that will translate all keywords to lowercase. As for the names of functions, they are all translated to uppercase in the object modules.

The second problem is the program output that may be in lowercase, most notably error messages from the compiler. This has been solved by including an alternate write routine called **jwrite**. This will translate all output to text files to uppercase, with the exception of **mem:**-files.

If the compiler is installed with the "JAPAN option", this routine will be used when linking the compiler, giving error messages in uppercase. The routine is not installed in the C library. It can, however, be used in a user program by including the member "JWRT@ASC" in the stage dataset when linking under OS or issuing "LOAD MYPROG JWRITE" under CMS provided that the stage disk is accessed.

WARNINGS

If programs are written in uppercase there can be conflicts between defines in uppercase and function names. Notably, the defines READ and WRITE in **std.h** conflict with the functions **read()** and **write()**.

Also, the compilation of Pascal-programs depends on the fact that **mem:**-files are not translated. It is therefore not possible to run **ptc** separately and then compile the program produced if the compiler has been installed with the JAPAN-option.

NAME

ASCII - character representation

FUNCTION

The default internal representation of text (characters and strings) in a C program is ASCII. Normally, this is hidden by the system interface, which translates text files, parameters and names of files and data sets to and from EBCDIC. Under certain circumstances, however, as when a file containing text is opened as a binary file (to be used by programs other than those written in C), or when a routine written in another language is called with text parameters, this fact might be of importance and the EBCDIC version of the compiler should be considered.

The conversion is governed by the tables **_atoe** and **_etoa**, which can be changed to conform to local standards.

The compiler and utilities always use ASCII internally.

The following table shows the positions of the standard ASCII characters in the EBCDIC alphabet used in the supplied tables.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0					SP	&	-						{	}	\	0	0
1							/		a	j	~		A	J		1	1
2									b	k	s		B	K	S	2	2
3									c	l	t		C	L	T	3	3
4									d	m	u		D	M	U	4	4
5	\t		\n						e	n	v		E	N	V	5	5
6		\b							f	o	w		F	O	W	6	6
7									g	p	x		G	P	X	7	7
8									h	q	y		H	Q	Y	8	8
9								`	i	r	z		I	R	Z	9	9
A						!		:									A
B	\v				.	$	,	#									B
C	\f				<	*	%	@									C
D	\r				(	)	_	'			[	]					D
E					+	;	>	=									E
F					¦	^	?	"									F

SEE ALSO

_atoe, _etoa, xlate, EBCDIC

NAME

EBCDIC - the EBCDIC version of the compiler

FUNCTION

The IBM/370 series uses the EBCDIC character representation. While the C language is in theory independent of the underlying character set, most implementations are strongly biased towards ASCII.

A related problem is the type **char**. In C it is promised that a character stored in a **char** variable will have a positive value. This means that **char** must be unsigned when using EBCDIC.

It is of course possible, (and preferable) to write C programs that do not make any assumption about the character set or the type **char**, and will therefore run in both ASCII and EBCDIC. Many existing programs, however, (including the compiler itself) assume ASCII and/or signed characters.

The Whitesmiths C compiler gives the user the option of using either EBCDIC or ASCII. Using EBCDIC on the 370 reduces the problems of handling files containing both binary data and text and increases I/O peformance slightly. ASCII, on the other hand, reduces porting problems. (i.e., porting programs with ASCII-dependencies to the 370.)

The support for multiple character sets is done in a general fashion, using a translation file to the character set, making it possible to compile programs for any character set. (see the **-map** flag on the manual page for **pp** in Section II of this manual).

In the same way, the type for **char** (signed or unsigned) can be specified. (see the **+u** flag on the **p1** manual page in Section II).

Native compiler

On the 370 the compiler itself runs in ASCII, and the default for the compiler driver is also ASCII and signed characters. The EBCDIC option to the compiler driver produces programs that use EBCDIC and unsigned **char**.

There are also two sets of libraries, one for EBCDIC and one for ASCII. The code produced must be linked with the right library.

Cross compilers on the 370

It is possible to cross-compile programs on the 370 for any character set on the target machine. This is done by adding the **-map** (and possibly +u) flag to the prototype. In the case of EBCDIC, the existing map-file can be used.

Note however that the standard libraries supplied with the cross-compiler are in ASCII. If the programs make use of any library functions that accept character values or strings (such as **printf**) these must be recompiled with the appropriate compiler.

Cross compiling to the 370

Cross-compilers targeting to the 370 contain two sets of prototypes and two sets of libraries. They can therefore be used to produce programs running in either EBCDIC or ASCII in the same way as the native compiler.

NAME

Interface - to CMS & OS

FUNCTION

Programs written in C for operation on machines with 370-like architecture under CMS (release 2 and upwards) and OS (PCP, MFT, MVT, VS1, VS2/SVS, VS2/MVS) are translated according to the following specifications:

external identifiers - may be written in both upper and lower case, but only one case is significant. The first seven letters must be distinct. Any underscore is changed to a $. A $ is prepended to each identifier unless the first character is in upper-case.

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

literal data - such as strings and switch tables are generated into a CSECT.

initialized data - are generated into a CSECT. External data names are published via ENTRY declarations.

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

function calls - are performed by

1) allocating, if necessary, the total number of bytes on the stack for a save area and parameters, which is rounded up to a multiple of eight (for alignment of doubles in the next frame). This is only done for the nested function calls since space for the first level of function calls is reserved at function entry. Doubles are aligned on double-word boundaries, and may cause "holes" among the arguments.

2) moving arguments on the stack, right to left. The addresses of the stacked arguments are ascending, and the first argument starts at 72(R10). Character data is zero-extended to integer, short is sign-extended to integer, and float is zero-padded to double.

3) calling via:

```
    l    15,1123
    balr 14,15
    ...
    .align 2
1123:
    .long _func
```

4) retracting the stack pointer to its former value, if necessary.

Except for any returned value, the registers R12, R15, F0, F2, F4 and F6 are undefined on return from a function call. All other registers are preserved.

The types **short**, **char**, **int**, **long**, and **pointer to** are returned in R12. Basically, all these types are coerced to **int**. They are sign-extended if signed to begin with, and zero-padded if not. Thus, **char** is zero-padded to **int** and **short** is sign-extended to **int**. The types **long** and **pointer to** are the same width as **int** to begin with.

Floating point numbers (**floats** and **doubles**) are returned in FRET. **floats** are widened to **double** before returning.

stack frames - are maintained by each C function, using R9 as a frame pointer. On entry to a function, the sequence:

```
b 20(15)
.long <space for 1:st level of function calls>
.long <space for auto, temp. storage, 1:st lvl of fun calls +8>
.long 4096  / constant used below
.long 0 / reserved for future use, must be zero
using _f,15
stm 14,11,12(13)
st 13,4(10)
lr 13,10
lr 9,10
s 9,12(15)
la 10,4000(9)
lr 11,15
drop 15
using _f,11
```

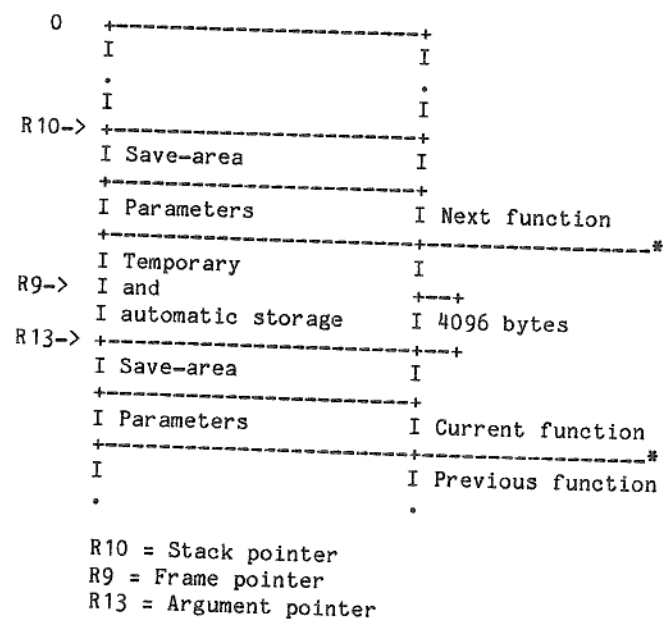
will save all registers in the save area, link save areas together, set R13 to R10, which now points on the new save area and arguments, establish a new R9 (4096 below R13), and allocate new stack by subtracting the size of the frame (which is at 8(R15)) from R10.

If the program has been compiled with the stack checking option, **-ck**, the function start-up will be performed by the external routine **c~ents**.

Arguments are now at 72(R13), 76(R13), etc. R9 is always biased -4096 to R13; this makes references to auto-storage possible with one instruction, provided that the total length of auto-storage variables is less than 4096 bytes.

The first allocated integer auto-variable is referenced via 4092(R9). To return, fetch the old R13 at 4(R13), restore registers R14-R11 and return via the statement **BR R14**. The previous R10 is ignored so the stack need not be balanced on exit.

A look at the stack (just before the the next function is entered):



(Note that for functions with little auto-storage, R9 will point at a lower address than R10.) The function start-up code handles OS save areas, which are allocated on the stack. R13 is always a pointer to a save area.

data representation - characters are treated as unsigned. Short integers are stored as two bytes, more significant byte first. Integer is the same as long; both are four bytes stored in descending order of significance. All signed integers are twos complement. Floats are four bytes, doubles are eight, in descending order of significance in all cases.

storage bounds - for the sake of efficiency on the 370 the strongest storage bounds are enforced. This means that a scalar datum **n** bytes in length is stored at an address that is a multiple of **n** bytes.

CSECT name - can be produced by the **-c** option to **tobj**.

module name - can be produced, along with alias names, by the **-l** option to **tobj**.

SEE ALSO

c~start(IV), c~ents(IV), p2.370(II), tobj(II)

NAME

Echo - documentation and source

SYNOPSIS

echo -[m n] <args>

FUNCTION

echo copies its arguments to STDOUT. They are not interpreted in any way, and may be arbitrary strings. By default, the arguments are output separated by a single space; the last argument is terminated by a newline character. If there are no arguments, nothing is output.

The flags are:

-m output each argument on a separate line.

-n suppress the newline following the last argument.

RETURNS

echo returns success if there are no arguments or if all characters are successfully written.

EXAMPLE

To make a one-line message file:

```
% echo happy new year! >motd
```

SOURCE

```
/*  ECHO ARGUMENTS TO STDOUT
 *  copyright (c) 1980, 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

/*  flags:
    -m      output newline between arguments
    -n      do not put newline at end of arguments
 */
BOOL mflag {NO};
BOOL nflag {NO};

TEXT *_pname {"echo"};

/*  output args separated by space or newline
 */
BOOL main(ac, av)
    BYTES ac;
    TEXT **av;
    {
    IMPORT BOOL mflag, nflag;
    FAST COUNT n, ns, nw;
    TEXT *q, *between;

    getflags(&ac, &av, "m,n:F <args>", &mflag, &nflag);
    if (!ac)
```

```
        return (YES);
    between = mflag ? "\n" : " ";
    for (nw = 0, ns = 0; ac; --ac, ++av)
        {
        if (nw)
            {
            nw += write(STDOUT, between, 1);
            ++ns;
            }
        nw += write(STDOUT, *av, n = lenstr(*av));
        ns += n;
        }
    if (!nflag)
        {
        nw += write(STDOUT, "\n", 1);
        ++ns;
        }
    return (nw == ns);
    }
```

SEE ALSO

Intro

NAME

_atoe - translation table from ASCII to EBCDIC

SYNOPSIS

```
/*  TRANSLATION TABLE FROM ASCII TO EBCDIC
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

GLOBAL TEXT _atoe[]
    {
    0x00, 0x01, 0x02, 0x03, 0x37, 0x2d, 0x2e, 0x2f,
    0x16, 0x05, 0x25, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x3c, 0x3d, 0x32, 0x26,
    0x18, 0x19, 0x3f, 0x27, 0x1c, 0x1d, 0x1e, 0x1f,
    0x40, 0x5a, 0x7f, 0x7b, 0x5b, 0x6c, 0x50, 0x7d,
    0x4d, 0x5d, 0x5c, 0x4e, 0x6b, 0x60, 0x4b, 0x61,
    0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
    0xf8, 0xf9, 0x7a, 0x5e, 0x4c, 0x7e, 0x6e, 0x6f,
    0x7c, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7,
    0xc8, 0xc9, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5, 0xd6,
    0xd7, 0xd8, 0xd9, 0xe2, 0xe3, 0xe4, 0xe5, 0xe6,
    0xe7, 0xe8, 0xe9, 0xad, 0xe0, 0xbd, 0x5f, 0x6d,
    0x79, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
    0x88, 0x89, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96,
    0x97, 0x98, 0x99, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6,
    0xa7, 0xa8, 0xa9, 0xc0, 0x4f, 0xd0, 0xa1, 0x07,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x15, 0x06, 0x17,
    0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x09, 0x0a, 0x1b,
    0x30, 0x31, 0x1a, 0x33, 0x34, 0x35, 0x36, 0x08,
    0x38, 0x39, 0x3a, 0x3b, 0x04, 0x14, 0x3e, 0xe1,
    0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48,
    0x49, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
    0x58, 0x59, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
    0x68, 0x69, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75,
    0x76, 0x77, 0x78, 0x80, 0x8a, 0x8b, 0x8c, 0x8d,
    0x8e, 0x8f, 0x90, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e,
    0x9f, 0xa0, 0xaa, 0xab, 0xac, 0x4a, 0xae, 0xaf,
    0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
    0xb8, 0xb9, 0xba, 0xbb, 0xbc, 0x6a, 0xbe, 0xbf,
    0xca, 0xcb, 0xcc, 0xcd, 0xce, 0xcf, 0xda, 0xdb,
    0xdc, 0xdd, 0xde, 0xdf, 0xea, 0xeb, 0xec, 0xed,
    0xee, 0xef, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff
    } ;
```

FUNCTION

This is the table provided for translation from ASCII to EBCDIC. It can be changed to comply to local conventions. If possible this table and _etoa should be complementary.

SEE ALSO

_etoa

NAME

_cmpswt - set CMS compiler switch

SYNOPSIS

```
VOID _cmpswt(flag)
    BOOL flag;
```

FUNCTION

_cmpswt sets the compiler-switch, COMPSWT, on or off. It affects the function of the LOAD, LINK and XTCL supervisor-calls and the xecv and xecl interface functions.

SEE ALSO

CMS User's Guide
_main, xecv, xecl

NAME

_cmsl - issue CMS commands with argument list

SYNOPSIS

```
ARGINT _cmsl(s0, s1, ..., NULL)
    TEXT *s0, *s1, ...;
```

FUNCTION

_cmsl can be used to issue CMS commands from within a C program.

_cmsl builds a CMS parameter list and calls _svc202 with the address of this list.

Each argument is translated to EBCDIC, truncated or blank-filled to a length of eight bytes and appended to the parameter list.

An extra eight bytes entry with all bits set to one is appended to the parameter list.

EXAMPLE

```
cmsl("filedef", "sysprint", "terminal", NULL);
```

RETURNS

_cmsl returns the returncode from _svc202.

SEE ALSO

_cmsl, _svc202

NAME

_cmsv - issue CMS commands with argument vector

SYNOPSIS

```
ARGINT _cmsv(av)
    TEXT **av;
```

FUNCTION

_cmsv can be used to issue CMS commands from within a C program.

_cmsv builds a CMS parameter list and calls _svc202 with the address of this list.

Each argument pointed by the elements in the argument vector is translated to EBCDIC, truncated or blank-filled to a length of eight bytes and appended to the parameter list.

An extra eight bytes entry with all bits set to one is appended to the parameter list.

EXAMPLE

```
GLOBAL TEXT *av[] = {"TYPE", "CHDRS", "MACLIB",
    "(", "MEMBER", "STD@H", NULL};
    BOOL main()
    {
    return (_cmsv(av) == 0);
    }
```

RETURNS

_cmsv returns the returncode from _svc202.

SEE ALSO

_cmsl, _svc202

NAME

_etoa - translation table from EBCDIC to ASCII

SYNOPSIS

```
/*  TRANSLATION TABLE FROM EBCDIC TO ASCII
 *  copyright (c) 1983 by Whitesmiths, Ltd.
 */
#include <std.h>

GLOBAL TEXT _etoa[]
    {
    0x00, 0x01, 0x02, 0x03, 0x9c, 0x09, 0x86, 0x7f,
    0x97, 0x8d, 0x8e, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x9d, 0x85, 0x08, 0x87,
    0x18, 0x19, 0x92, 0x8f, 0x1c, 0x1d, 0x1e, 0x1f,
    0x80, 0x81, 0x82, 0x83, 0x84, 0x0a, 0x17, 0x1b,
    0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x05, 0x06, 0x07,
    0x90, 0x91, 0x16, 0x93, 0x94, 0x95, 0x96, 0x04,
    0x98, 0x99, 0x9a, 0x9b, 0x14, 0x15, 0x9e, 0x1a,
    0x20, 0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6,
    0xa7, 0xa8, 0xd5, 0x2e, 0x3c, 0x28, 0x2b, 0x7c,
    0x26, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xae, 0xaf,
    0xb0, 0xb1, 0x21, 0x24, 0x2a, 0x29, 0x3b, 0x5e,
    0x2d, 0x2f, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7,
    0xb8, 0xb9, 0xe5, 0x2c, 0x25, 0x5f, 0x3e, 0x3f,
    0xba, 0xbb, 0xbc, 0xbd, 0xbe, 0xbf, 0xc0, 0xc1,
    0xc2, 0x60, 0x3a, 0x23, 0x40, 0x27, 0x3d, 0x22,
    0xc3, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
    0x68, 0x69, 0xc4, 0xc5, 0xc6, 0xc7, 0xc8, 0xc9,
    0xca, 0x6a, 0x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70,
    0x71, 0x72, 0xcb, 0xcc, 0xcd, 0xce, 0xcf, 0xd0,
    0xd1, 0x7e, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78,
    0x79, 0x7a, 0xd2, 0xd3, 0xd4, 0x5b, 0xd6, 0xd7,
    0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd, 0xde, 0xdf,
    0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0x5d, 0xe6, 0xe7,
    0x7b, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
    0x48, 0x49, 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed,
    0x7d, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f, 0x50,
    0x51, 0x52, 0xee, 0xef, 0xf0, 0xf1, 0xf2, 0xf3,
    0x5c, 0x9f, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58,
    0x59, 0x5a, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8, 0xf9,
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
    0x38, 0x39, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff
    } ;
```

FUNCTION

This is the table provided for translation from EBCDIC to ASCII. It can be changed to comply to local conventions. If possible this table and _atoe should be complementary.

SEE ALSO

_atoe

NAME

_iscms - determine if running under CMS

SYNOPSIS

```
BOOL _iscms()
```

FUNCTION

_iscms is a system dependent function that is provided for programs that need to know if they are running under CMS.

RETURNS

_iscms returns YES if the program is running under CMS, otherwise NO. This is determined by comparing the byte at offset 7 in the CVT. If it is equal to 255 _iscms return YES, otherwise NO. (The address of the CVT is contained in the fullword at location 0x10.)

SEE ALSO

_ists

NAME

_ists - determine if running interactively

SYNOPSIS

BOOL _ists()

FUNCTION

_ists is a system dependent function that is provided for programs that need to know if they are running interactively, and therefore whether they can have a reasonable conversation with whatever is associated to the standard input. The return value of _ists also affects the defaults chosen for STDIN/STDOUT.

RETURNS

_ists returns YES if the program is running under a time-sharing system otherwise NO.

SEE ALSO

Files

WARNINGS

The supplied function is a dummy that returns **_isems()**, but any installation can easily provide a true _ists().

NAME

_main - setup for main call

SYNOPSIS

```
BOOL _main()
```

FUNCTION

_main is the function called whenever a C program is started. It parses the PARM-field on the EXEC statement or the CMS command line into argument strings, sets argument zero to _pname under OS, sets _pname and argument zero to the command-name obtained from the extended parameter-list under CMS, causes STDIN to be "demand opened" as "seq:sysin" or "term:" (unless redirected with '<'), causes STDOUT to be "demand opened" as "seq:sysprint" or "term:" (unless redirected with '>'), causes STDERR to be opened as "wtp:" or "term:", and calls the (user-supplied) routine main.

The PARM-field is interpreted as a series of strings separated by spaces, commas, or both. Each letter in the field is converted to lower-case unless it is preceded by the escape-character "$" or you are running under CMS. The sequences "$ " and "$," represent a blank or comma that should be treated as a part of a parameter rather than as a delimiter. A "$" can be represented as "$$".

If a string begins with a '<', the remainder of the string is taken as the name of the 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 the file to be opened for writing and used as the standard output, STDOUT. All other strings are taken as argument strings to be passed to main.

Under CMS, the compiler switch, COMPSWT, will be set.

If the program has been started by xecl/xecv none of the above is done. _main just passes the arguments given on to main.

RETURNS

_main returns the boolean value obtained from the main call.

SEE ALSO

_pname, exit

NAME

_paths - the search path

SYNOPSIS

```
TEXT *_paths;
```

FUNCTION

_paths is a (NUL terminated) string used by some programs as a search path for commands or files. It consists of the prefixes to be tried, separated by '|'-characters. _paths is supplied primarily becase it is used by the compiler driver "c".

Under CMS _paths is initalized to the string:

"|A:|B:|C:|D: ... |Z:"

giving the same search order as xecl/xecv.

Under OS it is initalized to:

"lib:"

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 provides no definition for _pname, a library routine supplies the name "error" under OS, since it is used primarily for labelling diagnostic printouts.

Under CMS _pname is set to the command-name obtained from the extended parameter-list.

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

SEE ALSO

_main

NAME

_stksiz - size of the stack

SYNOPSIS

```
BYTES _stksiz {20000} ;
```

FUNCTION

_stksiz specifies the size in bytes of the stack for the program. This area is allocated at task startup time.

The default value provided in clib is 20000 bytes. To reduce or increase this value, define the variable _stksiz in your C program proper and statically initialize it to the value you want.

NAME

_svc - call OS

SYNOPSIS

```
ULONG _svc(svcno, osint)
    COUNT svcno;
    struct
        {
        ULONG r14;
        ULONG r15;
        ULONG r0;
        ULONG r1;
        } *osint;
```

FUNCTION

_svc is the C callable function that permits arbitrary calls to be made on OS. It loads registers r14 to r1 from the address pointed to by osint, issues the svcno svc and writes the contents of the registers r14 to r1 back in the user provided structure.

RETURNS

_svc returns the contents of r15.

NAME

_svc202 - call CMS

SYNOPSIS

```
ULONG _svc202(arg)
    DOUBLE *arg;
```

FUNCTION

_svc202 is the C callable function that permits arbitrary calls to be made on CMS. It loads register r1 with the given argument and issues a standard svc 202 call.

RETURNS

_svc202 returns the contents of r15.

SEE ALSO

_cmsl, _cmsv

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 an error code defined i <os.h>.

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(fname, mode, rsize)
    TEXT *fname;
    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 update (reading and writing)

If (rsize == 0) the file is treated as a textfile, otherwise as a binary file. For some of the I/O-drivers the value of rsize may affect the actual record-length. In those cases, a default will be chosen if (rsize == 1).

RETURNS

create returns a file descriptor for the created file or a negative number which is an error code defined in <os.h>.

EXAMPLE

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

SEE ALSO

Files, close, open, remove, uname

WARNINGS

Under OS:
Only really works for "mem:"-files, otherwise usually equivalent to open. The treatment of the file is then controlled by the DISP parameter in the DD-statement.

NAME

exit - terminate program execution

SYNOPSIS

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

FUNCTION

Terminates program execution by calling:

```
exitrc((success == YES) ? 0 : 16);
```

which will call all functions registered with onexit, exit via _exit, and close all files.

RETURNS

exit will never return to the caller

EXAMPLE

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

SEE ALSO

exitrc, onexit, _exit

NAME
exitrc - terminate program execution with return-code

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

FUNCTION
exitrc calls all functions registered with onexit, closes all files, and terminates program execution by calling _exit, which resides in the start-tup-code.

A portable program uses exit rather than exitrc.

RETURNS
exitrc will never return to the caller

EXAMPLE
```
if ((fd = open(fname, READ)) < 0)
    {
    putstr(STDERR, "can't open ", fname, "\n", NULL);
    exitrc(8);
    }
```

SEE ALSO
exit, onexit, _exit

NAME

fcall - call a Fortran program

SYNOPSIS

```
VOID fcall(FN, arg1, ...)
    VOID (*FN)();
    ....
```

FUNCTION

fcall is an interface function for calling Fortran programs. FN is the Fortran subroutine to call (which must be written in upper-case), and arg1, ... are the arguments to be passed to FN on the call.

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

The last argument must have the most significant bit set to one.

RETURNS

fcall returns nothing.

EXAMPLE

```
#define FLAST 0x80000000

IMPORT VOID fcall(), FTNSUB();
ARGINT arg;

fcall(&FTNSUB, FLAST | &arg);
```

SEE ALSO

fcalld, fcalli

NAME

fcalld - call a Fortran program and return DOUBLE

SYNOPSIS

```
DOUBLE fcalld(FN, arg1, ...)
    DOUBLE (*FN)();
    ...
```

FUNCTION

fcalld is an interface function for calling Fortran programs. FN is the Fortran function to call (which must be written in upper-case), and arg1, ... are the arguments to be passed to FN on the call.

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

The last argument must have the most significant bit set to one.

RETURNS

fcalld returns whatever float FN returns widened to DOUBLE.

EXAMPLE

```
#define FLAST 0x80000000

IMPORT DOUBLE fcalld(), DSIN(), d;

putfmt("sin %1.6d = %1.6d\n", d, fcalld(&DSIN, FLAST | &d));
```

SEE ALSO

fcall, fcalli

WARNINGS

COMPLEX and REAL*16 cannot be returned properly.

NAME

fcalli - call a Fortran program and return integer

SYNOPSIS

```
ARGINT fcalli(FN, arg1, ...)
    ARGINT (*FN)();
    ....
```

FUNCTION

fcalli is an interface function for calling Fortran programs. FN is the Fortran function to call (which must be written in upper-case), and arg1, ... are the arguments to be passed to FN on the call.

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

The last argument must have the most significant bit set to one.

RETURNS

fcalli returns whatever integer FN returns.

EXAMPLE

```
#define FLAST 0x80000000

IMPORT ARGINT fcalli(), ADDIJ();
ARGINT i, j;

return (fcalli(&ADDIJ, &i, FLAST | &j));
```

SEE ALSO

fcall, fcalld

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 and if (sense == 2) the offset is algebraically added to the length of the file.

The call lseek(fd, 0L, 1) is guaranteed to leave the file pointer unmodified and, more important, to succeed only if lseek calls with sense 0 and 1 are both acceptable and meaningful for the fd specified.

Only some of the I/O-drivers accept the lseek call and some drivers only allow special cases.

RETURNS
lseek returns the file descriptor if successeful, or a negative number which is an error code described in <os.h>.

EXAMPLE
To read a 512-byte block (if possible):

```
BOOL getblock(fd, buf, blkno)
    FILE fd;
    TEXT *buf;
    COUNT blkno;
    {
    if (lseek(fd, (LONG) blkno << 9, 0) != fd)
        return (NO) ;
    return (fread(fd, buf, 512) != 512);
    }
```

SEE ALSO
Files

NAME

mkexec - make file executable

SYNOPSIS

```
BOOL mkexec(fname)
    TEXT *fname;
```

FUNCTION

mkexec is supposed to convert the file fname to executable form, typically by adding (or replacing) a system dependant suffix (or "extent") to fname, or setting some bits to mark it executable.

RETURNS

mkexec returns true unconditionally

EXAMPLE

```
if (load1() && load2())
    return (mkexec(xfile));
```

WARNINGS

On this system, it is a dummy.

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

WARNINGS
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 occurence of an interrupt generated from the keyboard of a controlling terminal. (Typing a delete DEL, or sometimes a ctrl-c, ETX, performs this service on many systems.)

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

RETURNS

Nothing.

NAME

open - open a file

SYNOPSIS

```
FILE open(fname, mode, rsize)
    TEXT *fname;
    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 update (reading and writing)

If (rsize == 0) the file is treated as a text-file, otherwise as a binary file. For some of the I/O-drivers the value of rsize may affect the actual record-length. In those cases, a default will be chosen if (rsize == 1).

RETURNS

open returns a file descriptor for the opened file or a negative number which is an error code described in <os.h>.

EXAMPLE

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

SEE ALSO

Files, 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.

RETURNS

If an error occurs, read returns a negative number which is an error code defined in <os.h>; if end-of-file is initially encountered, read returns zero; otherwise the value returned is the number of bytes read, which is between 1 and size inclusive.

EXAMPLE

To copy a file:

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

SEE ALSO

Files, write

NAME

remove - remove a file

SYNOPSIS

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

FUNCTION

remove deletes the fname from the file system. If no I/O driver is specified, **mem** is used under OS and **cms** under CMS.

RETURNS

remove returns zero, if successful, otherwise e negative number. For those I/O-drivers that don't support removal of files it returns ENOTIMPL.

EXAMPLE

```
if (remove("mem:t2") < 0)
    putstr(STDERR, "can't remove temp file\n", NULL);
```

NAME

sbreak - set system break

SYNOPSIS

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

FUNCTION

sbreak buys at least size bytes of memory from the system by issuing a GETMAIN supervisor call.

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 space!\n", NULL);
    exit(NO);
    }
```

NAME

uname - create a unique file name

SYNOPSIS

```
TEXT *uname()
```

FUNCTION

uname returns a pointer to the start of a NUL terminated "mem:"-filename which is likely not to conflict with normal user names. The name may be modified by a suffix of up to three letters, so that a family of files may be dealt with. The name may be used as the first argument to a subsequent create or open call, so long as any such files created are removed before program termination. If they aren't, _main will try to write them out via the "seq:"-driver. 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 "mem:t". 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.

RETURNS

If an error occurs, write returns a negative number which is an error code defined in <os.h>; 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

IO, read

NAME

xecl - execute a file with argument list

SYNOPSIS

```
COUNT xecl(fname, sin, sout, flags, s0, s1, ..., NULL)
    TEXT *fname;
    FILE sin, sout;
    COUNT flags;
    TEXT *s0, *s1, ...
```

FUNCTION

xecl invokes the program file fname, connecting its STDIN to sin and STDOUT to sout and passing it the string arguments s0, s1, ... xecl will invoke fname by issuing a LINK supervisor call and will thus wait until the command has completed. If (flags & 2) xecl will exit when the command has completed. In this case, of course, xecl will never return to the caller.

If sin is not equal to STDIN, or if sout is not equal to STDOUT, the file (sin or sout) is closed before xecl returns.

If fname cannot be invoked, xecl, and your program, will have an ABEND (ABnormal END).

RETURNS

If the command executed successfully, xecl returns YES, otherwise NO.

EXAMPLE

```
if (!xecl(pgm, STDIN, create(file, WRITE), 0, f1, f2, NULL))
    putstr(STDERR, pgm, " failed\n", NULL);
```

SEE ALSO

xecv, _cmpswt

NAME

xecv - execute a file with argument vector

SYNOPSIS

```
COUNT xecv(fname, sin, sout, flags, av)
    TEXT *fname;
    FILE sin, sout;
    COUNT flags;
    TEXT **av;
```

FUNCTION

xecv invokes the program file fname, connecting its STDIN to sin and STDOUT to sout and passing it the string arguments specified in the NULL terminated vector av. It otherwise behaves like xecl.

SEE ALSO

xecl, _cmpswt

NAME

xlate - translate a buffer of characters

SYNOPSIS

```
TEXT *xlate(table, buf, n)
    TEXT table[], *buf;
    BYTES n;
```

FUNCTION

xlate translates n characters starting at buf, by replacing each character in buf with the byte in table that the character indexes. table should be a 256 byte array containing the replacement characters.

There are two tables defined for easy translation between ASCII and EBCDIC:

_atoe - is a 256 byte array consisting of the corresponding EBCDIC character for each ASCII value.

_etoa - is a 256 byte array consisting of the corresponding ASCII character for each EBCDIC value.

RETURNS

the value returned is the start address of buf.

EXAMPLE

To translate an input buffer buf of length len from EBCDIC to ASCII:

```
xlate(_etoa, buf, len);
```

SEE ALSO

_atoe, _etoa